Réflexivité, aspects et composants pour l'ingénierie des intergiciels et des applications réparties

Lionel Seinturier

THÈSE D'HABILITATION DE L'UNIVERSITÉ PIERRE ET MARIE CURIE PARIS VI

Spécialité :

INFORMATIQUE

Présentée par

Lionel SEINTURIER

Sujet de la thèse

Réflexivité, aspects et composants pour l'ingénierie des intergiciels et des applications réparties

Soutenance le 13 décembre 2005, devant le jury composé de :

Théo D'HONDT	Professeur Vrije Universiteit Brussels	Rapporteur
Jean-Charles FABRE	Professeur à l'INP Toulouse	Rapporteur
Jean-Bernard STEFANI	Directeur de recherche INRIA	Rapporteur
Bertil FOLLIOT	Professeur à l'Université Paris 6	Directeur
Jean-Marc GEIB	Professeur à l'Université Lille 1	Examinateur
Eric GRESSIER-SOUDAN	Professeur au CNAM Paris	Examinateur
Jacques MALENFANT	Professeur à l'Université Paris 6	Examinateur

Remerciements

Je remercie sincèrement toutes les personnes qui ont contribué de près ou de loin à l'aboutissement de cette habilitation, en particulier :

Monsieur Théo D'Hondt, Professeur à la Vrije Universiteit Brussels, pour m'avoir fait l'honneur de rapporter ce document.

Monsieur Jean-Charles Fabre, Professeur à l'Institut National Polytechnique de Toulouse et Directeur de recherche CNRS au LAAS, pour l'intérêt qu'il a manifesté pour mes travaux.

Monsieur Jean-Bernard Stefani, Directeur de recherche à l'INRIA Rhône-Alpes, pour sa disponibilité et pour m'avoir accueilli, il y a quelques années de cela, dans le laboratoire Architecture des Systèmes Répartis qu'il dirigeait au CNET (France Telecom R&D).

Monsieur Jacques Malenfant, Professeur à l'Université Paris VI, pour l'intérêt qu'il manifeste aux travaux que je mène.

Monsieur Bertil Folliot, Professeur à l'Université Paris VI, pour m'avoir accueilli au sein de son groupe en tant que maître de conférences, et pour ses conseils et ses encouragements permanents.

Monsieur Jean-Marc Geib, Professeur à l'Université Lille 1, pour m'avoir accueilli en délégation au sens du projet INRIA Jacquard, ses encouragements permanents et la confiance qu'il manifeste à mon égard. J'espère pouvoir en être digne.

Monsieur Eric Gressier-Soudan, Professeur au CNAM à Paris, pour les nombreux conseils qu'il a su me donner dans le domaine de la recherche en informatique et dans celui de l'escalade, et enfin pour le temps qu'il a pris pour participer à ce jury.

Ces remerciements ne seraient pas complets sans penser aux membres des équipes qui me côtoient depuis de nombreuses années : ceux de l'équipe SRC depuis 1999 et ceux du projet Jacquard depuis 2003. Dans le thème SRC, je remercie tout particulièrement Pierre Sens et Philippe Darche pour leur gentillesse et leur sympathie, mais aussi Jean-Luc, Marie-Pierre, Claude, Xavier, Cédric, Emmanuel, Denis et Isabelle. Bien évidemment je réserve une mention toute particulière à Thierry Lanfroy pour son efficacité, Jacqueline Narboni pour ses compétences, et à Mahmoud Boufaïda pour nos (maintenant) nombreuses années de collaboration. Enfin, merci à Claude Girault, aiguillon permanent et au combien nécessaire, Fabrice Kordon et Anne Derieux pour m'avoir soutenu et accepté ma délégation, Marie-France Le Roch, Marc Cheminaud et Gérard Nowak pour m'avoir fait confiance un certain jour de 1998. Je n'en serai certainement pas là sans eux. Au sein du projet Jacquard, mes remerciements vont à Axelle, Alexis, Anne-Françoise, Areski, Bassem, Bernard, Gilles, Jérôme, Maja, Olivier, Olivier, Philippe, Rafaël, Romain pour leurs nombreuses heures de discussions passionnées et leur sympathie.

Une pensée pour tous les étudiants, doctorants et autres, qui ont eu à me subir : merci Dolores, Fabrice, Frédéric, Nicolas et les autres. Ce travail est aussi un peu le votre.

Mention spéciale à Laurence, Renaud, Gérard, Laurent et Jean-Philippe pour tout le travail accompli ensemble.

À ma famille et mes proches pour leur soutien constant et leur amour.

L. Seinturier

Résumé

Les intergiciels et les applications réparties se caractérisent par un nombre élevé de fonctionnalités à intégrer pour aboutir à un logiciel fini. Par ailleurs ils s'appliquent à de nombreux contextes d'exécution, de l'embarqué fortement contraint aux systèmes d'information ouverts sur l'Internet. Cette diversité engendre des besoins multiples allant, pour n'en citer que quelques uns, de la tolérance aux fautes, aux communications distantes ou à l'ordonnancement sous contrainte. Ce mémoire porte sur l'ingénierie des intergiciels et des applications réparties. Notre objectif est de proposer des solutions pour l'intégration des différentes fonctionnalités de ces logiciels. Pour cela, les études décrites dans ce mémoire portent sur trois techniques principales : la réflexivité, les aspects et les composants.

Dans un premier axe, nous abordons la conception et la réalisation de service pour des applications réparties CORBA à l'aide de la réflexivité. Nous avons proposé un service d'observation d'exécution d'exécutions réparties. Nous montrons que la réflexivité permet de séparer clairement les différentes préoccupations. Les applications à observer et le service sont clairement découplés. L'intégration peut alors être faite de façon transparente. Nous montrons que le coût des mécanismes réflexifs est faible par rapport aux coûts globaux de communication dans un environnement distribué CORBA.

Dans un deuxième axe, nous nous focalisons sur les techniques à base d'aspect. Nous présentons JAC (*Java Aspect Components*) qui est une plate-forme pour la programmation d'applications orientées aspect. JAC propose la notion de tissage dynamique qui permet d'adapter au cours de l'exécution une application en ajoutant ou en retirant des aspects. En terme d'intergiciel et de répartition, JAC apporte la notion de conteneur ouvert à objets et à aspects et la notion de coupe distribuée.

Finalement, nous abordons les techniques à base de composant. L'objectif est de fournir un cadre unifiant les styles de développement à base d'aspect et de composant. Une étude préliminaire compare le développement d'un service de partage de documents répliqués avec des frameworks à base de composants (Fractal et Kilim) et un framework AOP (JAC). Nous montrons les forces et les faiblesses de chacun des styles et en tirons des conclusions pour étendre le modèle de composants Fractal en lui adjoignant la notion d'aspect. Dans un second temps, nous montrons que les aspects peuvent également être bénéfiques à l'ingénierie du modèle de composant lui-même. Nous aboutissons ainsi à une solution dans laquelle les aspects contribuent à la fois à la mise en œuvre des mécanismes internes du modèle et à son interface de programmation externe.

Abstract

Middleware and distributed applications are characterized by a high number of functionalities which must be integrated. Furthermore, they deal with various execution contexts, from constrained embedded systems to information systems reachable through the Internet. This diversity generates many requirements such as, just to name a few, fault-tolerance, remote communications or scheduling. This document is about the software engineering of middleware and distributed

applications. Our goal is to propose solutions for integrating the functionalities needed in these contexts. This document reports on three main techniques which have been studied : reflection, aspects and components.

The first chapter deals with the design and the implementation of a CORBA service based on the notions of reflection. We propose a service for observing distributed runs. We show that reflection allows separating clearly the various concerns. The applications to be observed and the service are clearly decoupled. The integration can then be performed seamlessly. We show that the cost of reflection is low compared to the costs induced by distributed communications in a CORBA environment.

The second chapter studies aspect-oriented techniques. We present JAC (*Java Aspect Components*), a platform for programming aspect-oriented applications. JAC proposes the notion of dynamic weaving which allows adapting at runtime an application by adding or removing aspects. Concerning middleware and distribution, JAC introduces the notion of an open container for objects and aspects and the notion of a distributed pointcut.

Component-based software engineering is addressed in the third chapter. The goal is to propose a framework which unifies aspects and components. A preliminary study compares, with an application for sharing replicated documents, two component frameworks (Fractal and Kilim) and an AOP framework (JAC). We report on the strengths and the weaknesses of these three approaches and, based on this experience, we propose to extend the Fractal component model with the notion of an aspect. Next, we show that aspects can also improve the engineering of the component model. We then propose a solution where aspects are used both for the internal implementation of the component model and for the external application programming interface.

Table des matières

Table des figures

Liste des tableaux

Chapitre 1

Introduction

Sommaire

Les intergiciels (en anglais *middleware*) et les applications réparties modernes sont de plus en plus complexes. La raison en est certainement due au nombre sans cesse croissant de fonctionnalités à intégrer pour parvenir à un produit fini. En 25 ans, le domaine de l'informatique répartie est passé des librairies de communication asynchrone aux serveurs d'applications 3 tiers de type J2EE [20] ou .Net [87], en passant par les RPC [17] et les bus logiciel orientés objet de type CORBA [162] ou les grilles de calcul.

Cette évolution s'est accompagnée d'une croissance importante des fonctionnalités offertes. Par exemple, alors que le mécanisme de RPC gère essentiellement les communications, le nommage, la liaison et la représentation des données, un intergiciel comme CORBA, gère en plus des activités concurrentes, des politiques d'activation, et fournit des abstractions pour s'interfacer, en aval, avec les systèmes d'exploitation et, en amont, avec les langages de programmation. Les architectures 3 tiers à base de composants, comme J2EE, .Net ou CORBA CCM [162], apportent un degré de complexité supplémentaire en intégrant des préoccupations comme la persistance ou les transactions traditionnellement dévolues aux bases de données. De plus, de part leur forte ouverture sur l'Internet, ces intergiciels imposent des besoins importants en terme de sécurité. Par ailleurs, les applications construites pour ces intergiciels deviennent elles-mêmes plus complexes et intègrent de nombreux principes : frameworks MVC, Web Services, intercepteurs, aspects, etc.

La tendance est donc de fournir aux développeurs des paradigmes de haut niveau afin de rendre le développement plus efficace. On s'oriente maintenant vers l'intégration des notions issues du domaine des langages de description d'architectures (ADL). On ne s'intéresse pas uniquement à la description des entités interagissant, mais aussi à leurs schémas d'interaction, de connexion et de collaboration. Par ailleurs, cette évolution, qui a permis d'élever le degré d'abstraction des entités manipulées par les intergiciels, s'est accompagnée d'une complexification : la prise en charge d'un composant EJB demande une infrastructure infiniment plus lourde que celle nécessaire à une souche RPC. Une limite a certainement été atteinte et un phénomène inverse d'épuration des

concepts se met en place : des modèles de composants comme Fractal [27] ou OpenCOM [39], des conteneurs légers comme PicoContainer ou des frameworks de composants comme Spring [77] ont comme objectif de fournir des solutions performantes et pouvant fonctionner avec des ressources limitées. L'objectif est ici de faire en sorte que les composants ne soient pas cantonnés aux développements applicatifs de type système d'information, mais puissent adresser d'autres domaines comme l'embarqué, les logiciels d'infrastructure pour les telecoms ou les intergiciels eux-mêmes.

Les concepts fournis aux développeurs sont de plus en plus évolués et abstraits dans le but de masquer les détails parfois nombreux et complexes, de l'intergiciel. D'autres techniques, comme la réflexivité [163, 164] ou les aspects [84], participent à ce phénomène d'élévation du degré d'abstraction. Pour la réflexivité, il s'agit de séparer la logique métier du niveau de contrôle et d'interprétation des programmes. Pour les aspects, il s'agit d'introduire différents niveaux de modularisation en rassemblant dans un aspect des éléments de code participant à une même logique mais dispersés dans divers objets ou composants. Ce principe est résumé par la formule : "un aspect modularise une fonctionnalité transversale".

Cette évolution s'est accompagnée d'un souci constant de transparence et d'ouverture. La transparence permet l'interfaçage avec les langages de programmation et les systèmes d'exploitation. Il s'agit de ne pas être lié à un langage particulier et de pouvoir déployer l'intergiciel sur différents systèmes d'exploitation. L'ouverture se manifeste de différentes façons : mécanisme de souches et squelettes dynamiques (DII/DSI) pour la découverte et l'invocation dynamique de services, protocole GIOP pour l'interopérabilité inter-ORB, framework d'extension du protocole de transport (*Extensible Transport Framework*), mécanisme d'interception (*Portable Interceptor*) pour la capture transparente des communications. Les frameworks de programmation orientée aspect comme JAC [134] (voir chapitre 3, section 3.3), JBoss AOP, AspectWerkz ou Spring AOP se placent également dans cette mouvance en fournissant par exemple, dans le cas de JBoss AOP, un accès transparents aux services de l'intergiciel via des aspects.

Ce mémoire présente une synthèse des travaux de recherche que j'ai effectué depuis 1999. Ils sont centrés sur l'**ingénierie logicielle** appliquée aux intergiciels et aux applications réparties. La difficulté réside dans l'**intégration** des nombreuses fonctionnalités à prendre en compte pour obtenir un logiciel fini. Je me suis attaché à proposer de nouveaux paradigmes, de nouveaux traits méthodologiques et de nouvelles techniques pour faciliter la conception et l'implémentation des intergiciels et de leurs applications. Ces recherches ont été menées depuis 1999 au sein du thème "Systèmes Répartis et Coopératifs" (SRC) du LIP6 à l'Université Pierre & Marie Curie (Paris 6), et simultanément, à partir de septembre 2003, dans le projet Jacquard de l'INRIA Futurs à Lille.

Tendances

La **complexification** des intergiciels et de leurs applications citée précédemment est une tendance lourde qui ne donne pas de signe de ralentissement. Les nouveaux domaines, comme les architectures pair à pair ou l'informatique ubiquitaire, apportent leurs lots de nouveaux besoins en termes de découverte de services, de conscience du contexte (*context-awareness*) ou de gestion de la mobilité et des déconnexions/reconnexions.

Cette complexification a comme corollaire qu'une équipe ne peut plus à elle seule développer un projet de A à Z. Il devient obligatoire de prévoir dès le départ la **réutilisation** et l'intégration de briques logicielles existantes. On rejoint là la tendance dite COTS (*Commercial Off-The-Shelf*)

qui consiste à construire un système ou une application par assemblage de briques existantes.

Une troisième tendance lourde est celle de l'**interopérabilité**. Un système ou une application ne s'envisage pas en autarcie, mais doit être ouvert et pouvoir interagir avec d'autres systèmes ou applications. Le développement des technologies d'interopérabilité de type CORBA/IIOP, de type Web Services, ou les travaux sur les intergiciels multi-personnalités (par exemple, Jonathan [50] ou PolyORB [127]) illustrent ce besoin.

Finalement, l'**adaptabilité** et la **flexibilité** font également partie des besoins des intergiciels actuels. Elles peuvent être dynamiques et s'opérer à chaud alors que le système ou l'application s'exécute. Il s'agit là de répondre à des changements dans le contexte d'exécution et d'ajouter, retirer ou modifier des services. Cette adaptabilité peut également être statique afin de décliner une même base de code en une gamme de produits. Chaque élément de la gamme présente plus ou moins de fonctionnalités, et est donc adapté à différents environnements.

Démarche

Face à ces quatre tendances lourdes, complexification, réutilisation, interopérabilité et adaptabilité, le monde de la recherche sur l'ingénierie logicielle a répondu avec un foisonnement de propositions : objet, réflexivité, composant, aspect, langage dynamique, framework, *design pattern*, conteneur léger à inversion de contrôle, ingénierie dirigée par les modèles (MDE), métamodélisation, transformation, langage dédié (DSL), architecture orientée services (SOA), etc. Si ce foisonnement est la preuve d'une certaine vitalité, il n'en est pas moins déroutant pour le nouveau venu ou pour le chef de projet devant faire des choix technologiques. Ces quatre tendances pourraient être complétées par d'autres besoins qui sont autant de directions de recherche à part entière dans le domaine des intergiciels : passage à l'échelle, haute performance, sûreté de fonctionnement, formalisation des concepts.

Loin de pouvoir embrasser l'ensemble de ces approches, la démarche que j'ai suivie au cours de mes recherches a été d'essayer de répondre à la question de l'**intégration** : quels sont les concepts proposés aux concepteurs pour réaliser l'intégration de leurs fonctionnalités. Cette démarche est passée successivement par différentes étapes : réflexivité, aspect, composant. À chaque étape je me suis attaché à la mise en œuvre concrète des concepts proposés. Ainsi chaque expérience résumée dans ce mémoire fait l'objet d'une implémentation réalisée en collaboration avec des stagiaires ou thésards. Cette démarche a été influencée par les langages de programmation tout en conservant comme domaine cible les intergiciels et leurs applications.

Les sections suivantes introduisent les trois domaines principaux, réflexivité, aspects et composants, qui sous-tendent mon travail. La section 1.4 développe le plan de ce mémoire.

1.1 Réflexivité

La réflexivité [163, 164] désigne la capacité d'un système à raisonner et à agir sur lui même [103]. Les systèmes réflexifs distinguent deux niveaux : le niveau de base qui correspond à l'application et le niveau méta qui fournit une couche d'interprétation et de contrôle du niveau de base.

De nombreux travaux envisagent l'utilisation de la réflexivité pour le développement de systèmes ou d'intergiciels. On peut citer notamment les projets OpenCorba [92, 93], OpenORB [18, 42] ou Apertos [181]. Par ailleurs, la réflexivité a également été mise en œuvre pour le développement

de services pour les intergiciels. On peut citer par exemple, la migration [108][121][181][109], la réplication [59][62][51], la persistance [59], la coordination [108][24][109][59][157, 158] ou la tolérance aux pannes [51].

L'intérêt pour la réflexivité réside dans sa capacité à découpler, dans une application, la logique de base, du niveau de contrôle et de supervision offert par la couche méta. Il est ainsi possible d'y implémenter des fonctionnalités système. En séparant les fonctionnalités système de la logique de base, on compte obtenir une application plus modulaire, plus claire, plus facilement maintenable et développable plus rapidement.

On retrouve là une préoccupation adressée également par les serveurs d'applications de type J2EE, CORBA CCM ou .Net dans lesquels des conteneurs offrent des services aux composants, par la programmation orientée aspect [84] (voir chapitre 3) ou par les modèles de composant comme Fractal [27] (voir chapitre 4, section 4.1.4) avec la notion de contrôleur. On retrouve également ce principe sous une autre forme dans l'ingénierie dirigée par les modèles (MDE) avec les notions de modèles indépendants et spécifiques aux plates-formes (PIM et PSM). Dans tous les cas, il s'agit de séparer la logique métier des propriétés non fonctionnelles ou système, de concevoir les deux séparément, puis d'envisager leur intégration. Si cette démarche simplifie dans un premier temps la conception et la mise en œuvre, elle fait peser une difficulté non négligeable sur la phase d'intégration.

En ce qui concerne la réflexivité, l'intégration entre les deux niveaux, base et méta, passe par les événements du niveau de base réifiés au niveau méta. Le mécanisme de réification rend explicite et manipulable des événements liés aux programmes eux-mêmes (par exemple appels de méthode, structures de contrôle) ou au fonctionnement interne du langage (par exemple gestion de la file d'attente des messages ou gestion des activités concurrentes). En permettant d'écrire des méta-programmes qui redéfinissent les comportements par défaut attachés à ces événements, la réflexivité permet d'intégrer de nouveaux comportements dans les applications.

Dans ce mémoire, nous avons utilisé la réflexivité pour définir et implémenter un service CORBA d'observation d'exécutions réparties (chapitre 2). Ce service, comme les services CORBA "traditionnels" (nommage, événements, transaction, etc.) possède des interfaces spécifiées en IDL. Son implémentation peut ainsi être réalisée avec un langage quelconque (en l'occurrence Java). Par contre, alors que les services "traditionnels" doivent être utilisés explicitement par les applications, la réflexivité permet d'intégrer de façon transparente ce service dans l'application.

1.2 Aspects

La programmation orientée aspect (AOP pour *Aspect-Oriented Programming*) [84] est un paradigme apparu au milieu des années 1990 issu des travaux menés au Xerox PARC dans l'équipe de Gregor Kiczales.

L'AOP part du constat que les logiciels complexes ne sont jamais linéaires et comprennent de nombreuses préoccupations entremêlées les unes dans les autres. Un des exemples fondateurs de l'AOP est celui du serveur de servlet Tomcat. G. Kiczales et son équipe ont montré que dans ce logiciel, écrit en Java, certaines fonctionnalités sont clairement modularisées dans une ou deux classes, tandis que d'autres, comme la gestion des sessions utilisateurs, étaient dispersées dans de nombreuses classes. Ce phénomène de dispersion (en anglais, le code est dit *scattered*) nuit à la évolutivité, la maintenabilité et à l'efficacité du code. Dans d'autres cas, il a été montré

qu'une même classe peut avoir à gérer de nombreuses préoccupations simultanément. C'est le cas par exemple, des applications J2EE utilisant des composants EJB dans lesquelles une même méthode de transfert bancaire peut avoir à gérer de la sécurité, des communications distantes, des transactions et de la persistance, en plus du code lié au transfert proprement dit. Dans ce cas, le code est dit entremêlé (en anglais *tangled*).

Si ces deux phénomènes, code dispersé et code entremêlé, sont clairement identifiés, leurs causes sont plus subtiles. Une première approche consisterait à dire qu'ils sont dus à une mauvaise conception, et qu'une meilleure analyse du problème permettrait de les faire disparaître. En fait, il apparaît que le processus qui aboutit à une décomposition en classes, ou en fonctions ou en procédures, est souvent influencé par une préoccupation particulière. Par exemple, dans les applications objet, on conçoit souvent les classes en fonction des données que l'on veut y encapsuler. Partant de cette décomposition influencée par les données, les autres préoccupations (sécurité, transaction, etc.) sont alors contraintes de s'insérer à l'intérieur de ce découpage initial, ce qui conduit à de l'entremêlage et de la dispersion. La démarche prônée par l'AOP est, plutôt que d'essayer vainement de faire disparaître ces phénomènes avec les outils existants, de proposer un autre concept, celui d'aspect, pour modulariser ces préoccupations dites transversales, c'est-à-dire qui, sans les aspects, seraient entremêlées ou dispersées.

Les aspects introduisent donc une dimension de modularisation supplémentaire dans les programmes. Ils ne remplacent pas les objets, ou les fonctions ou les procédures, mais les complètent en prenant en charge les fonctionnalités transversales. De nombreuses propositions ont vu le jour en Java, ou dans d'autres langages, pour supporter les aspects. Elles ont abouti soit à des compilateurs, par exemple AspectJ [83], soit à des frameworks, par exemple JAC [134] (voir chapitre 3, section 3.3), AspectWerkz, JBoss AOP ou Spring AOP. Cette dernière catégorie a de plus apporté à l'AOP la notion de dynamicité : le lien entre le code et les aspects n'est plus statique comme dans l'approche compilateur, mais peut être modifié à l'exécution. Au delà du monde des langages, les aspects ont été rapidement adoptés par la communauté des intergiciels qui y a vu un débouché pour gérer la complexité inhérente au domaine, et pour apporter des propriétés d'adaptabilité au contexte en ajoutant ou retirant dynamiquement des aspects.

Les aspects et la réflexivité partagent de nombreux points communs. Plusieurs équipes de recherche ont, comme nous, évolué de la réflexivité vers l'AOP. En effet, dans les deux cas, il s'agit d'isoler la logique métier pour faciliter son évolution et son développement. Par contre, les deux approches diffèrent dans la façon dont la séparation est abordée. La réflexivité propose, avec la réification, une solution centrée sur les mécanismes du langage de programmation. Elle contraint le développeur à réfléchir en terme de transformation : il s'agit de savoir comment les mécanismes de base du langage sont modifiés par telle ou telle préoccupation technique. Dans le cas de l'AOP, l'approche est plus centrée sur le métier : il s'agit d'identifier les fonctionnalités transversales. De plus, l'AOP offre, avec la notion de langage de coupe, un moyen plus riche pour décrire l'intégration entre le code métier et les fonctionnalités ajoutées. Au final, l'orientation plus métier et une meilleure solution en terme d'intégration font que les aspects semblent plus prometteurs que la réflexivité. Il n'en reste pas moins, que de manière interne, la plupart des langages ou des frameworks AOP sont mis en œuvre à l'aide de la réflexivité.

Dans ce mémoire, l'AOP est la composante majeure de nos travaux sur la plate-forme JAC (chapitre 3, section 3.3). JAC a été l'une des premières plates-formes internationales à mettre en œuvre le concept d'aspects dynamiques, pouvant être ajoutés et retirés lors de l'exécution. Outre

la dynamicité, les contributions majeures de ce projet ont été de proposer la notion de coupe distribuée pour la programmation d'applications réparties orientées aspect et d'offrir un mécanisme de configuration pour faciliter la réutilisation de librairies d'aspects. Nous présentons également nos travaux sur l'utilisation des aspects pour la simplification du développement des applications pour les plates-formes à base de composants répartis EJB et CCM (chapitre 3, section 3.4). Finalement, le chapitre 4 propose une unification des concepts de composants et d'aspects avec le modèle Fractal Aspect Component (FAC).

1.3 Composants

La notion de composant logiciel vise à favoriser la réutilisabilité du code. Son principe se veut similaire à celui de composant électronique : il s'agit de développer une brique logicielle qui puisse être connectée à d'autres composants afin de construire un système, en l'occurrence un logiciel, plus complexe. Le notion de composant [111] est apparu en 1968. Elle a connu un regain d'intérêt dans la seconde moitié des années 1990 avec les modèles de composants industriels EJB de Sun, CCM de l'OMG et .Net de Microsoft. On peut noter que, comme pour les aspects, le domaine des intergiciel a joué un rôle important dans la promotion de la notion de composant.

Au delà de ce principe général, la notion de composant est très diverse. De nombreux modèles de composants ont été proposés pour différents domaines d'applications, du temps réel pour l'embarqué aux systèmes d'informations. Selon les auteurs, le terme composant peut aussi bien correspondre à un fichier C externalisant des propriétés, à une entité complexe dont la structure et la sémantique sont définies par un méta-modèle et nécessitant une structure d'accueil (conteneur) pour l'exécuter. Cet état de fait est reflété par le nombre important de définitions associé à la notion de composant. Par exemple, [168] en recense neuf, la première étant due à Grady Booch en 1987 [23]. Dans les dernières années, il semblerait néanmoins que le paysage s'éclaircisse avec deux définitions qui émergent : [168] et [69]. Nous y reviendrons au chapitre 4. Ces deux définitions mettent en avant le fait qu'un composant fournit et requiert des services, et qu'il est une unité de déploiement et de composition.

Ce dernier point, la composition, fait que les composants promeuvent une démarche de programmation par assemblage. La démarche dite COTS (*Commercial Off-The-Shelf*) visant à construire l'application à l'aide de composants existants et/ou venant de parties tierces, est l'aboutissement d'une telle logique. Il s'agit de construire des schémas d'interconnexions afin de mettre en commun les fonctionnalités individuelles des composants. La notion de composant est fortement liée à celle d'architecture logicielle. Cette dernière fait l'objet d'un domaine à part entière : les langages de description d'architectures (ADL). Il s'agit de capturer et d'exprimer les schémas de connexion dans des langages dédiés. On exprime, à la manière d'un plan en architecture "traditionnelle", la structure de l'application. Notons que le domaine des composants et celui des ADL sont fortement liés : la plupart des ADL se basent sur des composants, et de nombreux modèles de composants proposent un ADL.

Dans ce mémoire, la notion de composant est au cœur des travaux présentés au chapitre 4. Nous y présentons trois expériences dont le dénominateur commun est le modèle de composants Fractal [27]. L'objectif final est d'aboutir une unification des notions d'aspects et de composants. Nous pensons effectivement que ces deux notions sont fortement complémentaires au sens où les composants permettent de structurer la partie métier des applications, tandis que les aspects

apportent une dimension supplémentaire dans la structuration des fonctionnalités transversales comme les services non fonctionnels. Une unification permettra d'aboutir à un modèle de développement prenant en compte ces deux dimensions de façon efficace. La section 4.3.1 jette les premières bases de cette unification avec le modèle Fractal Aspect Component (FAC). Cette proposition fait suite à une étude préliminaire (section 4.2) dans laquelle nous avons comparé deux modèles de composants, Fractal et Kilim, et un modèle d'aspect, JAC.

1.4 Plan du mémoire

Ce mémoire comprend trois chapitres reprenant les problématiques exposées précédemment. Ces trois chapitres suivent l'ordre chronologique de mes activités de recherche.

Le chapitre 2 présente un bilan de mes activités sur l'utilisation de la réflexivité en vue de la construction de services pour les intergiciels. J'y résume la conception et la réalisation d'un service d'observation d'exécution répartie pour des applications CORBA (section 2.2). Cette observation est réalisée à l'aide de méta-programmes OpenJava [171]. Cette expérience est précédée de brefs états de l'état sur CORBA (section 2.2.1) et la réflexivité (section 2.2.2).

Le chapitre 3 présente un bilan de mes activités sur les aspects. Les résultats de deux projets sont présentés : la plate-forme à aspects dynamiques JAC (section 3.3) et une couche abstraite à base d'aspects pour le développement d'applications EJB et CORBA CCM (section 3.4). Cette couche abstraite a été conçu et réalisée dans le cadre de la thèse de Fabrice Legond-Aubry [94]. Elle réifie un ensemble d'événements concernant les interactions entre la plate-forme et une application (recherche de noms, résolution de la fabrique, recherche/création d'instances de composants, etc.). Elle matérialise ces concepts dans des aspects et permet de développer de nouveaux services. Ces services peuvent alors être utilisé indifféremment avec l'une des deux technologies en sélectionnant la personnalité adéquate (EJB ou CCM) de la couche abstraite.

Le chapitre 4 porte sur les composants. Il commence par un rappel des définitions de base sur les composants (section 4.1.1), les architectures logicielles (section 4.1.2) et quelques modèles de composants (section 4.1.4) : OSGi, Fractal, Kilim, OpenCOM, K-Component, Comet. Nous présentons à la section 4.2 une expérience préliminaire portant sur la comparaison de deux modèles de composants, Fractal et Kilim, et d'une plate-forme AOP, JAC. Cette expérience a été portée par F. Loiret dans le cadre de son stage de DEA [97]. Elle a permis de concevoir et implémenter un service de partage de documents répliqués avec les trois approches. Elle a donné lieu à une comparaison des concepts, et a permis de jeter les bases des travaux suivants qui ont pour objectif d'unifier les concepts des composants et des aspects. Cet objectif est développé à la section 4.3.1 avec le modèle Fractal Aspect Component (FAC). Cette partie fait l'objet de la thèse de N. Pessemier qui a débuté en 2004 en collaboration avec France Telecom R&D. Finalement, la section 4.3.2 présente une seconde expérience de fusion entre aspects et composants. Il s'agit de la réalisation d'un modèle de composants, Fractal, à l'aide d'aspects. L'objectif de cette expérience est de montrer l'intérêt de l'AOP pour la conception et la réalisation des politiques de contrôles des composants.

Le chapitre 5 clot ce mémoire et présente nos conclusions et des perspectives de travaux futurs.

Encadrements et publications majeurs

Les travaux présentés dans ce mémoire ont donné lieu à plusieurs encadrements et publications. L'annexe A présente plus en détail ces éléments.

Les travaux présentés dans le chapitre 2 ont fait l'objet d'un article [49] dans la revue *International Journal of Parallel and Distributed Systems and Networks* et de plusieurs publications dans des conférences internationales.

Les travaux sur les aspects traités dans le chapitre 3 ont fait l'objet de deux thèses soutenues, R. Pawlak [128] et F. Legond-Aubry [94], que j'ai encadré. La thèse de R. Pawlak porte sur la plate-forme JAC et sur la définition d'un calcul pour vérifier des propriétés sur l'ordonnancement de l'exécution des aspects. La thèse de F. Legond-Aubry porte sur la définition d'une couche neutre orientée aspect pour la programmation d'applications à base de composants EJB et CCM et sur la définition de contrats pour les composants. Une troisième thèse, D. Diaz, est en cours depuis janvier 2004 et porte sur la définition d'un aspect de traçabilité pour faciliter l'évolution des applications. Trois DEA (G. Haïk, T. Garcia-Fernandez, L. Teboul) ont également été encadrés sur le domaine des aspects. En terme de publications, un ouvrage de synthèse a été publié sur l'AOP et sur son utilisation pour le développement d'applications J2EE. Cette ouvrage a fait l'objet de deux éditions : une en français [148] en mai 2004 (Eyrolles) et une en anglais [149] en septembre 2005 (APress). La version anglaise est une traduction à 75% : un nouveau chapitre a été ajouté et les anciens chapitres ont été mis à jour avec les nouveautés apparues entre temps. Une publication [134] dans la revue *Software Practise and Experience* et un chapitre [132] d'un ouvrage collectif présentent un bilan du projet JAC. Plusieurs autres publications dans des conférences internationales présentent nos travaux portant sur l'AOP.

Le thème des composants abordé dans le chapitre 4 fait l'objet de deux thèses en cours, N. Pessemier et F. Loiret, dont j'assure l'encadrement. La thèse de N. Pessemier se déroule dans le cadre d'un contrat avec France Telecom R&D. Elle porte sur l'extension du modèle de composants Fractal avec des notions issues de l'AOP. La thèse de F. Loiret fait l'objet d'une collaboration avec le CEA. Elle porte sur la définition d'un modèle de composants pour le temps-réel. J'ai encadré les stages de DEA de ces deux doctorants.

Chapitre 2

Construction d'un service réflexif pour CORBA

Sommaire

2.1 Problématique

Les années 1990 ont été marquées par une intense activité de recherche et de développement autour des intergiciels (plus connus en anglais sous le terme *middleware*). Plusieurs propositions majeures ont émergé : CORBA, Java RMI, DCOM, .Net, SOAP. Leur objectif commun est d'assurer une prise en charge de l'**hétérogénéité** des systèmes d'exploitation (CORBA, SOAP) et des langages de programmation (tout ceux cités sauf Java RMI). Il s'agit d'assurer une interopérabilité au niveau applicatif, en permettant à des logiciels de communiquer de façon aussi transparente que possible au travers d'un réseau.

Ces intergiciels ont tous plus ou moins développé la métaphore du **bus logiciel** : par analogie avec le matériel, il s'agit de fournir un substrat sur lequel viennent se connecter des entités logicielles (par ex. objets, composants, procédures). Ce substrat s'insère entre les applications et le système (d'où le terme de *middleware*). Il fournit aux applicatifs une abstraction (API, souches de

communication) leur permettant de communiquer avec un haut niveau d'abstraction (par ex. invocation de méthode). Il assure la gestion des mécanismes réseau (protocole, encodage de données) et système (réservation de ressources, gestion mémoire, gestion d'activité).

En terme de recherche, de nombreux besoin ont été identifiés dans le domaine des intergiciels. Pour n'en citer que quelques uns parmi les plus significatifs : hautes performances, large échelle, fiabilité, tolérance aux fautes, flexibilité, adaptabilité. Une caractéristique importante de ces problématiques est en général de ne pas être seulement propres aux intergiciels mais d'adresser aussi des domaines connexes comme les systèmes, les réseaux et le génie logiciel.

Parmi toutes ces besoins, mon travail m'a amené à développer plus particulièrement celui de l'**ingénierie logicielle**. Il s'agit de trouver des paradigmes pour gérer de façon efficace le développement des intergiciels. En d'autres termes, il s'agit de faire en sorte que ceux-ci ne soient plus développés de façon complètement ad-hoc, mais que des principes généraux puissent être appliqués et que des briques de base puissent être réutilisées autant que faire se peut.

De façon macroscopique, on peut distinguer (voir figure 2.1) l'ingénierie des intergiciels (i.e. comment l'intergiciel est construit), de l'ingénierie de l'intégration d'un intergiciel avec ses applications (i.e. comment l'application est liée à son intergiciel), de l'ingénierie de l'application elle-même. Ces points sont détaillés dans les sections suivantes. Les bénéfices attendus de ces ingénieries sont d'aboutir à des logiciels clairs, exempts autant que faire se peut de bugs, efficaces, facilement maintenables et pouvant évoluer. Mon travail a particulièrement porté sur l'ingénierie de l'intégration d'un intergiciel avec ses applications.

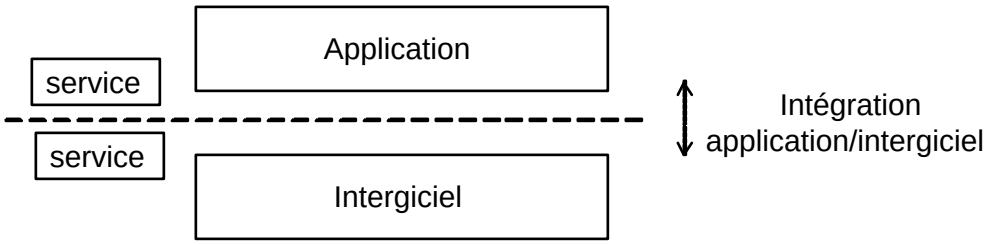

FIG. 2.1 – Ingéniere des intergiciels et de leurs applications.

2.1.1 Ingénierie des intergiciels

Du fait de leur complexité, les intergiciels sont des logiciels de taille conséquente. À titre d'exemples, JacORB [26], un intergiciel CORBA *open source* contient environ 1400 classes Java totalisant plus de 220 000 lignes de code (sans pour autant que la totalité des services définis dans les spécifications CORBA ne soit implémentée). De même, OpenCCM [105], l'implémentation du modèle de composants CORBA CCM totalise plus de 700 000 lignes de code.

La construction de logiciels d'une telle taille ne peut être empirique et nécessite des principes. Cette problématique est au cœur des recherches de nombreuses équipes qui cherchent à proposer des techniques novatrices pour la construction de ces intergiciels. Ce phénomène entraîne une co-évolution des domaines des intergiciels et du génie logiciel, le premier fournissant un cadre applicatif significatif au second, et le second permettant d'améliorer la qualité du premier. Parmi les techniques de génie logiciel mises en œuvre pour la construction des intergiciels, on peut citer les patrons de conception, les frameworks, la réflexivité, les aspects et les composants. Nous évoquons

ici les deux premières techniques. Les suivantes font l'objet respectivement de la section 2.2.1 et des chapitres 3 et 4.

Les **patrons de conception** (*design patterns*) sont des éléments de solutions réutilisables pour des problèmes informatiques récurrents. Les patrons ont été popularisés par Gamma et al. [58] et de nombreuses études ont cherché à les appliquer à différents domaines de l'informatique.

Dans le domaine des intergiciels, TAO [153] est certainement celui qui a intégré le plus de patrons dans l'architecture d'un ORB. À titre d'exemple, les patrons suivants sont utilisés dans TAO : façade pour encapsuler dans des classes des API système non orientées objet, réacteur pour traiter au niveau du serveur et de façon asynchrone les événements concurrents émis par les clients, accepteur/connecteur pour découpler la phase d'établissement de connexion de la phase de traitement, leader/suiveur pour gérer les *threads*, stratégie pour configurer les différentes politiques de concurrence, communication, ordonnancement et multiplexage utilisées dans l'ORB, fabrique dès qu'il s'agit de créer des instances et configurateur de composant pour pouvoir changer l'implémentation d'un "composant" (à prendre ici au sens le plus général du terme, i.e. entité logicielle, sans que cela fasse référence à un modèle de composants précis - EJB, CCM, Fractal, ou autre) sans recompiler l'application.

L'utilisation de patrons aboutit à une architecture d'ORB plus claire et structurée, elle documente le travail des concepteurs en permettant d'appréhender plus facilement les mécanismes utilisés et offre un meilleur degré de configurabilité en permettant d'étendre plus facilement l'ORB, par exemple par l'ajout de stratégies. Finalement, les patrons apportent un gain certain en termes d'évolutivité et de maintenabilité du code.

Les frameworks Bien qu'extérieurement les intergiciels existants soient très différents, ils s'appuient tous plus ou moins sur les mêmes mécanismes systèmes et réseaux pour l'acheminement et le traitement des requêtes. L'idée est alors de fournir un socle commun au-dessus duquel différentes versions, appelées souvent personnalités, seront construites. Nous mentionnons ci-dessous deux exemples d'intergiciels ayant mis en œuvre ce principe.

Jonathan [50] reprend les concepts généraux du modèle ODP [75] (*Open Distributed Processing*) et place la notion de liaison et d'objets de liaison au cœur de l'architecture du framework. Un objet de liaison, que l'on trouve également dans d'autres travaux comme [160], est un ensemble quelconque d'objets qui mettent en œuvre un chemin de communication (localement ou à distance) entre deux entités. Le patron fabrique présent dans Jonathan permet de systématiser la création de liaisons et offre au développeur la possibilité d'ajouter ses propres liaisons. Deux personnalités, David et Jérémie, sont présentes dans Jonathan et mettent en œuvre respectivement, les spécifications CORBA et RMI. Le concept de liaison est suffisamment souple et ouvert pour permettre l'incorporation de divers protocoles réseaux comme les protocoles AAL/5 pour ATM [156] ou multicast IP [50].

Sous le terme intergiciel schizophrène, Pautet [127] propose avec PolyORB, un socle commun pour la construction de différentes personnalités d'ORB. Les services fournis pour le noyau de schizophrénie permettent de gérer le référencement d'entités distantes, le transport, la liaison, la représentation des données, les protocoles, l'activation et l'aiguillage. Ces sept services de base ont été déclinés pour implémenter des personnalités CORBA, DSA, SOAP et JMS (Java Messaging Service).

De façon moins ambitieuse, les ORB du marché proposent aussi des mécanismes d'extension qui

permettent d'en personnaliser le fonctionnement, notamment au travers de différentes personnalités protocolaires. C'est notamment le cas du framework ETF (*Extensible Transport Framework*) de CORBA et des notions de *channels* (transport d'information) et de *formatter* (encodage de données) présentes dans .Net Remoting [144].

L'utilisation d'un framework permet d'abstraire les concepts fondamentaux des intergiciels. Ceux-ci ne sont alors plus noyé dans la masse mais apparaissent clairement dans l'architecture. La compréhension du fonctionnement global en est facilité. La séparation des concepts fondamentaux et de leur implémentation rend plus aisé le développement d'un nouvel intergiciel ou l'adaptation d'un intergiciel existant à un nouveau contexte.

2.1.2 Ingénierie de l'intégration application/intergiciel

L'intégration de nouveaux services peut être réalisée de différentes manières. Par rapport à une frontière (parfois floue) entre applications et intergiciels, les services peuvent être ajoutés dans la couche application ou dans la couche intergiciel.

Par ailleurs cette intégration a toujours été un problème compliqué. Cette difficulté s'explique par le nombre important de mécanismes à mettre en œuvre (protocoles de communication, représentation des données, gestion du nommage, établissement de liaisons, activation, etc.) et par les services parfois complexes que les développeurs souhaitent mettre en place pour faciliter le développement et tirer pleinement parti d'un environnement distribué (équilibrage de charge, tolérance aux fautes, migration, cohérence de données, etc.). Plusieurs approches peuvent être adoptées pour mener à bien cette tâche. Dans le cadre des langages orientés objet, Briot [25] et ses co-auteurs en ont identifié trois.

L'approche par librairie Les développeurs disposent de librairies (d'envoi de message, d'appel de procédure distante, de diffusion sur groupe, etc.) qu'ils peuvent lier à leurs programmes. Cette approche est très répandue mais peut être lourde à utiliser puisqu'elle nécessite de connaître l'ensemble des primitives de la librairie, leurs profils d'appel ainsi que leurs sémantiques. C'est le cas des intergiciels classiques comme CORBA où les services (*COS Services*) ont une API définie en IDL indépendante de leur implémentation.

L'approche intégrée Elle consiste à étendre un langage de programmation afin de lui intégrer des caractéristiques liées à la répartition. Il est alors nécessaire de connaître la sémantique de ces caractéristiques pour obtenir le comportement voulu. Cette approche a été adoptée par les systèmes répartis du début des années 1990 comme GUIDE [10], DOWL [2], COOL [88], Choices [31] ou SOS [161]. Cette approche est en générale simple d'utilisation. Par contre, le fait que le code implémentant la répartition soit enfoui dans le cœur du système, il est difficile d'adapter ou de changer le code lié à la répartition.

L'approche réflexive Cette approche, sur laquelle nous reviendrons à la section 2.2.2, consiste à introduire deux niveaux : le niveau de base qui comprend le programme proprement dit et un niveau dit méta de contrôle et de supervision du niveau de base. Certains mécanismes du niveau de base (gestion des messages, des activités, etc.) sont rendus accessibles au niveau méta (on dit qu'ils sont réifiés). Un méta-programme peut alors modifier ces mécanismes pour y intégrer des

propriétés ou des services supplémentaires. Cette approche apporte une certaine transparence en permettant de changer facilement le comportement des services implémentant la répartition.

À ces trois approches, on peut en ajouter une quatrième qui est l'**approche par aspects**. L'application se compose d'un cœur métier autour duquel viennent se greffer des services programmés sous forme d'aspects. Comme dans l'approche réflexive les aspects réifient des mécanismes de base des langages de programmation (appel, exécution de méthodes, lecture, écriture d'attributs, etc.) et définissent de nouveaux comportements à entreprendre. La notion de coupe différentie nettement les approches par aspects des approches réflexives. La coupe définit l'ensemble des localisations du programme de base auxquelles l'aspect intervient. Cette facilité ouvre plus de perspectives que l'approche réflexive mais introduit incidemment un problème de composition lorsque plusieurs aspects interviennent sur la même localisation. Les solutions actuelles offrent des règles, locales ou globales, pour définir l'ordre d'exécution des aspects lorsqu'un tel cas se présente.

La section 2.3 présente mes travaux sur l'utilisation de la réflexivité pour le développement d'un service CORBA d'observation d'exécutions réparties. C'est une mise en œuvre de l'approche réflexive. Il s'agit donc d'introduire un nouveau service dans la couche applicative.

En dehors des travaux présentés dans ce mémoire, nous avons eu l'occasion de mettre en œuvre l'approche intégrée via la conception et l'implémentation d'un service de réplication de composants EJB [20] pour le serveur J2EE JOnAS. Ce travail a été réalisé dans le cadre du projet RNTL IMPACT (2001-2003) qui a permis de mettre en place un ensemble de briques logicielles pour les intergiciels. Cette expérience a donné lieu au stage de DEA de K. Thini [175] dont j'ai assuré l'encadrement avec P. Sens. Par ailleurs, nous présentons au chapitre 3 les travaux réalisés autour de l'intergiciel orienté aspect JAC. Ceux-ci illustrent la mise en œuvre de l'approche par aspects.

2.2 Intergiciel CORBA et réflexivité

Cette section propose un bref état de l'art sur CORBA (section 2.2.1) et la réflexivité (section 2.2.2). La section 2.3 résume le travail effectué sur l'utilisation de la réflexivité pour l'intégration d'un service d'observation d'exécutions réparties dans CORBA.

2.2.1 CORBA

CORBA [162] (*Common Object Request Broker Architecture*) est un standard international de l'OMG (*Object Management Group*) spécifiant une architecture d'interopérabilité entre objets distants. En plus de 15 ans (les premières versions de la spécification datent de 1989), CORBA a connu de nombreuses évolutions, dont l'ajout du modèle de composants CCM (*CORBA Component Model*). L'objectif initial reste le même : il s'agit de fournir une couche intergiciel permettant de masquer l'hétérogénéité des machines, des réseaux, des systèmes, des langages et des formats de représentation de données pour permettre à des applications d'interagir à distance et d'échanger de l'information. Notons que si l'orientation objet de CORBA a très tôt été mise en avant, les spécifications sont ouvertes à des langages non orientés objet comme C ou Cobol afin de permettre à tout type d'application de s'intégrer dans un environnement CORBA.

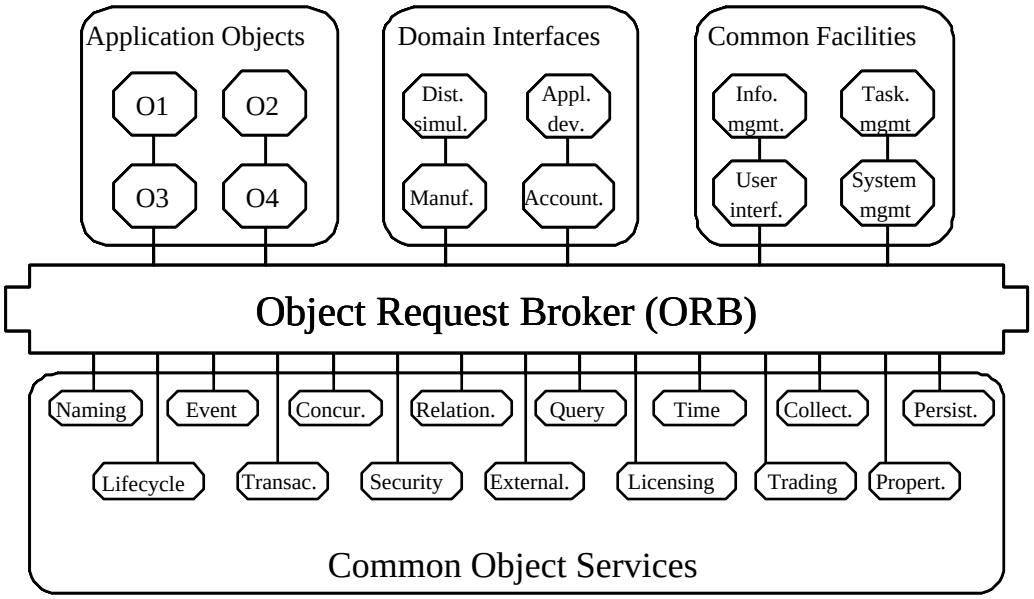

FIG. 2.2 – Architecture OMA pour CORBA.

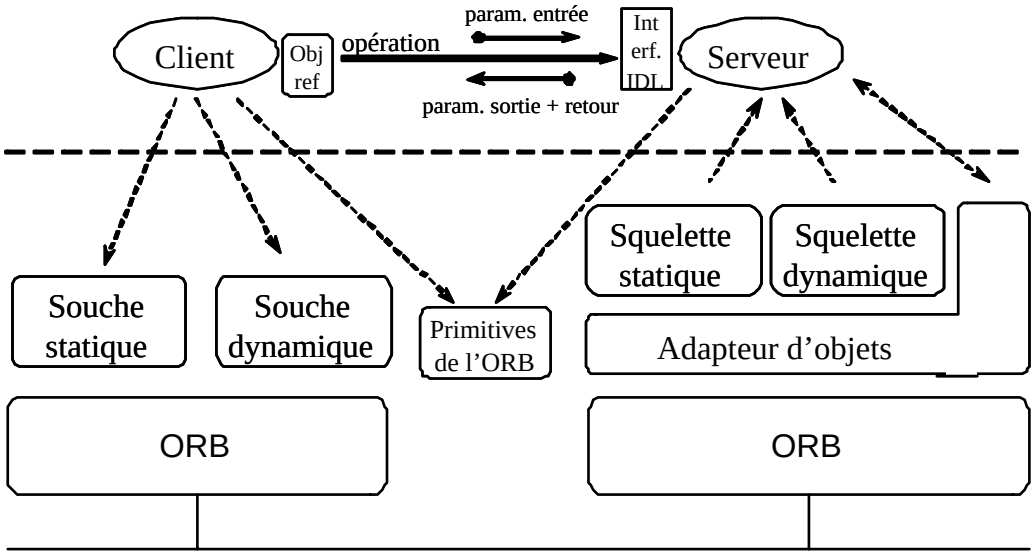

FIG. 2.3 – Architecture d'un ORB.

Au delà du standard CORBA produit par l'OMG, de nombreuses implémentations existent pour tout type de systèmes (Windows, Unix, MacOS) et de langages. Le site `cetus-links.org` recense plus de 40 ORB (implémentations des spécifications CORBA) commerciaux ou libres. L'adhérence de ces ORB aux spécifications de l'OMG garantit, en théorie, interopérabilité et transparence : deux applications fonctionnant avec des ORB différents peuvent interopérer, et une application conçue avec un ORB x peut fonctionner avec un ORB y.

De nombreuses activités de recherche et commerciale ont été développées autour de CORBA dans les années 1990 et ont abouti à des développements majeurs comme l'intégration de CORBA dans les logiciels CATIA (modélisation 3D et la CAO), SALOME (couplage de codes scientifiques) ou GNOME (gestion d'interfaces homme-machine pour Linux).

CORBA met en avant la métaphore du bus logiciel. Il s'agit d'atteindre le même niveau d'interopérabilité qu'avec les bus matériel et de permettre à des objets distants connectés au bus, de communiquer. Concrètement CORBA définit pour cela le langage de description d'interfaces IDL (*Interface Definition Language*). Indépendant des langages de programmation, il permet de décrire les services fournis par les objets connectés au bus. À partir d'une description IDL, des souches clientes et serveurs (appelées squelettes dans le vocabulaire CORBA) sont générées pour masquer aux applications la mécanique (complexe) des communications distantes.

L'architecture générale de CORBA distingue différentes catégories d'objets connectés au bus (voir figure 2.2). Les objets applicatifs sont ceux que les développeurs *lambda* écrivent. Les *Common Object Services* sont des objets implémentant des services communs aux applications distribuées. On y trouve des services d'annuaire (pages blanches et jaunes), un service de transaction, des services de diffusions d'événements, etc. Ces services sont très peu différents des objets applicatifs : comme eux, leurs interfaces sont définies en IDL, ce sont des objets serveurs CORBA et ils sont accessibles via le protocole IIOP (*Internet Inter-Orb Protocol*). La seule différence réside dans la façon dont on résout leur référence initiale. Seize services différents sont définis par l'OMG. Notons que quasiment aucun ORB ne fournit la totalité de ces services. Par contre, le fait que leurs interfaces soient standardisées en IDL permet de réutiliser assez facilement des services développés indépendamment. Les *Domain Interfaces* et les *Common Facilities* sont des frameworks (respectivement, verticaux et horizontaux) pour le développement d'applications CORBA.

L'architecture d'un ORB CORBA (voir figure 2.3) est organisée autour d'un noyau qui assure les communications distantes. Les opérations d'encodage et de décodage des données sont assurées par les souches côté client et des squelettes côté serveur. Les souches et les squelettes sont soit statiques, et dans ce cas, ils sont spécifiques à une interface IDL, soit génériques, et dans ce cas, ils peuvent être utilisés pour n'importe quel objet, quel que soit son interface IDL. La gestion des objets serveurs est prise en charge par un adaptateur d'objets (OA pour *Object Adapter*). Les adaptateurs fournissent aux objets un ensemble de services de bases comme la gestion du cycle de vie (création, activation, etc.), la gestion d'activité (*threads*, processus), la gestion des références (persistante ou non).

2.2.2 Réflexivité

La réflexivité [163, 164] désigne la capacité d'un système à raisonner et à agir sur lui même [103]. Les systèmes réflexifs distinguent deux niveaux : le niveau de base qui correspond à l'application et le niveau méta qui fournit une couche d'interprétation et de contrôle du niveau de base. La réification d'événements survenant au niveau de base permet au niveau méta de modifier le com-

portement de l'application. La réflexivité peut être structurelle ou comportementale [55]. Dans le premier cas, elle concerne la structure d'un programme (ses instructions, ses variables), tandis que dans le second elle concerne son comportement (gestion des messages, interprétation). Les langages réflexifs sont par ailleurs associés à des protocoles à méta-objet (MOP pour *Meta Object Protocol*). Ces protocoles définissent les interactions entre le niveau de base et le niveau méta. On distingue les MOP à la compilation (CT MOP pour *Compile Time MOP*), des MOP à l'exécution (RT MOP pour *Run Time MOP*). Les premiers s'appliquent lors de la compilation des programmes, tandis que les seconds opèrent à l'exécution. Les travaux de recherche sur la réflexivité ont été nombreux dans les années 1990, et ont conduit à de nombreuses implémentations. Celles-ci se présentent la plupart du temps comme des extensions de langages existants. Sans être exhaustif on peut citer :

- pour Smalltalk : CLOS [81], ABCL/R [179], CodA [109], Correlate [150],
- pour Java : OpenJava [171], Javassist [36], Reflex [170], MetaXa [63], Dalang [180], Iguana/J [147],
- pour C++ : OpenC++ [35], MPC++ [74].

Notons également que certains langages comme Smalltalk, Java, C#, Python ou Ruby possèdent de façon intrinsèque des mécanismes réflexifs (par exemple d'introspection). Par ailleurs, j'ai eu l'occasion au cours de ma thèse de doctorat de proposer un MOP pour la synchronisation d'objets concurrents [158] dans le système GUIDE [10].

La réflexivité est un domaine de recherche vaste qui peut faire l'objet d'un document complet. Dans la suite de ce document, nous ne l'abordons que par le prisme des travaux qui l'ont appliqué aux intergiciels et aux services pour la répartition, laissant volontairement de côté toutes ses autres facettes.

Dans le domaine des intergiciels, Ledoux dans [92, 93] a utilisé la programmation réflexive pour implémenter OpenCorba, un ORB CORBA réflexif. Quatre mécanismes fondamentaux de l'ORB ont été rendus réflexifs dans OpenCorba : le mécanisme d'invocation de méthodes distantes, le contrôle de type IDL lors d'une invocation, la cohérence du référentiel d'interfaces lors de la compilation d'une interface IDL et le verrouillage des *proxys* générés. Cette approche permet d'"ouvrir" l'ORB en fournissant un niveau de contrôle sur ses mécanismes internes. On est alors à même de modifier son fonctionnement en changeant les méta-objets, indépendamment du reste de l'ORB.

Dans le même esprit, le projet OpenORB [18, 42] définit un "espace méta" avec différents méta-objets pour contrôler le mécanisme de typage des interfaces, les connexions entre composants de base, l'interception de messages et la gestion des ressources.

Antérieurement, la réflexivité a également été mise en œuvre dans la construction du système d'exploitation Apertos [181]. Ce système est architecturé à l'aide d'objets et de méta-objets. Bien que généraliste, le domaine d'applications ciblé par ce système est l'informatique mobile et la possibilité de faire migrer des objets dans un environnement réparti. Un objet dans Apertos est associé à un espace méta (*metaspace*). Chaque espace méta définit le domaine de contrôle de l'objet, comprend un ou plusieurs méta-objets qui implémentent ce domaine et fournit aux objets une interface appelée *Reflector*. Les *Reflectors* sont organisés au sein d'une hiérarchie et fournissent différents services pour l'ordonnancement des tâches, la gestion des interruptions systèmes, la gestion mémoire, la gestion de la concurrence ou les invocations de méthodes. Apertos est architecturé autour d'un micronoyau, MetaCore, qui est un méta-objet terminal sans espace méta. Ce noyau fournit des primitives de base pour la création et le retrait des liens entre les objets et les espaces méta et des primitives de migration des objets entre espaces méta.

De façon complémentaire, des études se sont attachées à rendre réflexifs d'autres mécanismes ou services des intergiciels. On peut citer par exemple, la migration [108][121][181][109], la réplication [59][62][51], la persistance [59], la coordination [108][24][109][59][157, 158] ou la tolérance aux pannes [51].

2.3 Service réflexif pour l'observation d'exécutions réparties

Cette section fait le point sur les travaux que j'ai mené sur l'utilisation des techniques de programmation réflexive pour l'extension des intergiciels. Le but est de montrer que la réflexivité peut être utilisée de façon satisfaisante pour intégrer de façon transparente des services à une application s'exécutant sur un intergiciel CORBA. Pour cela un service d'observation d'une exécution répartie a été conçu et intégré à CORBA à l'aide de la réflexivité.

Cette expérience a servi de précurseur à mes travaux suivants sur l'utilisation de la programmation orientée aspect (chapitre 3). Il s'agit d'étudier comment des techniques de programmation évoluées telle que la réflexivité et les aspects peuvent améliorer l'ingénierie des applications distribuées et des intergiciels en facilitant le travail d'intégration des nombreuses fonctionnalités de ces logiciels.

Ce travail sur les intergiciels et la réflexivité a été mené entre 1995 et 2001, d'abord dans le cadre du système réparti orienté objet GUIDE [10] puis dans le cadre de CORBA. Ce travail a été mené en collaboration avec L. Duchien et deux stagiaires (DEA et ingénieur CNAM) P. Placide et E. Jeury. Il a donné lieu à la publication [49] qui synthétise et rassemble les résultats des publications préliminaires [140, 47, 48].

La contribution présentée dans cette section porte sur deux points :

1. Génie logiciel : nous avons montré que la réflexivité peut être utilisée pour intégrer de façon transparente de nouveaux services à une application CORBA. L'expérience a été mené avec l'ORB CORBA JacORB [26] et le langage réflexif à la compilation (CT MOP pour *Compile Time Meta Object Protocol* OpenJava [171]. Le service intégré réalise l'observation d'une exécution répartie.
2. Formalisation : nous avons défini une relation, dite ordre objet causal, notée $\rightarrow_o$, pour l'observation d'exécutions réparties. Cette relation étend la relation *Happened before* de Lamport et permet une observation plus sémantique du comportement de l'application. Cette relation est présentée en détail dans [49].

Cette étude porte sur l'observation d'exécutions réparties dans un environnement CORBA. Comme dans la plupart des études de ce type, nous supposons qu'il n'y a pas d'horloge globale ni d'horloges locales parfaitement synchronisées. L'observation d'une exécution nécessite donc un mécanisme additionnel pour ordonner les événements. La relation *Happened before* de Lamport fournit un bon point de départ pour cela. Elle a été utilisée par de nombreux auteurs [89][116][33][154][3] qui ont proposé des solutions pour la détermination d'un état global cohérent. Néanmoins cette relation s'appuie sur des dépendances qui sont issues des communications asynchrones. Premièrement on peut faire remarquer que les environnements comme CORBA promeuvent plutôt des schémas synchrones de type requête/réponse. Deuxièmement, d'autres sources de dépendances que les communications existent dans les applications réparties. Par exemple, la synchronisation des méthodes concurrentes introduit des dépendances qui ne sont pas capturées par la relation *Happened before*. Pour palier à cela, nous avons défini une relation d'ordre, appelée l'**ordre causal**

objet qui a pour but de capturer, non seulement les dépendances issues des communications, mais également les dépendances issues des synchronisations, des créations dynamiques de *threads* et des transactions.

Cette étude, présentée dans [49], étend et complète des travaux précédents [140, 47, 48]. [140] concernait l'observation d'exécutions réparties pour le système GUIDE [10]. [47] a été une première transposition au monde CORBA/Java. [48] a introduit un outil d'analyse *post-mortem* des exécutions. L'étude présente nos résultats sur l'observation en ligne, complète le modèle d'événements distribués, et fait le point sur les performances du système. L'environnement cible dans lequel a été réalisée l'étude comprend l'ORB CORBA *open source* JacORB [26] et le langage réflexif OpenJava [171]. OpenJava est un CT MOP que nous utilisons pour instrumenter de façon transparente une application en vue d'observer l'ordre causal objet.

La section 2.3.1 décrit l'architecture mise en place. La section 2.3.2 présente une évaluation du coût du service. La section 2.3.3 compare l'outil avec d'autres approches similaires.

2.3.1 Service d'observation

Dans cette section, nous présentons le service réflexif d'observation d'exécutions réparties pour CORBA/Java qui a été développé. L'ORB CORBA utilisé est JacORB [26]. L'observation est implémentée avec le langage réflexif OpenJava [171].

Architecture

Notre architecture comprend deux entités de base : un objet observateur et un méta-objet observateur (voir figure 2.4).

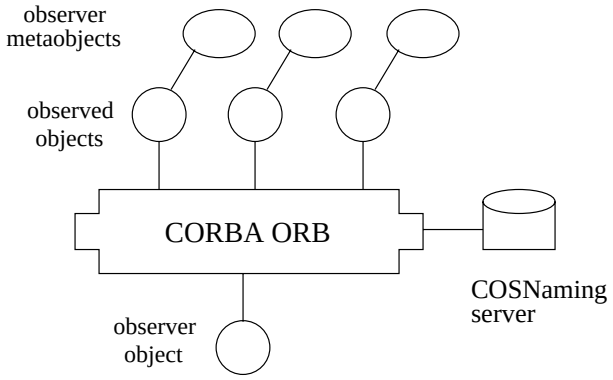

FIG. 2.4 – Architecture du service d'observation.

L'objet observateur est un objet CORBA standard. Il y a un objet de ce type par application observée. Cet objet possède une interface définissant des méthodes asynchrones pour l'enregistrement des événements observés. L'observateur est implémenté en Java et enregistre chaque événement reçu dans une table de vecteurs. Il y a un vecteur par objet observé et par variable partagée observée.

Le méta-objet observateur est associé à chaque objet applicatif devant être observé. Le processus de liaison entre les méta-objets observateurs et l'objet observateur est réduit à sa plus simple expression : l'observateur s'enregistre avec un nom connu de tous dans le service de nommage de CORBA, et chaque méta-objet observateur recherche ce nom. Les communications entre les

méta-objets observateurs et l'objet observateur sont asynchrones. Bien que la sémantique prévue dans les spécifications CORBA pour ces appels soit "au mieux" (i.e. il se peut que l'appel ne soit pas livré), ce mécanisme est plus efficace et moins intrusif que des appels synchrones.

Evénements observés

Les événements observés sont les suivants :

1. événements de communication : les objets communiquent via des appels de méthodes distantes, synchrones (requête/réponse, bloquant), ou asynchrone (requête seule, non bloquant). Six événements sont associés à ces opérations : appel de méthode, envoi, retour, arrivée, début et fin. L'événement appel de méthode est l'appel synchrone d'une méthode. L'événement retour de méthode est le retour associé à un tel appel. L'événement envoi de méthode est l'appel asynchrone d'une méthode. L'événement arrivée de méthode est généré quand une méthode est reçue du côté de l'objet appelé. Finalement, les événements début et fin de méthodes sont générés respectivement, quand une méthode débute et finit.
2. événements de gestion de *threads* : quatre événements liés aux *threads* sont pris en compte : création, début, fin, *join*.
3. événements de synchronisation : le modèle de gestion de la concurrence de Java permet de déclarer des méthodes synchronisées. Celles-ci sont associées à un moniteur et leur accès est protégé. Trois événements (parmi ceux déjà mentionnés) sont concernés par la synchronisation : arrivée, début et fin de méthode.
4. événements de lecture/écriture sur des variables partagées.

Meta-objet observateur

Le code nécessaire à l'observation des onze événements définis précédemment est fourni par des méta-classes OpenJava [171]. OpenJava est un protocole à méta-objet fonctionnant à la compilation (CT MOP) qui permet d'introspecter et de modifier un programme Java. Comme illustré dans la figure 2.5, la méta-classe de base de OpenJava est `OJClass`. Le mot-clé `instantiates` permet de créer un lien entre une classe de base et une méta-classe.

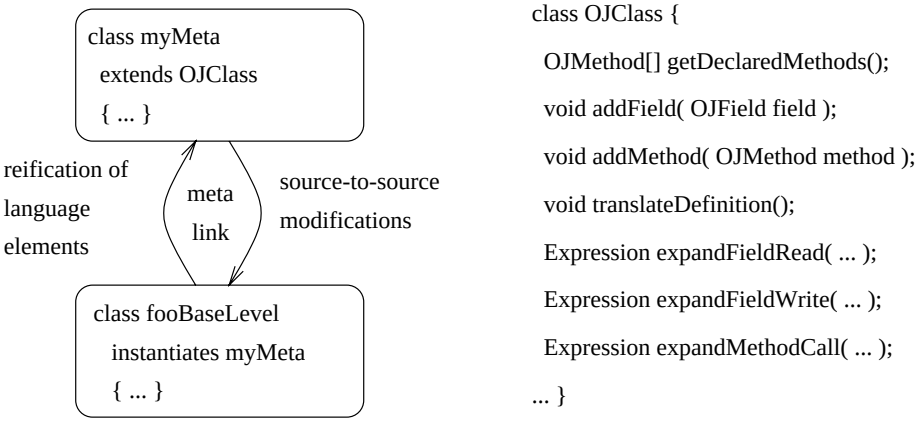

FIG. 2.5 – Lien méta avec OpenJava.

L'interface de classe `OJClass` comprend un nombre important de méthodes, offrant un éventail d'introspection et de modification très riche. En effet, chaque nœud de l'arbre syntaxique d'un programme Java est réifié par OpenJava et peut être inspecté et/ou modifié. Ajouté au fait que OpenJava est entièrement écrit en Java, cette richesse et cette finesse induisent un surcoût. Notons néanmoins qu'il s'agit d'un surcoût à la compilation uniquement. La figure 2.5 fournit un aperçu de l'interface de la classe `OJClass`.

Processus d'observation

La méta-classe `Observer` est la méta-classe de tout objet observé. Elle étend `OJClass` pour (1) enregistrer les événements de début et de fin de méthode (`translateDefinition`), (2) enregistrer les lectures et les écritures d'attributs (`expandFieldRead` et `expandFieldWrite`), (3) enregistrer les événements d'appel et de retour de méthodes (`expandMethodCall`). Les événement d'arrivée de méthode sont enregistrés avec un wrapper placé autour de toute méthode synchronisée. Les événements relatifs aux *threads* sont enregistrés avec un wrapper placé autour de la classe `java.lang.Thread`.

La méta-classe observateur fournit un mécanisme de configuration pour pouvoir sélectionner les attributs et les méthodes à observer. Il est ainsi possible de restreindre le volume d'information collecté et de concentrer l'observation sur les événements significatifs.

2.3.2 Analyse de performance

Dans cette section, nous présentons une évaluation du surcoût engendré par le processus d'observation. La plate-forme d'expérimentation comprend deux Sun Sparc Ultra 5 reliées par un réseau Ethernet 10 Mbits/s. Les deux machines fonctionnent sous Solaris 5.7 et utilisent la version 1.1.6 du JDK de Sun, JacORB 1.0b7 et OpenJava 1.0. Le tableau 2.1 résume nos résultats.

		Surcoût	Pourcentage de temps	
			réseau	CORBA ORB
Capture d'événement	Pire cas	1,7 ms	12 %	88 %
	Meilleur cas	1,5 ms	13 %	87 %
Piggy-backing		0,3 ms		

TAB. 2.1 – Mesure de performance.

Le premier ensemble de tests évalue le coût de la capture d'événements. Comme mentionné à la section 2.3.1, chaque événement est envoyé par un méta-objet observateur à l'objet observateur à l'aide d'un appel CORBA asynchrone. Dans le pire des cas, le coût du processus d'envoi sur le site émetteur est 1,7 ms dans le pire des cas et 1,5 ms dans le meilleur. Cette différence est due au volume d'informations différent à envoyer en fonction des événements. Ces résultats ont été obtenus en prenant les valeurs moyennes au cours de 10 séries de captures comprenant chacune 1000 événements. Le même test effectué quand l'observateur et l'observé sont colocalisés sur la même machine fournit respectivement 1,5 et 1,3 ms. On en déduit que la majeure partie du temps est passée dans les souches et les squelettes CORBA, et que le réseau n'intervient que pour une part faible (respectivement 12 et 13 %).

Le second test évalue le coût de l'envoi de paramètres supplémentaires (*piggy-backing*) pour

chaque appel de méthode applicative. Comme mentionné à la section 2.3.1, ces informations permettent d'enregistrer les dépendances causales entre les événements d'appel et d'arrivée de méthode. Nos tests montrent qu'un appel CORBA synchrone entre deux machines distantes prend 2,6 ms. Le surcoût introduit par le paramètre supplémentaire est 0,3 ms. Ce résultat correspond à la moyenne de 10 séries de 1000 appels synchrones.

2.3.3 Comparaison avec des travaux proches

Cette section présente quelques travaux proches et les compare à notre approche.

Plusieurs outils, comme [68], existent pour l'observation d'exécutions parallèles programmées avec la librairie PVM. La plupart des outils utilisent, comme dans notre approche, la relation *Happened before* de Lamport. L'approche PVM, basée sur des primitives d'envoi et de réception de messages, offre un degré d'abstraction quelque peu inférieure à une approche comme la notre basée sur CORBA. De plus, les informations fournies par l'ordre causal objet sont de plus haut niveau, plus proches de la sémantique de l'application que ce qu'offre la relation *Happened before*.

[176] décrit un analyseur de protocole IIOP basé sur *tcpdump*. L'analyseur affiche les messages IIOP envoyés et reçus par les objets CORBA. Inprise AppCenter [73] propose un outil similaire au niveau des invocations CORBA (une invocation CORBA peut être découpée en plusieurs messages IIOP). Ces deux outils se limitent aux événements de communication et ne prennent pas en compte, comme nous le faisons, les événements de niveau JVM (comme la création de *threads*), ou les événements de niveau applicatif (comme la synchronisation de méthodes).

Les projets MAScOTTE [107] et GoodeWatch [64] s'intéressent à l'observation d'ORB CORBA. MAScOTTE définit une base de gestion d'information pour superviser l'activité d'un ORB CORBA. GoodeWatch fournit des mécanismes pour récupérer des événements internes de l'ORB. Notre outil va plus loin que la capture et propose un calcul et un affichage des dépendances causales.

2.4 Conclusion

Ce chapitre a présenté la conception et la réalisation d'un service réflexif d'observation d'exécutions réparties pour CORBA.

Le premier résultat de cette étude réside dans la définition d'une relation causale, appelée ordre causal objet, pour l'observation d'exécutions CORBA réparties. Cette relation étend la relation *Happened before* de Lamport en (1) considérant à la fois les communications synchrones et asynchrones (la relation *Happened before* ne prend en compte que les relations asynchrones), et (2) incorporant des dépendances générées par les communications, les synchronisations de méthode et les partages de variables (*Happened before* ne prend en compte que les communications). L'ordre causal objet ainsi défini fournit des informations de niveau sémantique qui permettent de mieux comprendre, analyser et visualiser une exécution répartie.

Un deuxième résultat porte sur l'utilisation des capacités réflexives du langage OpenJava [171] pour mener à bien l'intégration du service dans l'application. Nous avons montré que le service d'observation peut être implémenté de façon transparente à l'aide d'un méta-programme. Ce méta-programme réifie les événements liés aux méthodes, aux *threads* et aux opérations de lecture/écriture sur les variables partagées. Ces événements sont alors envoyés en utilisant l'ORB à un observateur qui calcule et affiche interactivement les dépendances causales.

Plusieurs extensions peuvent être envisagées à cette étude. Premièrement, des algorithmes de détection de prédicats globaux (avec par exemple, des techniques issues de [34] ou [60]) pourraient être ajoutés. Cela permettrait de superviser plus automatiquement les exécutions et de lever des alarmes lorsque des conditions globales ne sont plus respectées. Cela permettrait ainsi d'améliorer la mise au point et la correction des applications réparties. Deuxième, un système de filtrage pourrait être mis en place pour analyser pour précisément les traces. De la même façon, la notion d'événement abstrait [14], qui est un ensemble non vide d'événements primitifs, pourrait être introduite afin de synthétiser le graphe généré, d'en donner une vision plus macroscopique et de réduire le volume de traces.

Chapitre 3

Programmation orientée aspect

Sommaire

3.1 Problématique

Le sentiment que les objets ont échoué [57] aussi bien en termes de processus d'analyse/conception que de langage, a conduit à l'émergence de divers paradigmes, que l'on peut qualifier de post-objet, dont les aspects que nous abordons dans ce chapitre.

La programmation orientée aspect (AOP pour *Aspect-Oriented Programming*) [84] est une technique d'ingénierie logicielle qui vise à améliorer la modularisation des applications. L'AOP part du constat que, quel que soit le style de programmation, objet ou procédural, le code de certaines fonctionnalités reste dispersé et n'est pas pris en compte de façon satisfaisante par les classes ou les procédures. Les aspects complètent donc les langages existants en permettant de modulariser proprement ces fonctionnalités.

L'objectif de l'AOP est de rendre les logiciels plus modulaires. On espère ainsi faciliter la conception, le développement, la maintenance et l'évolutivité de ces logiciels. Les intergiciels ayant une taille et une complexité conséquentes, l'AOP peut potentiellement beaucoup leur apporter. De nombreux projets se sont développés autour de l'utilisation de l'AOP, soit pour le développement d'intergiciels, soit pour le développement de services pour les intergiciels, soit pour le développement d'applications pour les intergiciels.

Ce chapitre commence par une présentation des principes de l'AOP (section 3.2). Les deux sections suivantes sont consacrées à notre contribution au domaine. La section 3.3 présente la plateforme JAC et la section 3.4 présente une couche d'abstraction, basée sur l'AOP, pour simplifier le développement d'applications EJB et CCM. Nous terminons ce chapitre par une conclusion et des perspectives sur ce domaine.

3.2 Principes

La programmation orientée aspect est un paradigme apparu au milieu des années 1990 issu des travaux menés au Xerox PARC dans l'équipe de Gregor Kiczales. Parmi les membres de cette équipe, Christina Lopes fait remonter l'apparition du terme AOP à 1995 [99, 100] et en attribue la paternité à Chris Maeda. La publication des papiers fondateurs ont lieu en 1996 dans un workshop de l'ACM [79] et en 1997 à la conférence ECOOP [84].

L'AOP est née du constat que si la programmation orientée objet fournit de bons résultats pour la modularisation des fonctionnalités verticales (ou métier), elle échoue par contre à bien prendre en compte les fonctionnalités horizontales (ou transversales). L'idée majeure de l'AOP est donc de modulariser dans une entité logicielle appelée aspect, l'implémentation des fonctionnalités qui dans une programmation "classique" (objet, procédurale ou fonctionnelle) sont dispersées dans l'ensemble de l'application. Cette dispersion est un frein à la maintenabilité et à l'évolutivité des applications. En rassemblant ce code dispersé dans une seule entité logicielle, l'AOP lève ces freins et conduit à des applications qui sont plus modulaires. Nous reviendrons plus en détail sur les différents concepts de l'AOP dans la section 3.2.3.

Rien dans l'AOP n'est spécifique à la programmation orientée objet en général, ou au langage Java en particulier. L'aspect est un concept général, qui, comme l'objet, peut être appliqué à différents langages. Bien que la plupart des langages ou frameworks pour la programmation orientée aspect soit en Java, on trouve aussi des outils d'AOP plus ou moins avancés en C, C++, C#, Smalltalk voire d'autres langages comme Cobol, PHP, Python. En fait, n'importe quel langage objet, procédural ou fonctionnel est susceptible d'être étendu avec le concept d'aspect. En tant que paradigme de programmation, l'AOP ne vise pas à remplacer la programmation objet mais a plutôt comme objectif de la compléter afin d'obtenir des programmes mieux structurés, plus clairs et modularisant les fonctionnalités transversales.

Séparation des préoccupations

L'AOP peut être vue à la base comme un technique de séparation des préoccupations. Ossher et Tarr définissent une préoccupation [125] (*concern*) comme :

Définition 3.1 *A concern is the part of a software system relevant to a particular concept, goal, or purpose.*

Ce principe, identifié dès 1972 par D. Parnas [126] et aussi connu sous l'expression "diviser pour régner" vise à découper un logiciel en unités aussi petites que possible pour mieux maîtriser la complexité de l'ensemble. Bien évidemment, on souhaite que ces entités soient aussi autonomes que possible de façon à faciliter leur développement et à maximiser les chances de pouvoir les réutiliser dans d'autres applications.

En pratique, l'application de cette notion est plus complexe et de nombreux paramètres entrent en jeu et brouillent la vision de ce que devrait être cette séparation :

1. tout d'abord la notion de préoccupation est subjective et peut varier en fonction du domaine, de l'application, voire du concepteur,
2. la granularité des préoccupations n'est pas uniforme : cela peut recouvrir un domaine aussi vaste que la sécurité comme cela peut concerner l'implémentation précise d'une fonction de tri,
3. la notion de préoccupation n'est pas figée dans le temps. Il est courant que les applications d'entreprise ou scientifiques ne soient pas conçues linéairement et que de nouveaux besoins émergent en cours de développement et engendrent l'apparition ou la disparition de préoccupations.

Si l'AOP ne répond pas directement à la question difficile de savoir comment on découvre une préoccupation, elle fournit un certain nombre de réponses sur la façon dont on isole une préoccupation et on la compose avec une application.

3.2.1 Origines et évolution

Bien que l'apparition de la notion d'aspect en tant que telle remonte à une dizaine d'années, elle prend ses racines et s'inspire directement de travaux antérieurs comme ceux sur la réflexivité [163, 164] et sur les implémentations ouvertes [80]. Par exemple, des notions importantes de l'AOP comme les codes advice *before*, *after* et *around* qui permettent de définir le code d'un aspect, existent déjà dans les extensions objet du langage Lisp comme Flavors, LOOPS ou CLOS [81]. Par contre, il semblerait que l'autre notion importante de l'AOP, celle de coupe (en anglais *pointcut*), qui permet de définir où un aspect doit être intégré dans une application, peut être attribuée de façon plus indéniable à l'AOP.

Les travaux de l'équipe de G. Kiczales autour de l'AOP ont d'abord pris deux directions (voir [99, 100] pour un historique plus complet de l'évolution de ces travaux) :

1. les travaux autour de RG [113], AML et ETCML avaient pour but de programmer des aspects d'optimisation de programmes existants. Il s'agit par exemple pour RG d'optimiser l'utilisation de la mémoire et de d'opérer des fusions de fonctions sur des programmes travaillant sur des matrices. Ces travaux font apparaître une influence et une filiation claire de la réflexivité sur l'AOP : les aspects sont envisagés comme des méta-programmes qui donnent une sémantique ou une interprétation particulière aux constructions d'un langage de base.
2. les travaux autour de D [101] et de son implémentation en Lisp DJ, se sont intéressés à la programmation distribuée. L'idée est qu'un programme réparti doit adresser de front plusieurs préoccupations (communications distantes, synchronisation, gestion de caches, réplication, cohérence, équilibrage, sécurité, etc.) qui se retrouve entremêlées. Face à ce problème, D propose deux langages, COOL pour la gestion de la synchronisation et RIDL pour la définition d'interfaces de communication distante. Implémentés comme des extensions du langage Java,

COOL et RIDL permettent donc de modulariser ces deux préoccupations. Il est intéressant de noter ici que la démarche est proche de celle adoptée par la communauté des DSL (*Domain Specific Language*) [41][178] ou des ASL (*Aspect Specific Language*) [44] : on cherche pour chaque aspect à trouver les constructions linguistiques qui permettent d'exprimer de manière adéquate les concepts propres à l'aspect.

Ces travaux ont permis de mettre en place les premiers concepts de l'AOP. Ils sont néanmoins très spécifiques à des aspects particuliers comme l'optimisation de programmes ou à la répartition. Afin d'élargir l'audience de l'AOP, l'équipe de Gregor Kiczales a décidé de s'orienter vers un langage AOP généraliste, i.e. permettant de programmer n'importe quel type d'aspect. Cet objectif a donné naissance en 1998 à AspectJ.

AspectJ

AspectJ [83] définit une syntaxe pour définir des aspects à appliquer sur des programmes Java. Le langage est implémenté par un compilateur qui prend en entrée un ensemble d'aspects et de classes, et qui fournit en sortie les classes augmentées des comportements définis par les aspects. Dans ses premières versions, le compilateur AspectJ effectuait une transformation source vers source pour intégrer les aspects et compilait avec le code Java généré. À partir de la version 1.1 (sortie en juin 2003), le compilateur génère directement du bytecode Java, ce qui accélère les temps de compilation et supprime l'étape de compilation Java.

AspectJ a subi de nombreuses évolutions se traduisant toutes soit par de meilleures performances, soit par de nouvelles fonctionnalités inclues dans le langage ou encore par des correctifs de bugs. La version 1.0, synonyme de maturité, est sortie en novembre 2001. En décembre 2002, le projet a quitté le giron du PARC pour rejoindre la communauté Eclipse. Sept ans après sa sortie, AspectJ reste le langage phare de l'AOP auquel toute la communauté se réfère et compare ses travaux (pour une présentation complète de AspectJ, voir le chapitre 3 de [148] ou [149]).

Communauté

La communauté AOP s'est fédérée autour de différentes manifestations comme des workshops à ECOOP (dès 1997) et OOPSLA (dès 1998). Rapidement, l'AOP a connu un engouement important : les chercheurs et les industriels étaient attirés par les promesses d'un développement plus modulaire et d'une intégration plus aisée. Cet engouement a culminé par l'inclusion en 2001 par le MIT de l'AOP comme l'une des "10 technologies[1] qui vont changer le monde" [117]. L'année 2001 a également vu la reconnaissance de l'AOP avec la publication d'un numéro spécial de la revue CACM faisant le point sur les recherches du domaine. La tenue en 2002 de la première conférence international exclusivement consacré au développement orienté aspect (AOSD 2002) a également conforté l'AOP comme domaine à part entière de l'informatique.

Comme nous venons de le constater le concept d'aspect a initialement été envisagé sous l'angle de la programmation. Néanmoins, il est suffisamment général pour pouvoir être appliqué, comme cela a été le cas avec les objets, à d'autres phases de l'ingénierie logicielle comme l'analyse et la conception. Les recherches sur les aspects sont ainsi passées du cadre de l'AOP à celui de l'AOSD (Aspect-Oriented Software Development).

[1] toutes disciplines confondues, i.e. pas spécifiquement en informatique.

AOP et intergiciel

La communauté des intergiciels (*middleware*) a été dès le départ très réceptive aux idées de l'AOP pour deux raisons principales. À titre indicatif, le livrable [8] du réseau d'excellence IST sur le développement orienté aspect (AOSD) auquel je participe, recense onze intergiciels qui utilisent ou incluent un support AOP. Deux raisons principales expliquent selon moi cet intérêt de la communauté des intergiciels pour les aspects : ils ouvrent des perspectives en terme de gestion de la dynamicité et de la complexité.

Dynamicité L'approche suivie par AspectJ est qualifiée d'AOP statique au sens où, une fois compilés, les aspects sont fondus dans la masse et il n'est plus possible de les distinguer aisément du reste du code. Le code des aspects est ainsi lié statiquement à celui des classes.

Dans le cas des intergiciels, certains serveurs ne peuvent pas être arrêtés fréquemment. Lorsque l'on souhaite néanmoins pouvoir étendre ou modifier les fonctionnalités de ces serveurs à chaud, des solutions à base de frameworks AOP dynamiques constituent de nouveau une solution envisageable.

Ainsi, dès 1998, le besoin d'une **liaison dynamique** entre aspects et classes a émergé [78]. L'idée est de pouvoir ajouter et retirer dynamiquement des aspects sur un programme en cours d'exécution. De nombreux frameworks implémentant ce concept sont alors apparus. Les quatre frameworks principaux, JAC [134, 133], AspectWerkz, JBoss AOP, Spring AOP, ont d'ailleurs des liens forts avec le domaine des intergiciels : AspectWerkz et JBoss AOP ont été développés par des équipes en charge de serveurs d'applications J2EE (respectivement, BEA WebLogic et JBoss) ; JAC et Spring AOP visent, comme J2EE, le domaine applicatif des serveurs d'information 3-tiers ouverts sur l'Internet.

Gestion de la complexité Les intergiciels et leurs applications ont des besoins très variés en termes de fonctionnalités : communications distantes, sûreté de fonctionnement, disponibilité, tolérances aux défaillances, mobilité, persistance de données, équilibrage de charge, réplication etc. Assembler les briques logicielles implémentant ces fonctionnalités n'est pas aisée. De plus, l'approche traditionnelle de l'intergiciel boîte noire est très peu flexible.

Dans ce cadre, l'AOP a rapidement été vue comme une solution permettant de simplifier l'architecture des intergiciels. Plusieurs études se sont ainsi attachées à appliquer une phase de ré-ingénierie orientée aspect à des intergiciels existants. On peut citer :

- [85] a modifié l'architecture du serveur Web JaWS (Java Web Server) développé par Sun et en a extrait sept aspects : gestion de la concurrence, des entrées/sorties, supervision de l'environnement et de la charge, gestion d'exceptions, gestion des connexions, configuration du protocole, communication avec les bases de données, supervision de l'activité du serveur.
- [183] a extrait un certain nombre d'aspects de trois ORB CORBA : ORBacus de la société IONA, JacORB et OpenORB, deux ORB *open source*. Les aspects concernent la gestion du modèle de programmation dynamique (mécanismes DII et DSI de CORBA), la gestion des intercepteurs (*Portable Interceptor*), la gestion des erreurs, des traces, des synchronisations et des pré/post conditions. Les auteurs montrent que l'utilisation de l'AOP a permis de simplifier l'architecture des ORB en réduisant la taille du code de 9%, qu'il est plus facile d'optimiser chacune des fonctionnalités aspectisées (elles ne sont plus entremêlées avec le reste du code) et qu'elle offre des ORB plus flexibles dans lesquelles les fonctionnalités aspectisées peuvent être retirées ou ajoutées en fonction des besoins.

– [40] est le projet certainement, à ce jour, le projet le plus ambitieux entrepris dans le domaine des aspects et des intergiciels. Les auteurs ont aspectisé le serveur d'applications WebSphere d'IBM (environ 15 000 fichiers source et plusieurs millions de lignes de code d'après [40]) et en ont extrait les aspects de gestion des traces, des erreurs, de supervision et de gestion des composants EJB. Ce dernier aspect est particulièrement intéressant en terme de flexibilité, puisqu'il permet, à partir de la même base de code, d'obtenir un serveur applications qui supporte ou non les EJB. Au delà de la flexibilité, l'objectif poursuivit par les auteurs est d'introduire une plus grande cohérence dans la ligne des produits IBM : à partir du moment où les briques logicielles communes à plusieurs logiciels sont clairement isolées dans des aspects, il est plus facile de les réutiliser et de garantir que tous les logiciels de la ligne utilisent des versions cohérentes de ces briques. Suite à cette étude, D. Sabbah, vice-président de IBM Software Group, a indiqué, lors d'un discours à la conférence AOSD 2004, que "pour IBM, intégrer l'AOP dans ses modèles de développement de logiciel est une question de survie" [152].

Notons que ces trois études ont été conduites avec AspectJ.

Au delà de la simplification de la conception et de la maintenance, l'introduction d'aspect permet également de faciliter la réutilisation des fonctionnalités logicielles : en les ayant externalisé dans des aspects, il est plus facile de les réutiliser. L'AOP est ainsi de la problématique de personnalisation des intergiciels et doit permettre d'aboutir à des lignes de produits adaptables et sélectionnables en fonction des besoins.

Au delà du rôle structurant pour l'architecture des intergiciels, l'AOP joue également un rôle dans le développement des services pour ces intergiciels. L'AOP a ainsi été utilisé pour implémenter des services de persistance [146, 145][165], de communication distante avec RMI [165], de transaction [52], équilibrage de charge [143], de diffusion d'événements [142] ou de politique de pré-chargement dans les caches web [169].

3.2.2 Limites de la programmation orientée objet

Avant d'aborder la définition des principaux concepts de l'AOP, cette section fait le point sur les limites identifiées dans la programmation orientée objet. Ces limites constituent le point de départ de la réflexion sur les aspects.

3.2.2.1 Fonctionnalités transversales

L'exemple fondateur ayant permis de justifier l'intérêt de la programmation orientée aspect a été exhibé par G. Kiczales et son équipe. Il s'agit d'une étude sur le serveur de servlets Tomcat. Si, dans ce logiciel écrit en Java, la plupart des fonctionnalités sont clairement modularisées à l'aide de classes, certaines fonctionnalités comme la gestion des sessions utilisateurs se retrouvent dispersées dans de nombreuses classes. De telles fonctionnalités, non clairement implémentées dans une classe, sont dites **transversales** (en anglais *crosscutting*).

Il apparaît que la transversalité nuit à la maintenabilité et à l'évolutivité des applications. En effet, toute correction ou modification du code entraîne l'examen d'un nombre élevé de classes, ce qui est sujet à erreur. Or, la programmation objet n'offre pas de solution pour la prise en compte de cette transversalité. La notion d'aspect vient compléter la programmation objet en offrant une solution pour la modularisation de ces fonctionnalités transversales. Cette caractéristique est fondamentale dans la détermination d'un aspect. Face au problème de savoir si une fonctionnalité

doit être implémentée avec une classe ou un aspect, le facteur déterminant est la réponse à la question : la fonctionnalité est-elle transversale ? Cela est résumé de la façon suivante par Kiczales : *"AOP is about capturing a cross-cutting concern"*.

3.2.2.2 Exemples de fonctionnalités transversales

De nombreux exemples de fonctionnalités transversales existent. Nous les regroupons selon qu'ils concernent les services techniques (ou système) ou la partie métier des applications.

D'une manière générale, de nombreux services pour les systèmes répartis (sécurité, persistance des données, tolérance aux défaillances, etc.) sont transversaux. Ils impactent rarement une seule partie du programme et se retrouvent la plupart du temps dispersés dans toute l'application. Ce sont donc des candidats naturels pour l'aspectisation. Notons néanmoins que l'on ne parle pas ici de l'implémentation de ces services (comme c'était le cas précédemment avec l'implémentation de la gestion des sessions dans Tomcat) mais de leur utilisation (le programme invoque par exemple une API de persistance). Cela ne change rien à la dispersion du code : les appels à l'API apparaissent "partout" et polluent l'application. On se retrouve face à un problème d'intégration auquel l'aspect répond en rassemblant en un seul endroit la définition de l'impact d'un service technique sur une application. Nous y reviendrons dans la section 3.2.4.

Au niveau métier, plusieurs éléments peuvent bénéficier d'une implémentation sous forme d'aspect. C'est le cas par exemple des contraintes d'intégrité référentielle, des patrons de conception, des règles de gestion métier, ou des contrats fonctionnels.

- Les contraintes d'intégrité introduisent une dépendance entre des données localisées dans des classes différentes. Ces contraintes ne sont propres à aucune classe en particulier. La contrainte vient se superposer aux classes mises en relation. La définition d'un aspect pour l'implémentation de cette contrainte est donc une solution plus intéressante que de reporter artificiellement la contrainte dans une des deux classes.
- Les patrons de conception (*design pattern*) sont aussi de bons candidats à une implémentation sous forme d'aspect. En effet ils introduisent la plupart du temps des structures qui se superposent aux classes métier. Plusieurs études [66][148, 149] ont montré qu'une implémentation des *design patterns* à l'aide d'aspect conduit à des solutions plus intéressantes qu'avec une approche pure objet.
- Les règles de gestion métier et les contrats fonctionnels (au sens de la programmation par contrats de Meyer [115]) superposent à une application des contraintes d'exécution spécifiques. Encore une fois il peut être intéressant de découpler leur implémentation pour découpler les règles de la partie métier et faciliter leurs évolutions respectives.

3.2.2.3 Plusieurs dimensions de modularisation

Partant du constat qu'il existe des fonctionnalités transversales dans les applications, la question de l'inéluctabilité de cette transversalité peut être posée. En d'autres termes : est-ce les fonctionnalités sont transversales parce que l'application a été mal conçue ? Pourrait-on reconcevoir l'application en faisant disparaître cette transversalité ? La réponse n'est pas simple et reste ouverte. Néanmoins, il est admis que les applications complexes ne sont jamais linéaires et que, quelle que soit la façon de concevoir l'application, il restera toujours des fonctionnalités non proprement modularisés.

Quel que soit le processus d'analyse/conception (RUP, etc.) retenu, la détermination des classes d'une application obéit la plupart du temps à une logique fondée sur un découpage des données métier. Or, si cette logique est importante, ce n'est pas la seule à prendre en compte dès lors que les applications sont un tant soit peu compliquées : on va devoir par exemple mettre en œuvre des politiques de sécurité, ou s'intéresser à l'intégration de l'application avec l'existant, ou encore aux performances. Ces impératifs conduisent à des découpages qui entrent potentiellement en conflit avec le découpage issu des données métier. On s'aperçoit qu'il y alors plusieurs **dimensions de modularisation** dans une application.

Il apparaît donc que plusieurs logiques doivent être prises en compte introduisant plusieurs vues sur une application. Notons tout d'abord que cette notion n'est pas entièrement nouvelle et qu'elle existe dans d'autres domaines : dans la norme RM-ODP avec la notion de points de vue pour la conception d'applications pour les télécoms, dans les SGBD (notion de vue sur des tables), dans des travaux [32] sur les vues dans UML, ou encore en analyse des besoins avec l'approche MDSoC [124] (*Multi-Dimensional Separation of Concerns*) d'IBM. L'AOP est une technique qui va permettre la prise en compte de ces vues (ou dimensions) au niveau des programmes dans un premier temps, puis avec l'émergence des travaux autour de la conception orientée aspect (AOSD pour *Aspect-Oriented Software Development*), dans les autres phases de l'ingénierie logicielle.

3.2.3 Concepts

Cette section présente la définition des concepts principaux de l'AOP. Malgré quelques variations sémantiques d'un langage à l'autre, ces définitions sont stables et acceptées par l'ensemble de la communauté.

Définition 3.2 *Un aspect est une entité logicielle qui capture une fonctionnalité transversale à une application.*

Structurellement la notion d'aspect est proche de celle de classe. Les frameworks AOP comme JAC demandent d'ailleurs aux développeurs d'écrire directement une classe pour définir un aspect. En AspectJ, un aspect est compilé vers une classe.

Dynamiquement, le modèle d'instanciation des aspects est par contre différent de celui des classes. Alors que le programmeur objet crée autant d'instances que nécessaire, c'est l'environnement d'exécution (ou le code généré par le compilateur) qui prend en charge la création des instances d'aspect. Notons néanmoins que cette création peut être personnalisée avec des *factories* fournies par le programmeur (voir JBoss AOP ou AspectWerkz).

Cette différence révèle le caractère **asymétrique** de l'AOP telle qu'elle est envisagée actuellement par la majorité des langages et frameworks actuels (AspectJ, JAC, AspectWerkz, etc.). La fonctionnalité transversale implémentée par un aspect n'est pas autonome (i.e. pas directement instanciable) et ne se conçoit pas indépendamment d'un socle (i.e. des fonctionnalités implémentées dans des classes) avec lequel elle se compose. Par rapport à l'évolution des recherches sur l'AOP, cette asymétrie a permis de bien expliquer la nature profonde des aspects et l'existence de fonctionnalités transversales. À plus long terme, il est certainement moins efficace en terme d'ingénierie logicielle de disposer de deux concepts pour architecturer les applications. Nous pensons effectivement qu'une approche plus **symétrique** réconciliant fonctionnalités verticales et transversales serait plus efficace. Nous reviendrons sur ce point dans le chapitre 4 dans lequel nous

présentons nos travaux en cours sur le modèle FAC qui unifie les notions de composant et d'aspect. D'autres approches comme que la programmation orientée sujet (SOP) [67, 123] ou le projet VVM [120] dans lequel le langage des VMlets sert à la fois au développement des applications et du support d'exécution, promeuvent également cette approche.

Les deux éléments principaux définis dans un aspect sont les coupes et les code advice. Les coupes définissent **où** l'aspect doit être intégré dans une application et les codes advice définissent ce que fait l'aspect (i.e. le **quoi**). La notion de coupe s'appuie sur celle de point de jonction.

Définition 3.3 *Un point de jonction (join point) est un point dans l'exécution d'un programme autour duquel un ou plusieurs aspects peuvent être ajoutés.*

Il existent différents types de point de jonction. Les plus courants sont ceux qui concernent les méthodes (exécution et appel d'une méthode), les attributs (lecture, écriture) et les classes (instanciation). Les langages et les frameworks AOP se distinguent en général par le nombre de types de point de jonction qu'ils supportent. Notons que la plupart des besoins sont en général adressés par les deux types de points de jonction les plus simples et les plus courant : exécution et appel de méthode.

Définition 3.4 *Une coupe (pointcut) désigne un ensemble de points de jonction.*

Une coupe est définie avec un langage de *patterns* qui permet d'indiquer **où** l'aspect doit être intégré dans l'application. Une coupe permet donc de "parler de l'application" en désignant les emplacements stratégiques devant recevoir des aspects. Filman parle à ce propos de quantification [56][2]. Cette quantification se base sur la structure (noms des classes, méthodes, attributs, etc.) de l'application sous-jacente. Une direction de recherche que nous menons actuellement consiste à fonder cette définition sur des notions plus comportementales.

Les notions de coupes et de points de jonction sont liées par leur définition. Pourtant, leur nature est assez différente. Une coupe est un élément de code défini dans un aspect, alors qu'un point de jonction est un point dans l'exécution d'un programme. Si une coupe désigne un ensemble de points de jonction, un point de jonction donné peut appartenir à plusieurs coupes d'un même aspect ou d'aspects différents.

Définition 3.5 *Un code advice est un bloc de code définissant le comportement d'un aspect.*

Le terme advice n'admet pas de traduction reconnue par la communauté AOP francophone. Littéralement, il signifie conseil, avis, mais aucun auteur francophone n'emploie ces termes dans ce sens. Nous nous en tenons donc à cet usage.

Concrètement, un code advice définit des instructions et est associé à une coupe. Il implémente la sémantique de l'aspect. Contrairement à une méthode, un code advice n'est pas invoqué directement, mais exécuté lorsque les points de jonction de la coupe à laquelle il est associé surviennent. De façon standard, trois types de code advice existent selon qu'ils sont exécutés avant, après ou autour des points de jonction. AspectJ introduit d'autres types comme `after returning` et `after throwing` qui exécutent le code advice, respectivement, après la fin normale d'une méthode et après la levée d'une exception.

Les codes advice de type autour (`around`) sont associés à une instruction appelée en général `proceed` (`invokeNext` avec JBoss AOP) qui permet d'exécuter le code correspond au point de

[2] *"Aspect-oriented programming is quantification and obliouvness".*

jonction. L'appel à `proceed` est facultatif. Par exemple, un aspect de contrôle d'accès peut décider de ne pas exécuter la méthode interceptée si l'utilisateur n'est pas authentifié. À la fin d'un code advice, l'exécution du programme reprend juste après le point de jonction. Lorsque plusieurs aspects sont tissés sur le même point de jonction, `proceed` provoque, soit l'exécution de l'aspect suivant, soit l'exécution du point de jonction s'il n'y a pas d'autre aspect à exécuter.

Nous terminons cette brève présentation par les concepts de tissage, d'aspect abstrait et le mécanisme d'introduction.

Tissage

Définition 3.6 *Le tissage (weaving) est le mécanisme qui à partir d'un ensemble d'aspects et d'une application de base fournit une application dont le comportement est étendu par les aspects.*

On distingue trois catégories de tisseur d'aspects selon le moment où s'effectue le tissage : à la compilation, au chargement ou à l'exécution.

Dans le premier cas, le tisseur s'apparente à un compilateur : il modifie le code d'une application avec le code des aspects. Les aspects sont ici des entités qui n'existent qu'à la compilation. Lors de l'exécution, il n'y a pas de distinction entre le code original et celui des aspects. À moins de mettre en place des processus de recompilation, il est peu aisé de retirer ou de modifier un aspect. Le tissage à la compilation peut s'effectuer sur du code source ou sur du pseudo-code. AspectJ [83] est la référence des tisseurs à la compilation[3].

Avec le tissage à l'exécution, le lien entre l'application et les aspects est réalisé au moment de l'exécution. Ce lien est dynamique et il est aisé de le retirer ou de le modifier ce qui provoque une évolution du tissage. Les aspects sont ici des entités (objets) propres qui cohabitent avec celles de l'application lors de l'exécution. Deux techniques principales utilisées par les tisseurs à l'exécution : l'instrumentation préalable (à la compilation ou au chargement) du code des applications en vue de leur tissage (JAC, AspectWerkz, JBoss AOP et Spring AOP emploient cette technique) ou l'utilisation des fonctionnalités offertes par les machines virtuelles pour remplacer à la volée le pseudo-code des classes (Steamloom [19], JAsCo [166], [38], [22], CVM [65]).

Le tissage au chargement est la troisième technique à avoir été introduite. Il s'agit de modifier le pseudo-code des applications lors de leur chargement en mémoire, juste avant leur utilisation.

Mécanisme d'introduction

Les codes advice étendent le comportement des applications en se basant sur la dynamique de l'application, i.e. sur les points de jonction survenant au cours de l'exécution. Si aucun des points de jonction inclus dans la coupe ne survient, les codes advice ne seront jamais exécutés.

Le mécanisme d'introduction permet d'étendre une application en dehors de toutes considération sur son comportement. Ce mécanisme se rapproche donc plus de l'extension autorisée par l'héritage dans les langages objet. Il prend différentes appellations selon les langages : *inter-type declaration* en AspectJ, méthodes de rôles avec JAC, mix-in avec JBoss AOP et AspectWerkz.

Avec le mécanisme d'introduction, il s'agit de définir dans un aspect, des méthodes et/ou des attributs à introduire dans des classes cibles. On va pouvoir, par exemple, introduire un attribut

[3] Depuis sa version 1.2 datant de mai 2004, AspectJ effectue également du tissage au chargement. Par ailleurs, en janvier 2005, les équipes de développements d'AspectJ et d'AspectWerkz ont annoncé leur fusion ce qui à terme fournira à AspectJ un support pour le tissage à l'exécution.

`date` dans les classes `Facture` et `Commande` d'une application. Comme pour les codes advice, il s'agit de modulariser dans l'aspect une préoccupation (ici la datation des objets métier) que l'on ne veut pas disperser dans le reste de l'application.

3.2.4 Guide de développement et bonnes pratiques

Nous avons vu qu'un aspect définit un comportement (code advice) et des localisations (coupes) où ce comportement doit être injecté dans une application. À partir de là un certain nombre de bonnes pratiques peuvent être dégagées pour rendre plus efficace l'écriture d'aspect. Nous en détaillons deux dans cette section.

Isoler la logique d'intégration

Bien qu'il n'y ait pas de limite à la complexité du comportement que l'on peut coder dans un aspect, une bonne pratique consiste à déléguer au maximum ces traitements à l'extérieur de l'aspect. Par exemple, plutôt que de coder directement la logique d'accès à un SGBD dans un aspect de persistance (voir figure 3.1), on préfèrera implémenter cette logique dans des classes ou s'appuyer sur une API existante, puis désigner à l'aide de coupes les localisations nécessitant de la persistance. Cette bonne pratique est valable pour tous les services techniques que l'on souhaite implémenter sous forme d'aspect.

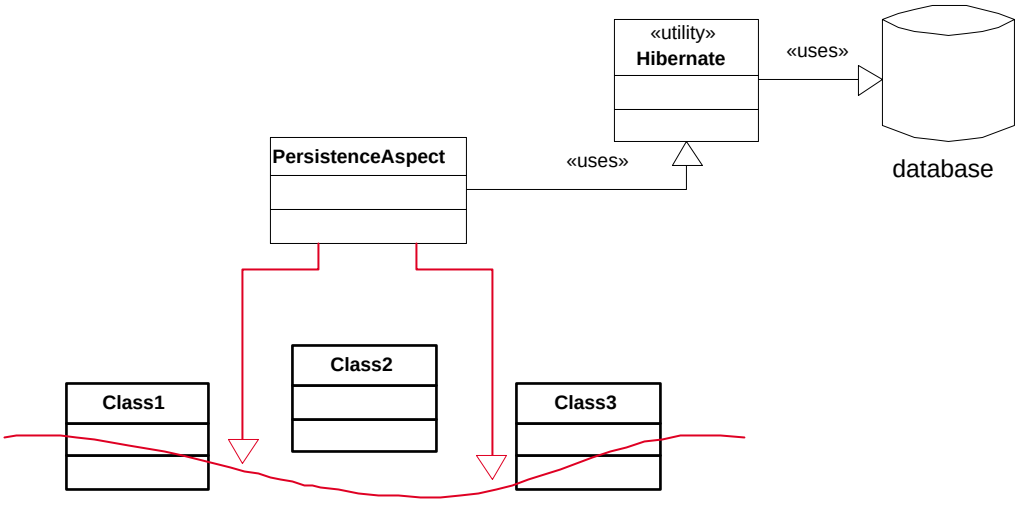

FIG. 3.1 – Intégration de service.

On obtient ainsi une double indépendance : indépendance de l'application vis-à-vis des services techniques, et indépendance des services techniques vis-à-vis des aspects. Les aspects se concentrent ainsi exclusivement sur la **logique d'intégration** en indiquant où injecter le service dans l'application. Loin d'être réductrice, cette vision de l'AOP que je défends, est au contraire très prometteuse. Elle permet en effet de faire le lien avec les composants logiciels et les langages de description d'architecture (ADL) et d'envisager une unification des notions de composant et d'aspect : le tissage fournit par les aspects est alors vu comme une liaison entre les composants de l'application et les composants implémentant les services.

Réutilisation d'aspects

Nous avons mentionné que les coupes et les codes advice sont les deux éléments majeurs qui définissent un aspect. Ils ne sont néanmoins pas égaux face à la réutilisation : les codes advice peuvent en général être réutilisés dans différentes applications, tandis que les coupes sont dépendantes d'une application donnée et doivent être quasiment systématiquement réécrites. Les langages et les frameworks existant abordent cette dichotomie dans la réutilisation de façon différentes : AspectJ introduit la notion d'**aspect abstrait** afin de concrétiser ultérieurement la définition des coupes, JAC, JBoss AOP et AspectWerkz externalisent cette définition dans des fichiers de configuration (texte brut pour JAC ou XML pour les deux autres).

3.3 La plate-forme Java Aspect Components (JAC)

JAC (Java Aspect Components) est une plate-forme open-source pour la programmation orientée aspect dynamique. Le projet a débuté en 1998 au laboratoire CEDRIC du CNAM dans le groupe dirigé par G. Florin, avec L. Duchien, R. Pawlak et moi-même. Le projet a fait par la suite l'objet de la thèse de Renaud Pawlak [128]. En 2003, JAC a rejoint le consortium ObjectWeb pour le *middleware open-source*. Entre temps, la société de services AOPSYS créée par R. Pawlak, L. Martelli et M. Pawlak a contribué fortement au développement de la base de code. JAC est un logiciel conséquent (environ 750 classes Java totalisant plus de 125 000 lignes de code). Au niveau international, JAC a été l'une des premières plates-formes d'AOP dynamique. Elle continue à faire partie des cinq plates-formes représentatives du domaine (avec AspectJ, JBoss AOP, AspectWerkz et Spring AOP).

Cette section présente le travail qui a été réalisé autour de la plate-forme JAC de 1999 à 2003. Elle en résume les caractéristiques principales. Une description plus détaillée peut être trouvée dans [148, 149][4] et [134] qui est l'article de référence sur la plate-forme (une version préliminaire est parue dans [133]). Par ailleurs une comparaison de JAC avec les autres approches majeures du domaine (AspectJ, JBoss AOP, AspectWerkz et Spring AOP) est fournie dans le chapitre 7 de [148, 149].

La section 3.3.1 introduit les problématiques majeures de recherche abordées par JAC. La section 3.3.2 présente l'architecture de JAC. Le modèle de programmation fait l'objet de la section 3.3.3. Les caractéristiques liées à la distribution sont présentées dans la section 3.3.4.

3.3.1 Problématiques abordées avec JAC

Cette section fait le point sur la façon dont JAC répond à différentes problématiques de recherche : l'adaptabilité, la dynamicité et la configurabilité.

3.3.1.1 Adaptabilité & dynamicité

Le tissage d'aspects dans JAC est dynamique. Il s'effectue à l'exécution et peut être distribué. D'un point de vue système, le tissage d'aspects dans JAC peut se ramener à une problématique de déploiement et de liaison :

- **Déploiement** : il s'agit d'installer des instances d'aspects qui offriront des services à une application éventuellement distribuée,

[4] [149] est la version anglaise mise à jour et complétée de [148].

- **Liaison** : il s'agit d'établir la liaison entre les objets applicatifs et les instances d'aspects. Contrairement aux liaisons habituelles avec les libraires, il s'agit ici d'une liaison d'interception : les instances d'aspect se lient aux objets applicatifs pour en intercepter les appels en fonction de la définition des coupes.

Comme pour toutes les problématiques de reconfiguration à chaud d'un système, cette dynamicité soulève de nombreuses questions :

- **quand** tisser un aspect ? Face à un système en cours d'exécution, il n'est pas évident de savoir si l'état courant est suffisamment cohérent pour autoriser une reconfiguration. Il faut pouvoir décider si on peut interrompre les opérations en cours, si il faut attendre leur fin, si il faut bloquer temporairement les nouvelles demandes d'opérations, etc. Ces questions se posent de façon encore plus aiguë lorsque l'application est distribuée comme cela peut être le cas avec JAC. Par ailleurs, le tissage pourrait également être paresseux de façon à ne tisser les aspects que lorsque l'application en a besoin.
- **comment** tisser un aspect ? Face à une opération s'étendant sur éventuellement plusieurs sites, des erreurs peuvent survenir au cours du déploiement et de la liaison d'un aspect. De même, plusieurs opérations de tissage et de détissage peuvent être déclenchées simultanément et conduire à des incohérences. Une solution serait alors de rendre le tissage transactionnel. Cette caractéristique du tissage pourrait être conçue sous d'aspect en prenant soin d'éviter une récursion infinie (ce nouvel aspect doit lui même être tissé).

Il est clair que les réponses que l'on peut apporter à ces problèmes améliorent la qualité du service mais ont un coût. L'approche adoptée par JAC a donc été la suivante : aucune vérification de cohérence, ni garantie transactionnelle ne sont fournies lors du tissage, mais les développeurs ont accès à un certain nombre d'éléments internes du framework qui leur permet de personnaliser le tissage. Par exemple :

- Tous les objets applicatifs sont référencés par un gestionnaire d'objets. Il est ainsi possible à tout instant de connaître la composition d'une application. Il en va de même des instances d'aspects qui sont sous la coupe d'un gestionnaire d'aspects.
- La méthode de tissage est redéfinissable individuellement pour chaque aspect.
- Chaque objet applicatif est associé à une API qui permet de le lier à un aspect.
- L'API de définition et de gestion des coupes peut être interrogé pour déterminer si une coupe s'applique à un objet applicatif.

Pour aller au delà de ces éléments de réponse mis en place dans JAC, une similitude importante apparaît entre les problématiques des intergiciels et celle du tissage : le déploiement et l'installation distante sont des problématiques d'actualité (voir par exemple les travaux autour d'OSGi, les API de déploiement dans J2EE, ou les travaux sur le déploiement transactionnel dans CCM [151]). Pour pouvoir déployer, il faut que les entités logicielles soient *packagées* de façon satisfaisante. Partant de là, on peut se demander, si le *packaging*, le déploiement et l'installation d'aspects, d'entités applicatives (objets ou composants), voire de services système, doivent être traitées séparément. Mon intuition est que la réponse à cette question est non. J'étayerai cette proposition dans le chapitre 4 où je défends le point de vue de l'unification des composants et des aspects d'une part, et du tissage et de la liaison de composants d'autre part.

3.3.1.2 Configuratibilité

Par configuration, nous entendons ici toute action qui permet d'adapter un logiciel aux besoins d'un utilisateur ou à un contexte d'exécution. Cette configuration peut être statique ou dynamique. Dans le premier cas, elle est appliquée au démarrage du logiciel, tandis que dans le second cas, elle peut être modifiée en cours d'exécution. La configuration est une caractéristique commune à de nombreux domaines de l'informatique, mais revêt une importance particulière en système. Les fichiers de configuration y pullulent et permettent de régler un grand nombre de caractéristiques allant de la définition des utilisateurs, aux paramètres réseau en passant par le fonctionnement des serveurs Web ou autres. Il en va de même dans les plates-formes J2EE ou CORBA. On peut distinguer la configuration de l'intergiciel lui-même, de la configuration des applications qui s'exécutent sur l'intergiciel. Ainsi les applications EJB sont accompagnées de fichiers XML qui permettent de configurer l'utilisation des services fournis par la plate-forme. On désigne ainsi la liste des méthodes devant être exécutées avec le service de transaction, les attributs persistants des composants, les rôles de sécurité ou les sources de données JDBC à utiliser pour la persistance. Le principe est identique avec CCM où un ensemble de DTD XML définissent le format de packaging des applications en vue de leur assemblage et de leur déploiement. Les fichiers XML qui décrivent un assemblage dans les modèles de composants comme Fractal peuvent également être vus comme des éléments participants à la configuration de l'application.

Dans tous les cas, l'objectif de la configuration est d'extraire un certain nombre d'informations de l'application et de les externaliser, en général, dans des fichiers. Par ce biais, on obtient un code plus générique, applicable à un plus grand nombre de cas d'utilisation. On obtient également des applications plus souples dont on peut modifier le comportement, dans les limites autorisées par les paramètres configurables, sans avoir à les recompiler.

Dans JAC, les entités configurables sont les aspects. Trois types d'information sont configurables à l'aide d'un mécanisme unique.

- Les **coupes**. Plutôt que de définir les coupes en dur dans le code des aspects, leur expression peut être reportée dans le fichier de configuration associé à l'aspect. C'est une caractéristique clé pour une bonne réutilisabilité des aspects. Comme nous l'avons mentionné à la section 3.2.3, les coupes sont dépendantes des applications, et seuls les codes advice sont réellement réutilisables en AOP. L'externalisation des coupes dans les fichiers de configuration est la solution adoptée par JAC pour la réutilisation : face à une nouvelle application pour laquelle on souhaite réutiliser un aspect, les coupes peuvent être modifiées dans le fichier de configuration sans avoir à recompiler l'aspect.
- Les paramètres **dépendant du contexte**. Ce sont les paramètres utilisés par les aspects et dont la valeur dépend de l'environnement dans lequel s'exécute l'aspect. Il s'agit par exemple de la source de données JDBC à utiliser pour un aspect de persistance ou de l'emplacement des fichiers de log pour un aspect de trace.
- Les paramètres **indépendant du contexte**. Ce sont les paramètres propres au fonctionnement des aspects. Ils sont indépendants de l'environnement d'exécution. Ils correspondent en général à différentes modalités de fonctionnement des aspects. Il s'agit par exemple du niveau de détail pour un aspect de trace ou des attributs transactionnels "à la J2EE" pour un aspect de transaction.

Ces trois types d'information configurable sont gérés à l'aide d'un mécanisme unique dans JAC. Toute classe implémentant un aspect est associée à une interface implicite composée par l'ensemble

de ses méthodes publiques. Une configuration est une suite d'appels aux méthodes publiques de l'aspect. Les fichiers de configuration sont donc des fichiers texte contenant une suite de noms de méthode (les méthodes à appeler pour configurer un aspect) associées aux paramètres de l'appel. À chaque instanciation de l'aspect (donc notamment lors du déploiement/tissage d'un aspect), ce fichier est interprété et les appels de méthodes correspondant sont émis sur l'instance créée. Le fichier de configuration fournit ainsi un script d'initialisation de l'aspect qui est exécuté en complément du constructeur.

3.3.2 Architecture

La fonctionnalité première de JAC est de fournir un support d'exécution pour des applications orientées aspects. En ce sens, JAC peut être comparé aux serveurs d'applications de type J2EE, CCM ou .Net dans lesquels des conteneurs hébergent les entités (composants) de l'application et leur fournissent des services techniques. Tout comme ces approches, JAC permet de séparer les services techniques de la logique métier de l'application. Par contre, JAC se différencie par le fait qu'aucun service n'est codé en dur, que de nouveaux services peuvent être développés et que les services peuvent être retirés ou ajoutés à chaud.

Dans le cas de JAC, tous les services sont programmés sous forme d'aspects. Il n'y a pas d'aspect prédéfini et tous les aspects sont des entités applicatives au même titre que les objets. Une application JAC est donc composée d'un ensemble d'objets et d'un ensemble d'aspects.

Lors du lancement, JAC est un **conteneur ouvert** sans aucun service hormis le chargeur. Les aspects, donc les services techniques, sont chargés à la demande en fonction des besoins des applications, qui sont elles-mêmes chargées selon le même principe. JAC apporte donc une solution souple dans laquelle seuls les services réellement nécessaires sont chargés. Par ailleurs, la gestion dynamique des aspects permet le retrait ou l'ajout à chaud de nouveaux services. Cette problématique est proche d'autres environnements adaptatifs comme la VVM [119, 120] qui permet de reconfigurer un environnement d'exécution en chargeant dynamiquement des descripteurs de jeux d'instructions (appelés VMlet) et des applications. Comparée à JAC, la granularité de l'adaptation est néanmoins plus fine avec la VVM : celle ci intervient au niveau du jeu d'instructions alors que JAC se place au dessus d'un environnement d'exécution existant (la machine virtuelle Java) et adapte l'application et ses services.

L'architecture d'un conteneur JAC comprend les entités suivantes :

- Chargeur : comme mentionné précédemment, cette entité est responsable du chargement des classes et des aspects. Concrètement, cette entité est un chargeur de classes (*class loader*) Java qui utilise la librairie BCEL [43] de manipulation de pseudo-code (*bytecode*). Les manipulations réalisées préparent les classes applicatives en vue de leur tissage à l'exécution. Il s'agit d'insérer avant chaque méthode un code qui, si la méthode est aspectisée, intercepte l'invocation et la transmet à l'aspect (ou les aspects) concerné.
- Gestionnaire d'aspects : cet entité enregistre les aspects chargés dans le conteneur. Chaque aspect est représenté par une instance unique (singleton) et est associé à un ensemble de coupes. Celles-ci sont définies par le programmeur et fournissent les informations sur les localisations où l'aspect s'applique. Comme nous le verrons à la section 3.3.4, le gestionnaire d'aspects joue un rôle important dans le cadre de la répartition avec les notions d'aspect et de coupe distribués.
- Référentiel d'applications : cet entité enregistre les applications chargées dans le conteneur.

Chaque application est associée aux aspects qui sont tissés à un instant donné, sur l'application.

- Référentiel de méta-données : à la manière de Java 5 ou de C#, JAC fournit un service appelé RTTI (*Run Time Type Information*) pour gérer des annotations. Ce service permet d'attacher des méta-données aux éléments (classe, attribut, méthode) des applications et des aspects. Ces méta-données sont des marqueurs sémantiques (par exemple, pour les attributs persistants d'une classe, ou pour les classes accessibles à distance) placés par les développeurs et exploitables par les aspects pour orienter le tissage.
- Orchestrateur d'aspects : lorsque plusieurs aspects s'appliquent sur un même point de jonction, l'orchestrateur d'aspects est chargé de définir l'ordre d'application de ces aspects, plus précisément, dans le cas de JAC, des wrappers (voir section 3.3.3.4). Cet ordre est fourni par le développeur via un paramètre de configuration. Comme dans d'autres approches (par exemple AspectJ), l'ordre est global dans JAC : il s'applique de façon identique pour tous les aspects d'une application.

3.3.3 Modèle de programmation

Contrairement à AspectJ qui étend la syntaxe du langage Java pour l'expression des aspects[5], JAC est un framework Java pur qui fournit une API pour utiliser l'ensemble des concepts liés aux aspects. Cette API permet par exemple de définir des coupes, de tisser des aspects ou d'en retirer. L'ensemble de ces opérations se déroule lors de l'exécution de l'application. Les aspects sont des entités qui ont une existence propre lors de l'exécution (concrètement des instances d'aspects cohabitent avec les objets applicatifs dans la JVM). Nous fournissons ci-dessous un aperçu du modèle de programmation de JAC.

3.3.3.1 Aspect

Comme dans les autres approches de programmation orientée aspect, un aspect, appelé *aspect component* (AC) en JAC, implémente une préoccupation transversale. Contrairement à AspectJ où un aspect définit des coupes et des codes advice, JAC sépare ces deux définitions : les coupes sont définies dans les aspects et les codes advice sont définis dans des wrappers (voir section 3.3.3.4). En effet, la nature de ces deux concepts est assez différente. Les codes advice sont uniquement liés à l'aspect, ils en définissent le comportement. Par contre, les coupes sont liées à l'application : elles définissent où l'aspect doit être intégré dans l'application. Cela confère à la coupe un caractère éminemment fragile. Dès lors que l'application évolue ou que l'on change d'application, la coupe n'est plus valable et doit être réécrite. Finalement, seuls les codes advice sont réutilisables. AspectJ résout ce problème avec la notion d'aspect abstrait. Nous avons choisi de séparer clairement la définition des coupes et des codes advice. Cette approche est reprise par les autres frameworks AOP dynamique du domaine (JBoss AOP, AspectWerkz).

Dans JAC, les aspects sont simplement des classes qui étendent une classe de base (`AspectComponent`) fournie par le framework.

Configuration

[5] Cette distinction tend à s'estomper : les versions d'AspectJ en cours de développement (AspectJ 5) proposent un modèle de programmation à base d'annotations qui rend obsolète cette syntaxe ad-hoc.

En plus de la définition des coupes, les aspects JAC jouent un rôle important dans le cadre de la configuration. Souvent laissée de côté dans les modèles de programmation, cette problématique est néanmoins primordiale dès lors que l'on souhaite fournir des logiciels facilement réutilisables et adaptables à des contextes d'utilisation variés.

Les plates-formes pour les composants distribuées J2EE ou CCM résolvent le problème de la configuration en fournissant un ensemble de DTD XML qui permettent de configurer les applications : on y trouve soit des paramètres propres à l'application, soit des paramètres liés aux services techniques fournis par la plate-forme : attributs persistants, méthodes devant être exécutées de façon transactionnelle, etc. De même que les services fournis par ces plates-formes sont fixes, les paramètres de configuration disponibles sont figés.

Dans JAC, les services fournis par la plate-forme étant programmés sous forme d'aspect, l'éventail des paramètres de configuration est nettement plus important et ne peut être connu a priori. De ce fait, la configuration ne peut être imposée par la plate-forme mais doit être du ressort des aspects eux-mêmes. Chaque aspect doit donc fournir une interface de configuration (en fait, l'ensemble de ces méthodes publiques). Chaque aspect est associé à un script de configuration qui est un fichier texte contenant un ensemble d'appels aux méthodes de l'interface de configuration. Lors de l'instanciation de l'aspect, la plate-forme JAC charge en plus le script de configuration et l'exécute. Il n'y a donc pas comme dans J2EE ou CCM, un seul fichier XML de configuration, mais autant de fichiers de configuration qu'il y a d'aspects. Notons enfin qu'un fichier de configuration peut être changé et pris en compte sans avoir à recompiler l'aspect. Par ailleurs, la configuration d'un aspect peut également être modifiée dynamiquement après l'instanciation de l'aspect.

Les fichiers de configuration d'aspect jouent un rôle clé dans JAC : ce sont grâce à eux que les aspects de la librairie (voir section 3.3.3.6) peuvent être facilement réutilisés.

3.3.3.2 Point de jonction

Les points de jonction représentent les localisations où un aspect peut se greffer à un objet. Ils matérialisent en quelque sorte une interface entre l'aspect et l'objet. En dehors des points de jonction, il n'y a pas d'interaction possible entre l'aspect et l'objet.

Les langages AOP statiques supportent en général plus de types de points de jonction différents que les langages dynamiques. Par exemple, AspectJ en supporte neuf[6]. JAC n'en supporte que deux : exécution de méthode, exécution de constructeur. Cette différence s'explique par le fait que, en travaillant à la compilation, AspectJ peut effectuer une analyse syntaxique poussée du code et extraire un nombre important d'informations sur les points de jonction. A contrario, en étant dynamique, JAC est contraint par les mécanismes liés à l'exécution des objets : essentiellement les messages correspondant aux exécutions de méthodes ou de constructeurs.

Lorsqu'un point de jonction survient dans l'exécution d'un programme, une interface, dite API d'introspection, fournit à l'aspect des informations contextuelles sur ce point de jonction. L'API d'introspection de JAC est illustrée dans la partie gauche de la figure 3.2. Cette API est issue d'une initiative *open source* AOP Alliance[7] dont le but est de proposer un standard pour les interfaces d'interception de différentes plates-formes AOP dynamiques. À l'heure actuelle, outre JAC, cette API est implémentée par les plates-formes Spring AOP, dynaop et JoyAop.

[6] appel/exécution de méthode, lecture/écriture d'attribut, exécution de constructeur, exécution de constructeur hérité, début de bloc `catch`, exécution de bloc `static`, exécution de code advice.

[7] `aopalliance.sf.net`

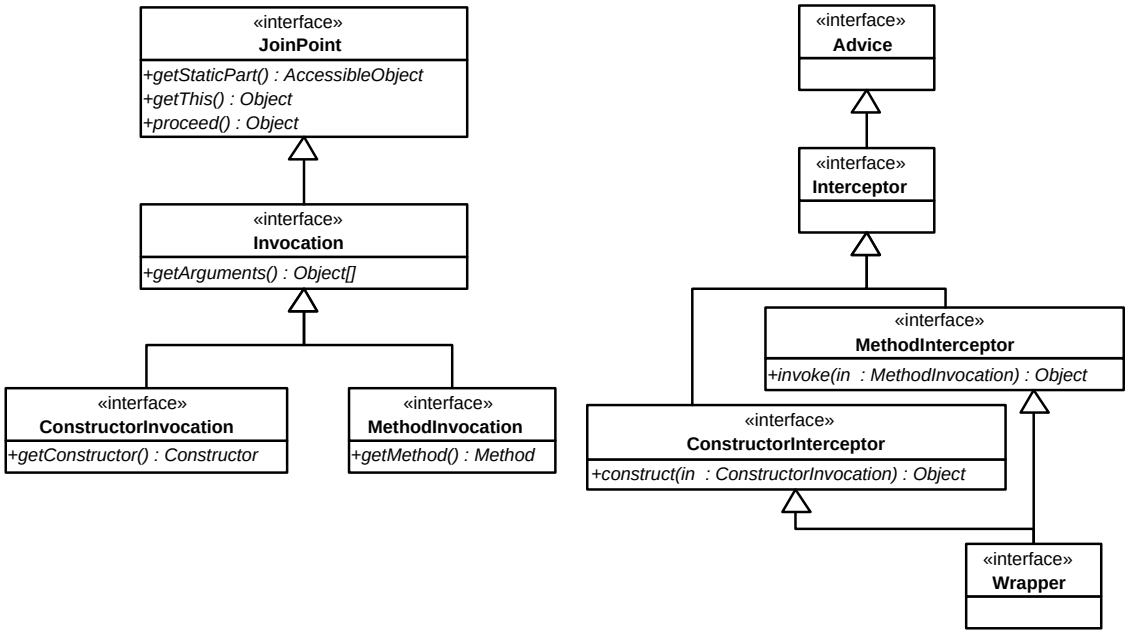

FIG. 3.2 – API AOP Alliance d'introspection et d'interception.

3.3.3.3 Coupe

Les coupes permettent, à l'aide d'un langage de *patterns*, de construite des expressions qui désignent des ensembles de points de jonction. Ceux-ci correspondent alors aux localisations où un aspect s'applique.

Les coupes JAC sont définies à l'aide d'expressions régulières. De part la nature objet du langage Java, trois types expressions ont été retenues. Elles concernent respectivement les classes, les objets et les méthodes. Un quatrième type d'expression intervient dans le cadre de la répartition comme nous le verrons à la section 3.3.4. Leur syntaxe de ces expressions est celle des expressions GNU regexp[8] ; elle comprend des *wildcards* et des opérateurs logiques. Ces expressions permettent de filtrer l'ensemble des objets d'une application pour sélectionner ceux où l'aspect s'applique. Cette sélection est donc éminemment contextuelle : en fonction du moment où le tissage est effectué, les objets de l'application peuvent changer et la même coupe ne sélectionnera pas le même ensemble d'objets. Les trois expressions régulières des coupes JAC sont :

- expression de classe : à partir des noms de classes et de package, cette expression permet de filtrer l'ensemble des classes de l'application ;
- expression de méthode : à partir des signatures des méthodes (nom, paramètres, type de retour), cette expression permet de filtrer l'ensemble des méthodes de l'application ;
- expression d'objet : cette expression permet de filtrer l'ensemble des objets de l'application. Elle s'appuie sur un schéma de nommage des objets applicatifs. Ces noms sont attribués par le développeur ou, à défaut, sont inférés automatiquement par JAC à l'aide d'un schéma de désignation reposant sur le nom de la classe et d'un compteur d'instance.

Les expressions précédentes permettent de faire une sélection syntaxique sur les objets à inclure dans une coupe. JAC fournit également un ensemble d'opérateurs sémantiques pour raisonner sur

[8] `www.cacas.org/java/gnu/regexp/`

le comportement des méthodes. L'idée est de pouvoir identifier le comportement d'une méthode vis-à-vis d'un attribut et de savoir par exemple si cette méthode accède ou modifie un attribut. Ces informations sont importantes, par exemple, dans les aspects de persistance lorsqu'il s'agit de déterminer si l'état d'un objet a été mis à jour. Cette analyse sémantique est effectuée automatiquement par JAC lors du chargement des classes en mémoire : un analyseur de bytecode parcourt chaque méthode et positionne des annotations dans le référentiel de méta-données. Lors de la définition d'une coupe, des opérateurs sont disponibles pour interroger ces méta-données. Les opérateurs, au nombre de six, sont définis en fonction des différentes manipulations pouvant être opérées sur l'état d'un objet : GETTER (la méthode retourne la valeur d'un attribut), SETTER (la méthode positionne la valeur d'un attribut), MODIFIER (la méthode modifie un attribut), ACCESSOR (la méthode lit un attribut), ADDER (la méthode ajoute des éléments dans une collection), REMOVER (la méthode retire des éléments d'une collection).

Ces opérateurs sémantiques sont une caractéristique originale de JAC. À notre connaissance, ils n'existent dans aucune autre plate-forme. Ils rendent les coupes plus robustes aux changements : plutôt que de s'appuyer sur des conventions de nommage pour déterminer la nature des méthodes, comme dans les modèles Java Beans ou Hibernate, cette information est déduite du comportement réel des méthodes. Les développeurs sont ainsi libre de changer les noms des méthodes tout en ayant la garantie que les coupes définies continuent à être valables.

3.3.3.4 Wrapper

Alors que les coupes définissent le "où" d'un aspect, les codes advice répondent à la question "quoi", en d'autres termes, quel est le comportement d'un aspect. Les frameworks AOP préfèrent à l'expression code advice, des termes qui mettent plus en avant le caractère du lien dynamique entre les objets et les aspects. Ainsi, JBoss AOP parle d'intercepteur, tandis que JAC emploie le terme wrapper. Ce dernier remonte aux travaux sur les design patterns. [58] le présente comme une variante des patterns adaptateur et décorateur : il s'agit d'"entourer" un objet de base pour adapter l'interface qu'il présente à l'extérieur et/ou étendre son comportement. Par la suite, les wrappers ont connu de nombreuses variantes dont [29], les travaux sur les *composition filters* [4] et leur adaptation aux composants (voir nos travaux présentés au chapitre 4).

Les wrappers JAC permettent de définir du code avant/après qui sera exécuté autour des points de jonction associés à l'aspect. Les wrappers sont des classes Java qui implémentent l'interface `Wrapper` (voir figure 3.2). Contraiment à AspectJ qui définit cinq types de code advice (before, after, around, after returning, after throwing), il n'y a qu'un seul type de wrapper en JAC : around. En fait, ce type est le plus général et recouvre tous les autres.

Contrairement à une classe applicative, un wrapper JAC n'est jamais instancié directement par le développeur. C'est le framework qui, en fonction des coupes définies, crée les instances de wrapper et les lie aux objets. Le développeur peut néanmoins contrôler la cardinalité des wrappers lors de la définition d'une coupe en précisant si le wrapper est un singleton (la même instance est partagée par tous les objets aspectisés), ou si une instance différente de wrapper doit être créée pour chaque objet aspectisé.

De même, un wrapper n'est jamais invoqué directement par le développeur. Le framework est responsable de la gestion du mécanisme d'indirection entre un point de jonction et le ou les wrappers associés. Lors de cette indirection, les informations d'introspection sont transmises au wrapper (paramètre `MethodInvocation` dans la signature de la méthode `invoke` figure 3.2).

L'instance contenant ces informations fournit également une méthode `proceed` qui comme avec AspectJ, permet d'invoquer l'objet aspectisé et délimite ainsi, dans le wrapper, les parties de code avant et après.

3.3.3.5 Profil UML

Le modèle de programmation de JAC est associé à une profil UML qui permet d'exprimer graphiquement la conception d'une application comportant des aspects. Ce profil [129] concerne les diagrammes de classe. Il comprend deux stéréotypes principaux : `«aspect»` et `«pointcut»`. Le premier permet de créer un aspect, et le second qualifie une relation entre un aspect et des classes comme étant la définition d'une coupe. Cette relation comprend en plus des paramètres pour préciser l'expression associée à la coupe. Les méthodes définies dans un aspect représentent des wrappers. Elle sont qualifiées en fonction du type de wrapper : `«before»`, `«after»`, `«around»` ou `«replace»` (wrapper de type *around* ne faisant pas appel au code de base). Ce profil UML est intégré à UMLAF (*UML Aspectual Factory*) qui l'environnement de conception et de programmation associé à JAC.

La figure 3.3 illustre ce profil. L'aspect `Authentication` comporte deux wrappers *before*, `askAuthInfos` et `checkAuthinfos`. Le premier aspectise dans la classe `Client` toutes les invocations d'une méthode de la classe `Server`, tandis que le second aspectise toutes les exécutions de méthodes de la classe `Server`. À partir de diagrammes de ce type, des squelettes de code pouvant être complétés par le programmeurs, sont générés par UMLAF.

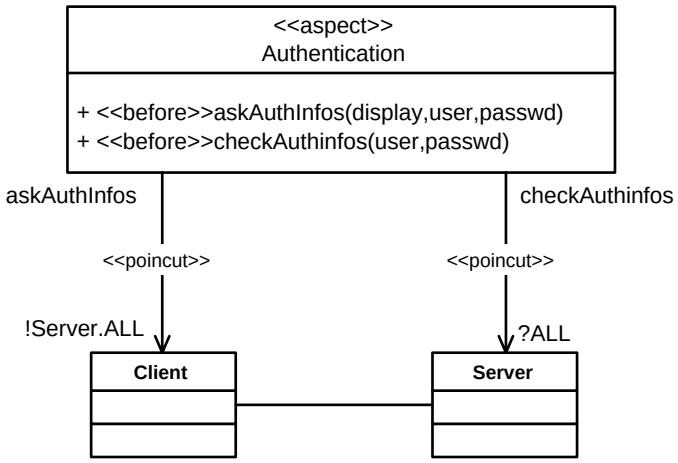

FIG. 3.3 – Notation UML pour des diagrammes de classes et d'aspects.

Notons que les travaux autour d'UML et des aspects sont nombreux. On peut citer notamment ceux sur l'application des aspects à d'autres diagrammes UML comme les cas d'utilisation (*use case*) [76][135] ou les diagrammes de séquence. Au delà le tissage de modèles est un domaine de recherche d'actualité avec de nombreuses proposition (par exemple, [70] et [30]). Loin de vouloir proposer une solution complète, notre profil est une tentative pragmatique et simple pour couvrir l'utilisation des aspects dans les diagrammes de classes.

3.3.3.6 Librairie d'aspects

JAC fournit une librairie de quinze aspects qui implémentent les services techniques nécessaires de façon courante pour le développement d'applications client/serveur 3 tiers. Certains services n'implémentent pas directement de services mais suivent le principe de séparation de la logique d'intégration énoncé à la section 3.2.4 : ils s'appuient sur une librairie externe et intègrent ce service à l'application. On a ainsi pu réutiliser plusieurs services existants dont le moteur transaction JOTM[9] du consortium ObjectWeb et le mapping objet/relationnel Hibernate[10].

Un aspect implémentant le patron MVC (*Model View Controller*) est également disponible pour construire les IHM des applications JAC. Cet aspect est disponible en deux versions : Java Swing et HTML/servlet. Il est ainsi possible très rapidement de construire des applications soit en mode client lourd, soit en mode client léger.

Les aspects liés à la programmation distribuée sont présentés dans la section suivante.

3.3.4 Support pour la programmation distribuée

JAC fournit en standard un certain nombre de fonctionnalités pour le développement d'applications distribuées. Ces fonctionnalités se manifestent à trois niveaux : architecture, modèle de programmation et librairie d'aspects. Le principal résultat est celui de coupe distribuée. Il permet de regrouper dans une même coupe des points de jonction situés sur des sites distants. C'est un résultat original qui distingue clairement JAC des autres outils de programmation orientée aspect et qui depuis a été repris dans [118].

Architecture pour la distribution Nous avons mentionné à la section 3.3.2 que JAC se présente sous la forme d'un conteneur dont l'interface offre différents services, entre autres le chargement des aspects et des applications et le tissage. Cette interface, accessible en local dans un fonctionnement normal, est également accessible à distance via deux protocoles : Java RMI et CORBA (avec l'ORB JacORB [26]). Il est ainsi possible d'invoquer à distance un conteneur JAC en lui transmettant une application et des aspects et de déclencher le tissage.

En fonctionnement distribué, les conteneurs JAC fonctionnent comme des démons installés sur chacun des sites de l'environnement. Les interactions entre ces conteneurs permettent d'assurer le fonctionnement de l'application et de réaliser les opérations de gestion telles que le tissage.

Modèle de programmation pour la distribution Nous avons mentionné à la section 3.3.3.3 que les coupes JAC comprennent trois expressions régulières (sur les noms de classes, d'objets et sur les signatures de méthodes) qui permettent de sélectionner les localisations où s'applique un aspect.

Dans le cadre de la distribution, une quatrième expression est ajoutée : il s'agit de l'expression d'hôte. C'est une expression régulière sur les noms des conteneurs. La forme concrète de ces noms dépend de la technologie employée : URL RMI pour Java RMI et nom de type `corbaname` pour CORBA. Cette expression supplémentaire permet de désigner des points de jonction localisés dans des conteneurs différents (voir figure 3.4). On aboutit ainsi à la notion de **coupe distribuée** dont le principe a été repris dans d'autres approches (voir notamment [118]).

[9] `jotm.objectweb.org`
[10] `hibernate.bluemars.net`

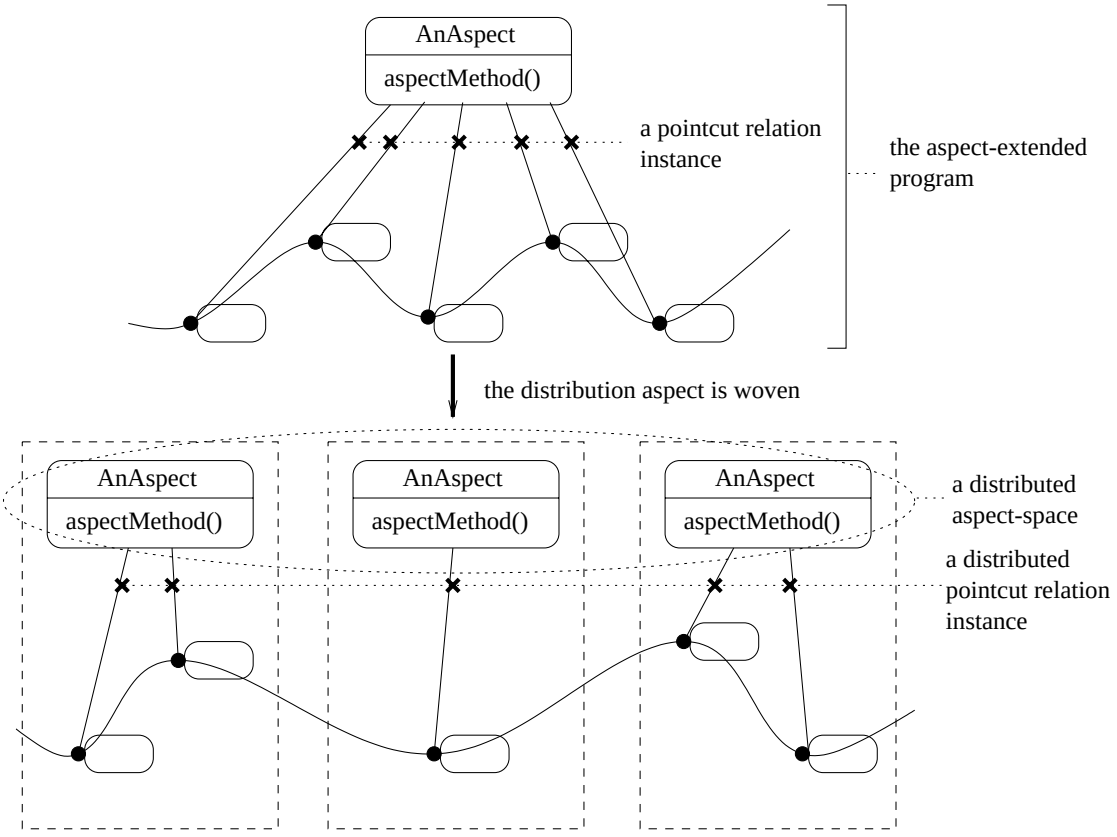

FIG. 3.4 – Notion de coupe distribuée.

Techniquement la mise en œuvre de la notion de coupe distribuée repose sur le gestionnaire d'aspects mentionné à la section 3.3.2. Cette entité est répliquée sur tous les conteneurs JAC (cela impose que la topologie de l'environnement soit connue) et ses données (la définition des coupes notamment) soient maintenues en cohérence : toute modification sur une réplique est propagée sur les autres. Ainsi tous les conteneurs ont à leur disposition la définition de toutes les coupes et connaissent les points de jonction de ces coupes. Par défaut, les répliques du gestionnaire d'aspects n'implémentent pas de politique de cohérence mémoire : potentiellement, des incohérences peuvent survenir si la même coupe est modifiée simultanément sur des conteneurs distincts. Cet écueil ne nous semble pas rédhibitoire pour deux raisons ;

1. la définition ou la modification d'une coupe est une opération peu fréquente. Elle a lieu au lancement de l'application puis s'apparente à une opération de reconfiguration. On suppose que celles-ci sont peu fréquentes. Par ailleurs, elle s'opère quasiment systématiquement à partir d'un seul site d'administration ce qui limite les risques d'incohérence.

2. si malgré tout, les hypothèses précédentes ne sont pas vérifiées, il est possible d'implémenter une politique de cohérence mémoire plus stricte pour le gestionnaire d'aspects. En effet celui-ci est considéré par le framework JAC comme un objet applicatif et peut être à ce titre aspectisé par un aspect de cohérence mémoire.

Librairie d'aspects pour la distribution Le principal aspect lié à la distribution est celui de déploiement et de communication distante. À partir de la description d'une topologie de sites distants, il permet d'installer les objets d'une application et les souches de communication en fonction des directives du développeur. Cet aspect existe en deux versions différentes, Java RMI et CORBA (avec l'ORB JacORB [26]). Cet aspect opère en deux temps :

1. **Déploiement des objets applicatifs**. L'aspect commence par installer l'application localement, puis procède au déploiement en recopiant à distance l'état des objets. Par exemple sur la figure 3.5, l'objet `o2` a été déployé sur l'hôte 2 où il y est représenté par l'objet `remote o2`. Notons que cet aspect permet également d'installer des répliques d'un même objet sur différents sites.

2. **Déploiement des souches**. En fonction des opérations d'installation précédentes, des souches et des wrappers sont installés. Par exemple, sur le scénario de la figure 3.5, `o2` le représentant local de `remote o2` est associé à un wrapper qui intercepte les invocations de méthode et les transmet à une souche cliente (qui elle-même les transportent jusqu'à la souche serveur, qui les délivre à l'objet). Lors du retour, le résultat suit le chemin inverse jusqu'à `o1`.

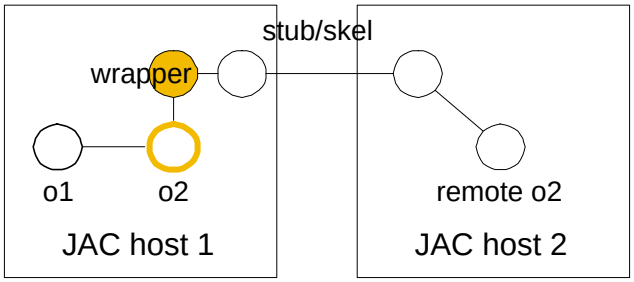

FIG. 3.5 – Aspect de communication distante.

Sur le même principe que les versions Java RMI et CORBA, Luc Teboul [172, 173], stagiaire de DEA (co-encadrement avec E. Gressier-Soudan), a réalisé en 2002 une version de l'aspect de déploiement et de communication distante basée sur la norme TASE.2 [72]. Ce protocole, de type messagerie industrielle, a permis de montrer que les concepts mis en place dans JAC sont applicables à d'autres paradigmes de communication que ceux des intergiciels requête/réponse.

Par ailleurs, toujours dans le domaine de la répartition, Frédéric Loiret [97, 98], stagiaire de DEA (co-encadrement avec E. Gressier-Soudan), a implémenté en 2003, un aspect de cohérence mémoire (protocole de Li & Hudak [96]) pour des documents répliqués. Cet aspect se décompose en six sous-aspects : communications distantes, localisation, invalidation, verrouillage, cohérence et journalisation. Cette implémentation orientée aspect a également été comparée à deux autres implémentations à base de composants Fractal [27] et Kilim [71].

Finalement, pour compléter ce panorama des fonctionnalités liées à la répartition présentes dans JAC, nous pouvons mentionner que des aspects d'équilibrage de charge et de cache sont également disponibles à l'état de prototype.

3.4 Aspects pour les plates-formes à composants

La section précédente a présenté les résultats obtenus au cours du développement de la plate-forme AOP JAC. Nous avons construit une plate-forme AOP basée sur le principe d'AOP dynamique. Cette plate-forme peut être utilisée dans n'importe quel domaine applicatif. Elle possède néanmoins un ensemble de caractéristiques spécifiques pour la programmation distribuée. Dans cette section, nous présentons des résultats liés à l'utilisation de l'AOP en tant que modèle de programmation pour faciliter le développement d'applications à base de composants EJB et CCM. Nous avons obtenu les deux résultats suivants :

- Le code d'une application EJB ou CCM comporte de nombreuses préoccupations liés aux API des plates-formes, aux concepts de ces modèles de programmation ou aux services systèmes (nommage, transaction, persistance, etc.) fournis par les plates-formes. Nous avons montré que des principes généraux peuvent être dégagés afin de donner une vue commune au développement applicatif pour ces deux types de plates-formes. Ces principes communs concernent par exemple, l'interaction avec le service de nommage ou la construction et la recherche d'instances avec une fabrique (*factory*).
- Ces principes communs, regroupés dans une couche abstraite programmée sous forme d'aspect, peuvent être utilisés pour faciliter le développement de certaines parties d'applications. Par exemple, cette couche a servi à réaliser un vérificateur de contrats comportementaux et un répartiteur de requêtes. Ces deux fonctionnalités dépendent uniquement de la couche abstraite. Elles peuvent être appliquées, selon la personnalité choisie, indifféremment à un des deux types de plate-forme : EJB ou CCM.

Ce travail a été mené dans le cadre de la thèse de doctorat de F. Legond-Aubry [94] dont j'ai assuré l'encadrement (directeur G. Florin) de 2003 à 2005. Cette thèse comporte d'autres résultats, notamment sur l'introduction de la notion de contrat dans les modèles de composants, que nous ne présentons pas ici (voir [94]). Les travaux présentés ici ont fait l'objet de la publication [95].

3.4.1 Contexte

Le contexte de cette étude est celui des applications pour les plates-formes à composants EJB [20] et CORBA CCM [162].

Le développement de ces applications demande la maîtrise de nombreux concepts (en termes de nommage, transaction, persistance, etc.) et la connaissance d'un certain nombre d'API. Il en résulte une difficulté importante pour le développeur lambda. De plus, ces différentes préoccupations doivent être adressées de front. Cela conduit à un code très entremêlé. À titre d'exemple, le code suivant[11] est celui d'une méthode banale d'un composant EJB positionnant la valeur d'un attribut `X`. On constate que chaque ligne correspond à une préoccupation différente, ce qui nuit à sa compréhension, sa mise au point et son développement.

```
1 public void setX(int value)
2 throws RemoteException {              // communications distantes
3    if (user.getAuthentification())    // sécurité
4    tx.beginTransaction();             // transaction
5    // code JDBC : stockage de X       // persistance
```

[11] Notons en plus que ce code est simplifié : il ne contient ni les instructions de récupération de l'utilisateur courant (`user`), ni celle de la transaction (`tx`). Le code réel est plus conséquent.

```
6    tx.commitTransaction();            // transaction
7 }
```

Face à ce problème, qui est de même nature avec CCM, l'AOP a très tôt été envisagée comme une technique pour démêler le code des applications. Dans [148, 149] (chapitre 11), nous avons montré, sur le cas concret d'une application J2EE existante, que les aspects permettent de simplifier le code des composants EJB. Différents aspects sont proposés pour les façades session et entité, la résolution des fabriques, la gestion des pré-conditions, de la persistance et des transactions. Notons que pour ces deux derniers aspects, le code des composants n'est pas entièrement dépourvu de persistance et de transaction : le développeur continue par exemple à démarquer ses transactions sans avoir à connaître les détails de l'API JTA de J2EE.

Dans la section suivante, nous poursuivons cette réflexion et nous l'étendons à la plate-forme CCM. Nous montrons que les applications EJB et CCM suivent un schéma commun qui peut être abstrait dans une couche neutre programmée à l'aide d'aspects. La vision macroscopique de cette approche est résumée par la figure 3.6.

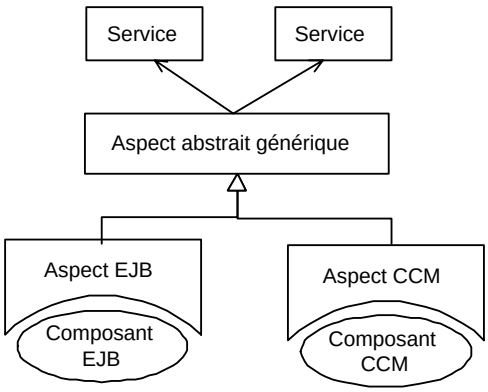

FIG. 3.6 – Schéma de principe.

3.4.2 Structure applicative commune

Nous avons identifié quatre phases principales dans la structure d'une application EJB ou CCM : découverte de service, obtention d'instances de composants, invocation de service et relaxation des instances. Elles sont résumées sur la figure 3.7. La figure mentionne des appels J2EE. Une figure identique pourrait être faite pour CCM.

La première phase concerne la **découverte des services de base**. Elle permet à l'application d'obtenir les références des services fournis par la plate-forme : nommage, transaction, etc. Pour CCM, il s'agit d'initialiser l'ORB, puis de récupérer les services en invoquant `resolve_initial_references`. Pour J2EE, il s'agit de récupérer le contexte de nommage par défaut en instanciant la classe `InitialContext`.

Obtention d'instances de composants La phase suivante consiste à obtenir la référence du ou des composants avec lesquels on souhaite travailler. J2EE et CCM s'appuient pour cela sur la

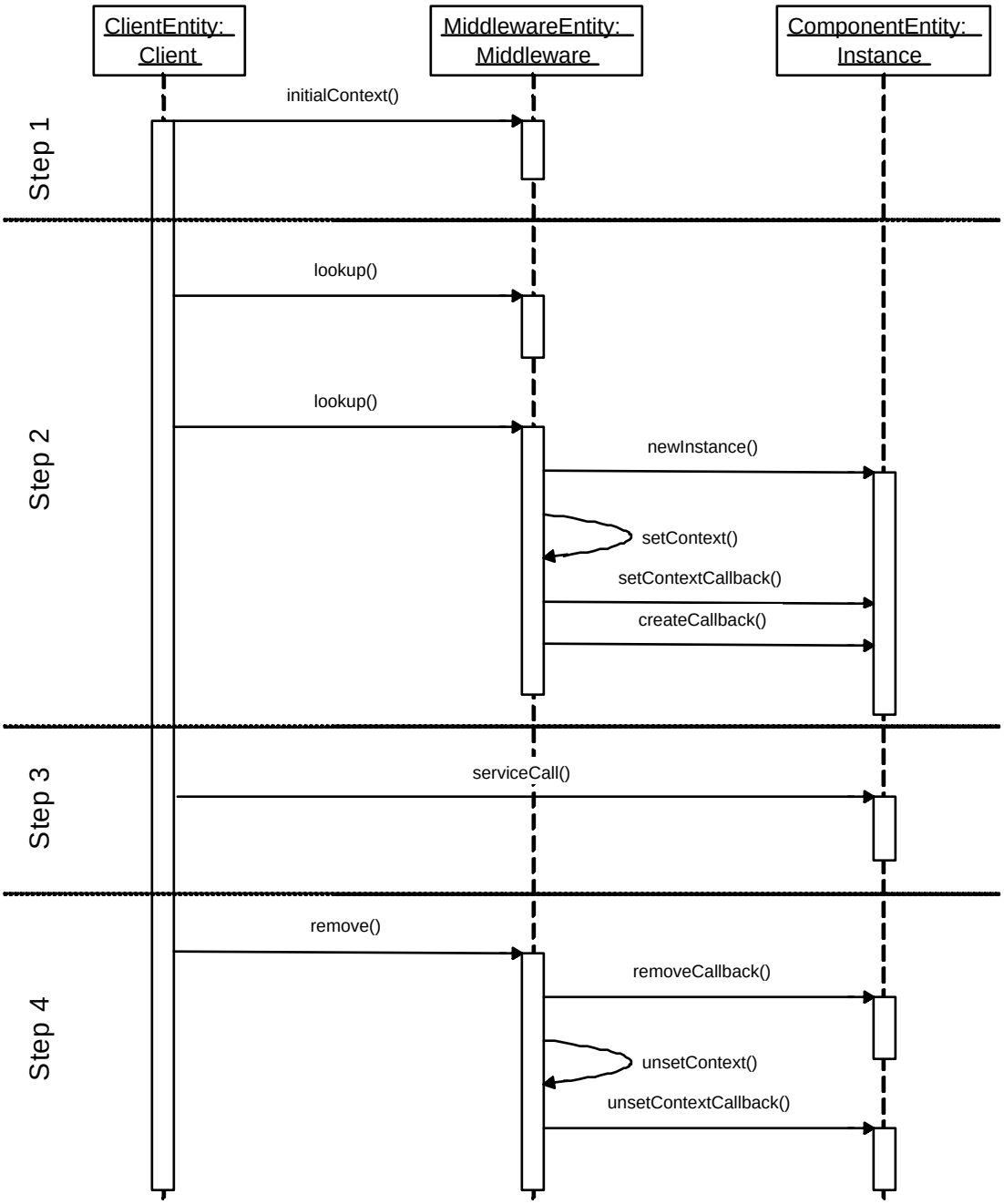

FIG. 3.7 – Phases types de l'exécution d'une application EJB.

notion de fabrique (*factory*) de composants. Chaque fabrique fournit des méthodes pour créer de nouveaux composants ou pour rechercher des instances de composants existants. Ces recherches se font en général à partir d'une clé primaire, mais peuvent aussi suivre d'autres modalités.

Invocation du service Cette phase contient l'invocation des méthodes fournies par le composant. Elle dépend des traitements que l'on souhaite faire exécuter aux composants.

Relaxation des instances de composants Il s'agit dans cette dernière phase de libérer les références acquises sur les composants. Cette phase permet à la plate-forme de libérer les ressources acquises pour gérer le composant.

3.4.3 Aspect et mappings

L'implémentation de la couche abstraite repose sur AspectJ [83]. Un ensemble de coupes abstraites sont définies. Chaque coupe capture un événement correspond aux phases identifiées précédemment. Deux versions de chacune de ces coupes existent : une pour EJB et une pour CCM (voir tableaux 3.1 et 3.2). L'aspect n'a pas de comportement propre (i.e. pas d'advice code). Il s'agit juste d'un conteneur de coupes qui "sait" intercepter des événements, mais ne fait rien.

L'utilisation de la couche abstraite se fait par extension de l'aspect précédent. On associe aux coupes un comportement en fonction des besoins. Ce ou ces comportements seront exécutés avant et/ou après les événements interceptés par les coupes.

[94] présente deux services développés avec cette couche abstraite. Le premier est un service de vérification de comportement. Chaque composant est associé à un automate état-transition représentant une spécification de son comportement. Les transitions sont étiquetées avec des noms de méthodes. Un chemin dans l'automate représente donc un enchaînement légal d'exécution de méthodes. Le service vérifie que le comportement du composant est conforme à sa spécification et lève une exception sinon. Ce service utilise les coupes `clientside_componentCall` et `serverside_componentCall`.

Le deuxième service développé est un service de répartition de charge pour composants. Il utilise les coupes `clientside_middlewareBinding` pour configurer les sites mirroirs, et `clientside_creation` et `clientside_lookup` pour choisir un réplicat lors de la création ou la recherche d'un composant.

3.5 Conclusion

Ce chapitre présente une synthèse de mes travaux de recherche sur la programmation orientée aspect. Après avoir résumé les concepts de base à la section 3.2, la section 3.3 présente la plate-forme orientée aspect dynamique JAC et la section 3.4 une couche d'abstraction orientée aspect pour la programmation d'applications à base de composants CCM et EJB.

JAC est un framework Java fournissant un support pour l'ensemble des concepts de l'AOP (aspect, coupe, etc.). Plusieurs résultats originaux peuvent être mis en avant.

Tout d'abord, JAC a été l'une des premières plates-formes internationales implémentant le concept d'AOP dynamique, c'est-à-dire fournissant un support d'exécution dans lequel les aspects peuvent être ajoutés et retirés dynamiquement pendant l'exécution de l'application. Le projet a

Étape	Nom de la coupe	Rôle	Expression AspectJ de la coupe
1	clientside_middlewareBinding()	client	call(javax.naming.InitialContext.new (..)) ;
1	serverside_middlewareBinding()	serveur	ϕ (le serveur est automatiquement lié au serveur au chargement)
2	clientside_lookup()	client	call(* javax.naming.Context.lookup (String)) ;
2	serverside_lookup()	server	ϕ (un serveur ne cherche pas, il est recherché)
2	clientside_narrowing()	client	call(* javax.rmi.PortableRemoteObject.narrow (..)) ;
2	serverside_narrowing()	server	ϕ (un serveur ne fait pas de narrow)
2	clientside_create()	client	! withinEJBMiddleware() && call(* javax.ejb.EJBHome+.create (..)) ;
2	serverside_create()	serveur (callback)	execution(* javax.ejb.*Bean+.ejbCreate(..)) ;
2	clientside_setContext()	client	ϕ (le contexte n'est pas crée ni exposé au niveau du client)
2	serverside_setContext()	server	call(void javax.ejb.*Bean+.set*Context (..)) ;
3	clientside_componentCall()	client	(call(* javax.ejb.EJBObject+.* (..)) || call(* javax.ejb.EJBHome+.* (..))) && ! (withinEJBMiddleware()) ;
3	serverside_componentCall()	serveur	execution(public * javax.ejb.*Bean+.* (..)) && ! execution(public * javax.ejb.*Bean+.ejb* (..)) ;
4	clientside_remove()	client	call(* javax.ejb.EJBHome+.remove (..)) ;
4	serverside_remove()	serveur (callback)	execution(* javax.ejb.*Bean+.ejbRemove(..)) ;
4	serverside_unsetContext()	serveur	call(void javax.ejb.*Bean+.unset*Context(..)) ;

TAB. 3.1 – Expressions de coupes abstraites pour une plate-forme EJB.

Étape	Nom de la coupe	Rôle	Expression AspectJ de la coupe
1	clientside_middlewareBinding()	client	call(org.omg.CORBA.ORB+ org.objectweb.ccm.CORBA.TheORB.getORB (..)) || call(org.omg.CORBA.ORB+ org.omg.CORBA.ORB.init (..)) ;
1	serverside_middlewareBinding()	serveur	ϕ (le serveur est automatiquement lié au serveur au chargement)
1	clientside_serviceBinding()	client	call(org.omg.CORBA.Object+ org.omg.CORBA.ORB+.resolve_initial_reference (String)) ;
1	serverside_serviceBinding()	serveur	ϕ la partie serveur du service ne nous concerne pas
2	clientside_lookup()	client	call(org.omg.CORBA.Object+ org.omg.CosNaming.NamingContext.resolve (..)) ;
2	serverside_lookup()	serveur	ϕ (un serveur ne cherche pas, il est recherché)
2	clientside_narrowing()	client	call(org.omg.CORBA.Object+ ..*Helper.narrow (org.omg.CORBA.Object+)) || call(org.omg.CORBA.Object+ ..*Helper.unchecked_narrow (org.omg.CORBA. Object+)) ;
2	serverside_narrowing()	serveur	ϕ (un serveur ne fait pas de narrow)
2	clientside_create()	client	!withinCCMMiddleware() && call(org.omg.CORBA.Object+ org.omg.CCMObject+.create(..)) ;
2	serverside_create()	serveur (callback)	execution(void ccm_create(..))
2	clientside_setContext()	client	ϕ (le contexte n'est pas créé ni exposé au niveau du client)
2	serverside_setContext()	serveur (callback)	execution(void set_ccm_context (org.omg.Components.CCMContext)) ;
2	clientside_find()	client	call(org.omg.CORBA.Object+ find* (..))
2	serverside_find()	serveur	Un serveur ne cherche pas
3	clientside_componentCall()	client	call(* org.omg.Components.CCMObjectOperations+.* (..)) && ! call(* org.omg.Components.CCMObjectOperations+.ccm_* (..)) && ! call(* org.omg.Components.CCMObjectOperations+.provide_* (..)) ;
3	serverside_componentCall()	serveur	execution(org.omg.Components.*Component+.* (..))) ;
4	clientside_remove()	client	call(void org.omg.CCMObject+.remove (..)) ;
4	serverside_remove()	serveur (callback)	execution (void ccm_remove ()) ;
4	serverside_unsetContext()	serveur	call (void unset_ccm_context()) ;

TAB. 3.2 – Expressions de coupes abstraites pour une plate-forme CCM.

débuté en 1999. Le premier article international référé est paru en 2000 [130], suivi par d'autres (notamment [133] et [134]).

JAC offre un support pour la distribution par le biais d'une libraire d'aspects pour la programmation distribuée et via la notion de coupe distribuée. Cette notion est un résultat original qui a depuis été repris dans d'autres approches (voir [118]).

Dans le domaine des intergiciels, JAC a également apporté une contribution dans le domaine des architectures ouvertes. En effet, JAC peut être vu comme un conteneur ouvert dans lequel les services techniques sont chargés à la demande, en fonction des besoins. Par rapport à d'autres approches sur les conteneurs ouverts comme [177], l'originalité réside dans la programmation des services sous forme d'aspects.

En terme de conception, JAC est associé à un profil UML qui fournit des extensions pour l'expression des aspects et des coupes au sein de diagrammes UML. Cette extension est supportée par un environnement de développement intégré qui est lui-même une application orientée aspect construite avec JAC.

Enfin, soulignons que, en tant que logiciel *open source* du consortium ObjectWeb, JAC a atteint un degré de maturité qui a permis à la société de services AOPSYS de développer des applications pour des clients industriels.

La section 3.4 part du constat que les applications EJB et CCM partagent de nombreux concepts et que leur programmation suit le même schéma. Nous avons donc proposée une couche abstraite à base d'aspects qui matérialise ces concepts et qui permet de développer de nouveaux services pour composants. Ces services peuvent alors être utilisés indifféremment avec l'une des deux technologies en sélectionnant la personnalité adéquate (EJB ou CCM) de la couche abstraite.

Chapitre 4

Composants et ADL

Sommaire

4.1 Développement à base de composants

Le concept de composant logiciel est presque aussi vieux que celui d'objet. Sa première utilisation [111] remonte certainement à 1968. Il a fallu néanmoins attendre presque 30 ans pour que cette notion se popularise sous l'impulsion notamment des modèles industriels EJB (*Enterprise Java Bean*) de Sun et COM (sous ses différentes versions successives, COM, COM+, DCOM, .Net) de Microsoft. Le principe du composant logiciel se veut similaire à celui du composant électronique : il s'agit de développer une brique logicielle qui puisse être connectée à d'autres briques afin de construire les logiciels par assemblage de briques élémentaires. Cette ligne directrice a pour but de rationaliser, standardiser et réduire les coûts de l'ingénierie logicielle en maximisant autant que faire ce peut la réutilisation des composants.

Cette possibilité de créer des assemblages de composant est un caractéristique majeure qui différencie ce style de programmation, des objets, des procédures ou de la programmation fonctionnelle. Une seconde caractéristique majeure des approches à base de composants est la présence d'une

structure d'accueil, souvent appelée conteneur, pour l'hébergement des composants. Cette structure fournit un ensemble de services permettant l'exécution du composant. Ces services sont en général d'ordre technique, par exemple nommage, transaction, sécurité et persistance, et offrent un niveau de contrôle et/ou d'interprétation sur le composant. Le principal problème auquel doivent faire face les concepteurs des modèles de composant est celui de la granularité de ce conteneur. Il s'agit souvent de trouver un compromis entre le nombre de services, le degré d'ouverture du conteneur et la facilité de développement. Par ailleurs, les services offerts par un conteneur ne sont pas souvent universels : les services à offrir aux systèmes d'information ouvert sur Internet ne sont pas les mêmes que ceux pour des composants destinés à être embarqués dans des environnements contraints. Il y a donc une réflexion à mener autour du métier auquel est destiné le modèle de composants. Par ailleurs, les besoins en service peuvent aussi varier au cours du temps ce qui nécessite une adaptation du conteneur. On rejoint là une problématique d'adaptabilité telle que l'on a pu l'étudier dans le cadre des aspects au chapitre précédent. Au final, il l'apparaît que l'ingénierie du conteneur, qui est en donc une **ingénierie du contrôle** du composant, est un problème de recherche actuel. Ce chapitre propose un certain nombre de solutions dans ce domaine. Ces solutions sont largement inspirées de notre expérience sur les aspects.

4.1.1 Définitions

Le terme composant logiciel recouvre de nombreuses réalités selon le domaine applicatif auquel on se réfère. Du temps réel aux systèmes d'informations, un composant peut être aussi bien un fichier C externalisant des propriétés, qu'une entité logicielle proposant des concepts (ports, interfaces, liaisons, connecteurs, etc.) dont la structure, parfois la sémantique, est définie de façon précise dans un méta-modèle.

Alors que le concept de classe, même s'il peut varier légèrement d'un langage à l'autre, est maintenant bien établi, il n'en est pas de même avec celui de composant. Une classe est définie en fonction de ses éléments internes, données et traitements. Les composants sont définis quant à eux, en fonction de propriétés externes. Ils sont vus la plupart du temps comme des boîtes noires. Le but est de permettre une réutilisation optimale du composant et sa composabilité avec d'autres composants. Pour cela, il est admis que les détails d'implémentation constituent des informations de trop bas niveau limitant la réutilisabilité et la composabilité. Il s'ensuit que les définitions proposées s'appuient sur une vision de haut niveau du composant. Il est alors paradoxal de constater qu'un parti pris consistant à faire abstraction de détails d'implémentation ne permet pas de converger mais au contraire, donne lieu à de nombreuses interprétations.

Deux définitions semblent néanmoins émerger. La première est due à Szyperski [168] et date de 1997. Elle établit qu'un composant est :

Définition 4.1 *A component is a unit of composition with contractually specified interfaces and context dependencies only. A software component can be deployed independently and is subject to composition by third parties.*

La première partie de la définition établit clairement la présence de deux contrats correspondant aux services requis et fournis par le composant. Selon les approches, ces contrats se matérialisent par des interfaces, des ports ou des connecteurs. La seconde partie de la définition est néanmoins plus sujette à interprétation puisqu'elle établit qu'un composant peut être déployé et composé, sans définir précisément les éléments qui permettent cela.

En 2001, Heineman et Councill [69] ont proposé la définition suivante :

Définition 4.2 *A software component is a software element that conforms to a component model that can be independently deployed and composed without modification according to a composition standard.*

A component model defines specific interaction and composition standards. A component model implementation is the dedicated set of executable software elements required to support the execution of components that conform to the model.

A software component infrastructure is a set of interacting software components designed to ensure that a software system or subsystem constructed using those components and interfaces will satisfy clearly defined performance specifications.

La première partie de la définition met en lumière le fait qu'il peut y avoir différents modèles de composants. C'est peut-être là un point clé de l'approche à base de composant : il ne peut pas y avoir une solution unique qui répond à tous les besoins, de ceux de l'embarqué fortement contraint à ceux des systèmes d'information. La deuxième partie de la définition insiste sur les modèles d'interaction et de composition entre composants. On touche ici à un point clé des composants qui est absent de l'approche objet, mais que l'on retrouve dans d'autres domaines comme celui des langages de description d'architecture (ADL). Finalement, la troisième partie de la définition semble plus sujette à discussion puisqu'elle met l'accent sur des spécifications de performance qui ne sont pas forcément l'objectif de tous les modèles de composants (par exemple EJB).

Bien que l'on pourrait continuer cette revue des définitions proposées pour le terme composant ([168] en recense neuf, la première étant due à Grady Booch en 1987 [23], non compris les deux précédentes), un certain nombre de caractéristiques se dégagent : un composant doit se conformer à un modèle, il spécifie des services offerts et requis, il interagit avec d'autres composants selon des règles d'interactions précises faisant partie du modèle, il est conçu pour être déployé et composé.

4.1.2 Composants et langages de description d'architecture

De même que la notion de composant, la notion d'architecture logicielle a donné lieu à plusieurs définitions. La définition qui, à mon sens, résume le mieux l'état des lieux actuel est due à [13] :

Définition 4.3 *A software architecture of a program or computing system is the structure or structures of the system, which comprise software components, the externally visible properties of those components, and the relationships among them.*

Lorsqu'il s'agit de décrire une architecture logicielle, on peut distinguer deux grandes catégories d'approches : les modèles et les langages. Avec les modèles, il s'agit d'organiser, par exemple en couches, l'ensemble des concepts qui décrivent l'architecture à laquelle on s'intéresse. L'architecture OMA (*Object Management Architecture*) pour les environnements CORBA ou le modèle RM-ODP [75] sont des exemples de modèles d'architectures. Les langages de description d'architecture [112] (ADL pour *Architecture Description Language*), quant à eux, mettent en place un ensemble d'artefacts (composant, connecteur, liaison, port, interfaces, etc.) et une syntaxe concrète qui permettent aux développeurs de décrire la structure de leur application. AADL [9], Acme [61], ArchJava [5], Darwin [104], Rapide [102], UniCon [182] et Wright [6] sont des exemples d'ADL.

En un certain sens, le concept d'ADL peut être vu comme une extension du concept d'IDL (*Interface Definition Language*). Alors qu'un IDL se contente de décrire les services offerts par

une entité (parfois aussi les services requis comme c'est le cas de l'IDL3 du modèle de composants CORBA CCM), un ADL va au-delà de cette première approche et prend en compte l'organisation globale des entités d'une application. Un ADL va ainsi décrire des schémas de connexion (la topologie de l'application) et/ou des schémas d'imbrication (lorsque les entités manipulées supportent des relations de type "est-contenu-dans") et/ou de décrire finement la sémantique des interactions. Il s'agit donc, comme pour l'architecture du monde concret, de donner le plan des applications.

Bien que les domaines des ADL et des composants logiciels puissent être étudiés indépendamment, il apparaît que dans de nombreux travaux actuels, les entités logicielles manipulées par les ADL sont des composants et que les modèles de composants proposent pratiquement tous un ADL. Les ADL permettent donc d'exprimer la composition des composants.

4.1.3 Composants et intergiciels

Lorsqu'ils sont employés conjointement les intergiciels et les composants donnent lieu à deux catégories de travaux : les intergiciels pour les applications à base de composants et les modèles de composants pour la construction d'intergiciels.

Intergiciels pour les applications à base de composants

La catégorie des intergiciels pour les applications à base de composants est la plus courante dans le monde industriel. Chronologiquement elle s'est développée dans la seconde moitié des années 1990, avant les modèles de composants pour la construction d'intergiciels qui ont été étudiés principalement dans la première moitié des années 2000. Les trois représentants principaux de la catégorie sont J2EE [20], CORBA CCM [162] (notamment le chapitre 5) et .Net/COM+ [87]. Nous ne fournirons pas de description détaillée de ces environnements, et nous contenterons d'en relever quelques faits saillants. Nous renvoyons le lecteur aux références citées pour de plus amples informations.

Ces intergiciels fournissent un support pour des applications à base de composants mais sont eux-mêmes construits à l'aide de techniques traditionnelles, le plus souvent objet.

Les trois approches citées mettent en avant une architecture à base de conteneurs et ciblent principalement le domaine du client/serveur 3-tiers pour les systèmes d'information (bien que CORBA CCM soit également utilisé dans le domaine de l'informatique industrielle). Dans les trois cas, le conteneur héberge les composants, leur fournit des services systèmes (nommage, communication, persistance, sécurité, transaction) et gère leur cycle de vie. Les trois approches offrent aussi des mécanismes de *packaging* et de déploiement des composants. J2EE distingue les composants de présentation (JSP, servlet), des composants métier (EJB). CORBA CCM implémente des notions d'interfaces requises et fournies et supporte les communications de type requête/réponse et événementielle. .Net s'appuie sur le modèle COM+ pour offrir des services systèmes identiques à ceux des composants EJB. En général, ces intergiciels ne sont pas extensibles de façon standard : les services proposés aux composants sont fixes et prédéfinis. Une direction de recherche consiste à proposer des mécanismes pour rendre ces intergiciels plus flexibles : on peut notamment citer les travaux sur les conteneurs ouverts de CCM [177], sur l'extension de services dans les conteneurs EJB [12] ou sur la CVM *Container Virtual Machine* [65].

Modèles de composants pour la construction d'intergiciels

La catégorie des modèles de composants pour la construction d'intergiciels est celle qui, à l'heure actuelle, est l'objet du plus grand intérêt dans la communauté scientifique. Il s'agit de concevoir des modèles de composants adaptés au développement d'intergiciels. Les composants se trouvent cette fois-ci dans la couche intergiciel. Il s'agit de déterminer quelles sont les abstractions les plus aptes à remplacer ou à compléter les techniques objet afin que le développement d'intergiciels soit plus fiable, flexible, rapide, aisé, réutilisable. Les modèles OSGi [122], Fractal [27], OpenCOM [39] et dans une certaine mesure JMX font partie de cette catégorie.

Finalement, notons que des expériences récentes prouvent que les composants peuvent être utilisés à la fois dans la couche intergiciel et la couche applicative : "JOnAS à la carte" [1] utilise Fractal au niveau intergiciel et permet de développer des applications à base de composants EJB ; [45] est une initiative similaire dans laquelle l'intergiciel est construit à l'aide de composants OSGi.

4.1.4 Modèles de composants

Cette section présente un aperçu de quelques modèles de composants existants. La section 4.3.1 complètera cet aperçu avec des modèles de composants qui mettent en œuvre un rapprochement entre les notions de composant et d'aspect.

OSGi [122] est un modèle de composants Java pour le déploiement de services en réseaux. Initialement le domaine applicatif cible était les plates-formes résidentielles pour la domotique. Rapidement, le modèle s'est avéré apte à être utilisé non seulement pour l'embarqué (transports, téléphonie, etc.) mais aussi pour des applications destinées à des machines de bureau (voir l'IDE Eclipse). OSGi définit un environnement d'exécution minimal comprenant un service de chargement des composants, un service de gestion du cycle de vie des composants (installation, démarrage, arrêt, mise à jour, retrait). OSGi offre un mécanisme de *packaging* (les *bundles*) et de déploiement des composants. Ceux-ci fournissent des services et peuvent exprimer des dépendances vers des services requis. L'assemblage de composants OSGi est dynamique. On peut noter que, même si le modèle OSGi est moins riche que par exemple le modèle Fractal, sa légèreté, sa standardisation (via le consortium OSGi Alliance) et son adoption par de nombreux acteurs industriels lui confèrent un certain succès.

Fractal [27] est un modèle de composants ouvert et extensible conçu par France Telecom R&D et l'INRIA. L'apport principal de Fractal réside dans la distinction claire qui est faite entre les fonctions métier des composants (services fournis et requis) et les caractéristiques non fonctionnelles (notions de contrôleurs et d'interfaces de contrôles). Il est ainsi possible de personnaliser le fonctionnement d'un composant en modifiant sa partie contrôle.

Le modèle est dynamique, hiérarchique (un composant peut contenir d'autres composants et est alors appelé composite), autorise le partage de composants entre différents composites et incorpore la notion de *template* de composants. Le modèle est associé à un langage de définition d'architecture (ADL) qui permet de définir aisément des assemblages complexes de composants.

Différentes implémentations du modèle existent, en C (Think [54]), en Java (Julia [27], ProActive [15], AOKell [159]) et en Smalltalk (FracTalk). Plusieurs librairies de composants ont été

développées pour la construction de systèmes. Par exemple, Kortex [54] est une librairie de composants pour la construction de systèmes d'exploitation pour les architectures RISC PowerPC. Ces composants assurent une gestion des pilotes de périphérique, de la mémoire, des threads, de l'ordonnancement, des processus, des chargeurs d'applications, des réseaux et des liaisons. Différents systèmes ont été implémentés avec Kortex dont un micro-noyau de type L4, un environnement d'exécution pour le langage de programmation de routeurs PlanP, une machine virtuelle Java Kaffe et un système dédié pour l'exécution du jeu vidéo Doom. Ces différentes expériences ont montré que l'approche à composants est viable pour la construction de systèmes et qu'elle n'entraîne pas de surcoût majeur au niveau de l'exécution. De même, Dream [90, 91] est une librairie de composants pour la construction d'intergiciels. Plusieurs intergiciels ont été construits, dont AAA, un intergiciel orienté message permettant le déploiement d'agents, LeWYS, un framework pour la construction de systèmes de monitoring distribués, ou Tribe, un framework pour la construction d'intergiciels basés sur les communications de groupe.

Kilim [71] est un modèle de composants développé par la société Kelua dans le cadre du consortium *open source* ObjectWeb. Dans Kilim, une séparation claire est faite entre la vue composant et la vue objet avec des règles de projection explicites entre elles. Trois points importants caractérisent la vue composant : le modèle de composant qui est similaire à celui de Fractal, et deux niveaux de description. Le premier, statique et textuel, basé sur XML correspond à une description de l'assemblage de composants appelée *template*. Le second concerne la description du système au niveau exécutif.

Les trois concepts de base de Kilim sont ceux de *port*, de *provider* et de *transformer*. Un *port* est une variable utilisée pour récupérer la référence de l'interface d'un objet. Un *provider* est une abstraction d'un mécanisme de niveau langage permettant d'obtenir des références d'objets (par exemple, constructeur ou méthode de type *getter*). De façon symétrique, un *transformer* permet de modifier l'état d'un objet (par exemple, méthode de type *setter*).

Finalement, la notion de *template* permet, à l'aide d'un ADL XML, de fournir la méta-information nécessaire à la description, l'instanciation, la composition et la configuration des composants.

OpenCOM [39] est un modèle de composants conçu à l'Université de Lancaster (UK). Il étend le modèle COM de Microsoft avec des notions de reconfiguration, d'interception et de gestion des dépendances. OpenCOM partage de nombreux points communs avec Fractal : les deux modèles sont issus de l'expérience de leurs auteurs avec le modèle RM-ODP [75] ; les deux modèles sont proches dans la façon dont ils gèrent les interfaces, les connexions et le cycle de vie des composants ; ils ont tous les deux la volonté de fournir un modèle léger et performant en minimisant autant que faire se peut le surcoût par rapport à un modèle objet. Fractal a néanmoins poussé la réflexion sur les composants plus loin (avec notamment la notion d'interface de contrôle et l'intégration d'un langage de description d'architectures). Fractal dispose également d'une base de code plus importante que OpenCOM. Le modèle OpenCOM a été utilisé pour construire l'intergiciel OpenORB [18, 42] que nous avons déjà cité dans le cadre de la réflexivité.

K-Component [46] est un modèle de composants conçu au Trinity College de Dublin. Les composants fournissent des services et en requièrent d'autres pour fonctionner. Les connecteurs sont des entités de première classe dans ce modèle. Ils sont générés à partir d'une description

IDL3 CORBA des services requis et fournis par le composant. Un méta-modèle permet de réifier l'architecture d'une application programmée en K-Component. La reconfiguration et l'adaptation d'une architecture se font en écrivant des programmes appelés contrats d'adaptation : ils parcourent le graphe représentant l'architecture et le modifient en fonction des besoins. Cette fonctionnalité permet de séparer clairement la logique d'adaptation du reste de l'application.

Comet [136] est un modèle de composants conçu à l'Université Paris 6 par F. Peschanski. Un composant Comet est associé à un descripteur d'événements entrants et à un descripteur d'événements sortants qui représentent donc les ports de communication de ce composant. Les événements sont typés. Comet est un modèle réflexif et qui fournit un support pour l'adaptation. Le niveau dit méta-comportement fournit les propriétés réflexives de Comet. À la manière de CodA [109], ce niveau réifie les événements liés à la gestion des messages : réception, envoi, mise en file d'attente, sortie de file d'attente, exécution. Le comportement par défaut peut alors être modifié pour personnaliser le fonctionnement des composants. Un second niveau, dit méta-rôle, fournit les capacités d'adaptation. Un rôle permet d'ajouter dynamiquement à un composant Comet un comportement non prévu initialement. Il peut être employé pour par exemple, dans le cadre d'un protocole de communication, assigner à un composant un rôle particulier, ou ajouter de nouvelles fonctionnalités telles que la réplication de l'état du composant.

Le foisonnement d'activités autour des composants fait que plusieurs autres modèles pourraient être présentés comme par exemple Accord/UML [174], SOFA [141] et Arctic Beans [7]. Finalement, ce panorama ne serait complet sans citer les modèles de composants plus industriels qui sont largement présentés et commentés dans la littérature : Sun EJB [20]), CORBA CCM [162][114], Microsoft COM+/.Net [87], IBM SanFrancisco [21].

4.2 Une expérience comparative aspect-composant

Cette section fait le point sur une étude comparative sur la programmation orientée aspect et la programmation à base de composants. Un service de réplication et de partage de documents dont la cohérence est gérée avec l'algorithme de K. Li et P. Hudak, a été développée selon deux approches composants (Fractal [27] et Kilim [71]) et une approche AOP (JAC voir chapitre 3 section 3.3). Les trois conceptions et réalisations ont alors été comparées.

Cette étude a été conduite en 2004 dans le cadre du stage de DEA de F. Loiret [97] (co-encadrement avec E. Gressier-Soudan). Elle a donné lieu à la publication [98].

4.2.1 Choix de l'algorithme

L'algorithme de cohérence est celui défini par K. Li & P. Hudak [96]. Son principe est simple : il fonctionne suivant un protocole à invalidation sur écriture. Le dernier écrivain est le propriétaire de la copie de référence, il délivre des copies à chaque lecteur. Dans le système, il n'y a qu'une seule copie en écriture, celle du propriétaire, ou n copies en lecture. L'invalidation consiste à envoyer un message à tous les lecteurs en cours quand un nouvel écrivain récupère la copie de référence du précédent propriétaire. Le prédicat "1 écrivain ou exclusivement n lecteurs" est assuré par un jeu de verrous d'exclusion mutuelle dans les tables d'accès aux données entre tous les acteurs effectuant un accès simultané à la même page. Il existe plusieurs variantes de l'algorithme. La

version avec gestionnaire centralisé repose sur un annuaire des propriétaires qui permet de localiser directement le détenteur de la copie de référence d'une donnée. Celui-ci joue le rôle de séquenceur et ordonne totalement toutes les demandes d'accès. D'autres versions plus réparties jouent, soit sur la localisation du propriétaire en utilisant une heuristique de localisation par propriétaire probable, soit sur une gestion répartie de la liste des lecteurs (*copyset* distribué) en cours qui implique une gestion distribuée des invalidations. Un ordre total sur les accès est toujours mis en œuvre à travers la localisation et la gestion de verrous sur les différents sites impliqués.

Après un premier travail sur la version centralisée optimisée, les résultats présentés ci-dessous sont relatifs à la version répartie, souvent baptisée "avec propriétaire probable et *copyset* distribué".

4.2.2 Les fonctionnalités séparées

À partir des spécifications de l'algorithme, nous avons réalisé une analyse dont le but est de faire apparaître les différents blocs fonctionnels intervenant dans le service de données réparties partagées cohérentes. Comme toute analyse, elle est subjective : d'autres solutions pourraient être proposées. Néanmoins, cette analyse ne constitue pas l'objectif de ce travail qui reste la comparaison de Fractal, Kilim et JAC. Partant de là, la même analyse a été mise en œuvre dans chacune des trois approches.

À partir des spécifications de l'algorithme, nous avons identifié et isolé sept fonctionnalités. Cette analyse résulte de plusieurs itérations sur l'implémentation du service. Notamment, la mise en œuvre de la version centralisée a fait apparaître le découpage actuel qui a été consolidé lors du passage à la version distribuée.

Nous avons considéré les deux opérations de lecture et d'écriture (1) correspondent à la logique fonctionnelle (`métier`) de notre application. Le choix nous semble justifié par le fait que l'objectif premier du service reste d'accéder en lecture et en écriture à des données partagées et réparties.

Les autres fonctionnalités sont alors considérées comme relevant du plan non-fonctionnel : au sein de l'algorithme, la cohérence des données est assurée par la mise en œuvre du mécanisme d'`invalidation` (2) et par la gestion du `verrouillage` (3) des accès aux données. Le service de `localisation` (4) de la copie de référence est nécessaire lors des défauts. Rappelons qu'à chaque défaut, il y a toujours une localisation de copie, mais lors d'une lecture, on cherche la première copie disponible. Certains traitements algorithmiques nécessaires à la `mise en œuvre de l'algorithme` (5) ont été isolés. Ces traitements, indépendants des fonctionnalités d'invalidation, de verrouillage et de localisation, concernent par exemple, la gestion de l'ordre total. La gestion de la répartition de notre application a été modularisée au sein d'une fonctionnalité de `communications distantes` (6), dont les traitements sont délégués à la couche Java RMI. Enfin, nous avons ajouté une fonctionnalité de `journalisation` (7). Cette fonction n'existe pas dans la spécification d'origine mais elle est intéressante pour superviser l'exécution du service. De plus, c'est une fonctionnalité que l'on retrouve dans quasiment tous les services système.

La vision des fonctionnalités séparées est schématisée sur la figure 4.1. Il est important de noter que ces fonctionnalités ne sont pas orthogonales les unes par rapport aux autres. Elles répondent à des besoins différents, mais sont en interaction les unes avec les autres pour aboutir au service final. Sur la figure 4.1, deux fonctionnalités partageant une même frontière sont en interaction l'une avec l'autre. Les flèches illustrent le sens des dépendances entre ces fonctionnalités.

Après cette étape d'identification des préoccupations, il est possible de modéliser de manière générique l'application sous forme de diagramme de classes UML (voir [97] pour plus de détails). La

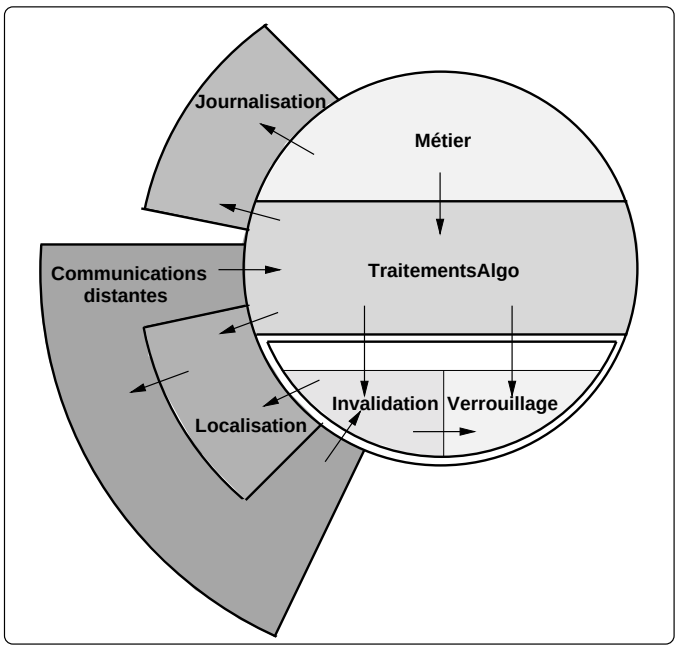

FIG. 4.1 – Schématisation des fonctionnalités séparées.

transposition consiste à associer à chaque fonctionnalité une classe. Les dépendances entre classes sont modélisées par des liens d'agrégation. Chaque classe implante un ensemble de méthodes lié à la sémantique de la fonctionnalité qui lui est associée. De manière à capturer l'ensemble de la sémantique de l'application, on modélise alors les diagrammes d'interaction liés au déclenchement de chaque opération source (par exemple une lecture).

4.2.3 Comparaison des trois mises en œuvre

Trois conceptions et réalisations de cet algorithme ont été mises en œuvre, avec Fractal, Kilim et JAC. À titre d'illustration, la figure 4.2 présente l'architecture Fractal de l'algorithme. Plus d'informations sur cette architecture et sur les versions Kilim et JAC sont disponibles dans [97].

Dans cette section, nous comparons les différentes approches selon des critères qui nous ont paru pertinents.

4.2.3.1 En terme de modèle de programmation

Les critères retenus en terme de modèle de programmation sont issus de [106].

1. Les points d'entrée du composant au sein du modèle Fractal sont définis par les interfaces de contrôle (formulées par les spécifications) et les interfaces fonctionnelles (construites par le

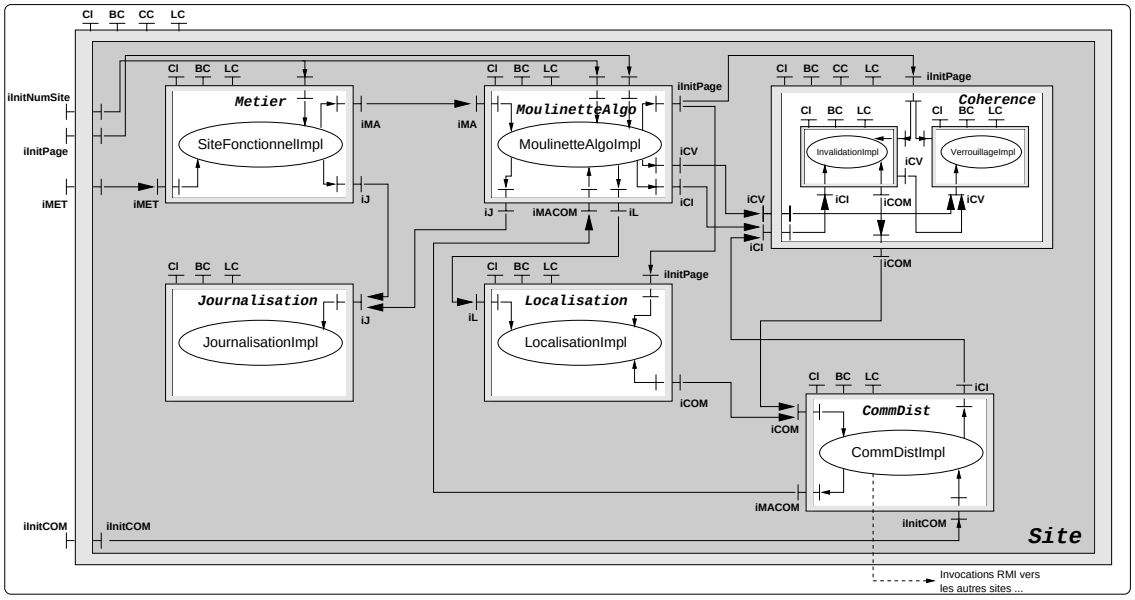

FIG. 4.2 – Architecture de l'algorithme en Fractal.

développeur). Dans Kilim, les points d'entrée sont les *ports*. Au sein de JAC, nous pouvons considérer que les points de jonction constituent les points d'entrée entre les objets de base et les composants d'aspects. La définition de l'encapsulation pour les trois modèles peut être déduite de cette définition de points d'entrée.

2. La notion d'identité (la capacité à définir de manière non ambiguë une entité logicielle) d'un composant dans Fractal se fait par l'intermédiaire d'un type de composant au niveau du modèle et d'une interface `ComponentIdentity` au niveau de l'exécution qui permet alors de récupérer toutes les références d'interfaces externes du composant. Dans le modèle proposé par Kilim, on trouve respectivement la notion d'instance de *template* et la classe `RtComponent` à l'exécution. Dans JAC, l'identité d'une entité logicielle n'est marquée que par la distinction entre un objet de base et un composant d'aspect.

3. La composabilité au sein de Fractal est structurelle, elle est assurée par des liaisons entre interfaces clientes et serveurs, la notion de composant composite assure la récursivité. Une vérification sémantique de l'assemblage est rendue possible au moment de l'analyse de l'ADL puisque le typage des interfaces y est défini. Au sein de Kilim, l'assemblage est également structurel, il s'effectue par l'intermédiaire des opérateurs `plug` et `bind`, la composition entre *templates* assure la récursivité. Les composants Kilim ne sont pas typés, la sémantique de l'assemblage est donc à la charge du développeur. Dans JAC, l'assemblage entre objets de base et composants d'aspect est effectué par le processus de tissage, le long des coupes transversales définissant le déploiement des aspects. Le modèle est récursif dans le sens où l'on peut tisser un nombre illimité d'aspects autour d'une même méthode de base. La sémantique de la composition est comportementale puisque celle-ci se manifeste à l'exécution (par exemple, lors d'un appel de méthode). La vérification de la sémantique d'assemblage est à la charge du développeur qui doit s'assurer du bon ordre d'exécution des *wrappers*.

4.2.3.2 En terme d'impact sur la séparation des préoccupations

La vision de la séparation des préoccupations détermine la manière d'implanter notre application, qu'elle soit développée avec un modèle de composants techniques ou avec un modèle de programmation orientée aspects. C'est de cette vision qu'est déterminée une modélisation sous forme de diagrammes UML, que sont explicités les dépendances entre fonctionnalités et le fil d'exécution correspondant à la sémantique de l'application.

4.2.3.3 En terme de dispersion de code

Dans le cadre des modèles Fractal et Kilim, la construction de l'application s'effectue par assemblage de composants. Des liaisons entre composants peuvent être créées et détruites. Néanmoins, l'utilisation d'un composant et donc d'une fonctionnalité, se fait via des interfaces et des appels de méthode. Le code des composants est bien sûr modulaire, mais son utilisation reste dispersée et donc transverse aux différents autres composants.

Au sein de la plate-forme JAC, l'assemblage des entités se configure de manière indépendante du code de l'application et s'effectue alors dynamiquement au moment du processus de tissage. Il n'y a donc aucune référence codée en dur entre objets de base et composants d'aspects.

4.2.3.4 En terme de vision de l'application et de son évolutivité

La vision d'une application assemblée avec Fractal est claire puisque le développeur est contraint par des règles d'écriture définies dans les spécifications (définitions d'interfaces et leurs implémentations, implémentations des mécanismes de liaisons). Les liaisons de type client/serveur permettent de montrer le sens de la dépendance entre les composants. Avec Kilim, la vision de l'application est plus ambiguë car le modèle n'impose aucune règle d'écriture du code, l'opérateur `plug` n'est pas asymétrique, le "sens" de la liaison ne reflète pas obligatoirement le sens de la dépendance entre deux composants et au niveau de l'ADL, les mécanismes de liaisons sont manipulés au même titre que les mécanismes d'instanciation. Cependant, l'utilisation de Kilim n'impose aucune contrainte au niveau du code de l'application, contrairement à Fractal. Avec JAC, la vision de l'application peut s'avérer complexe, surtout dans le cadre de notre prototype où de nombreux aspects non-fonctionnels sont tissés et demeurent inter-dépendants.

De manière à tester les capacités d'évolution des plates-formes, nous avons étudié la mise en œuvre d'une autre alternative concernant la gestion d'une cohérence causale au sein de l'algorithme initialement proposé. Cette modification a un impact sur les structures de données utilisées, sur des fractions de code concernant différentes fonctionnalités et bien évidemment sur le fil d'exécution lié à la sémantique de l'application. Nous nous sommes alors rendus compte que de nombreux changements d'implémentation étaient alors nécessaires concernant les plates-formes de composants techniques, essentiellement à cause de modifications de signatures de certaines méthodes des interfaces des composants permettant l'assemblage de l'application. Dans le cas de JAC, la transposition est plus souple : pour cet exemple précisément, les dépendances entre aspects non-fonctionnels demeurent inchangées, les modifications de code sont localisées et influencent de manière moins marquée le reste de l'application.

4.2.3.5 En terme de lignes de code

Le nombre de lignes de code nécessaires à l'implémentation de l'algorithme est représenté figure 4.3. À titre indicatif, les versions Fractal, Kilim et JAC comportent respectivement 1095, 1334 et 1220 lignes.

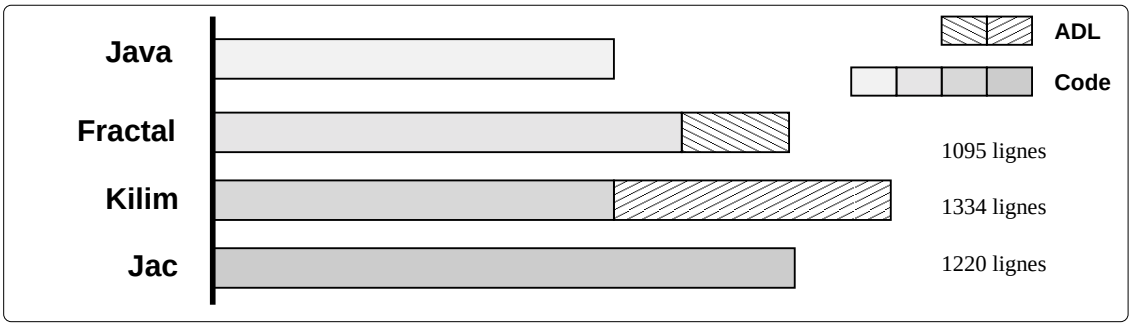

FIG. 4.3 – Quantité de lignes de code en fonction de la plate-forme d'accueil.

Nous avons implanté un prototype en Java "pur", d'abord de manière monolithique (sans SoC[1]) puis en prenant en compte notre vision des fonctionnalités séparées (avec SoC). Dans le deuxième cas, le code engendré est bien sûr plus élevé puisque les classes sont fragmentées. Nous avons donc besoin de code supplémentaire pour les "reconstituer". Il en est de même pour les trois plates-formes étudiées : en utilisant les interfaces Fractal, les méthodes *setter/getter* avec Kilim et les composants d'aspect avec JAC.

Le code de la version Java (avec SoC) est strictement identique à celui utilisé avec Kilim : dans le premier cas, on utilise une méthode `main` pour assembler les instances (par les constructeurs et les méthodes *setter*), alors que dans le cas de Kilim, l'assemblage est décrit par l'ADL.

Le code engendré par le prototype Fractal et JAC est plus élevé que celui de Kilim : en effet, au sein de ces deux premières plates-formes, le modèle de composants intervient au niveau du langage (gestion du mécanisme de liaison entre interfaces avec Fractal, gestion des *wrappers* avec JAC). La description de l'application avec l'ADL de Kilim est plus coûteuse que celle de Fractal : on manipule beaucoup de mécanismes avec Kilim pour obtenir une méta-description complète de l'application, avec Fractal, on ne décrit que l'assemblage des composants alors que les mécanismes mis en œuvre interviennent au niveau des sources.

4.2.3.6 En terme de performances

Nous avons cherché à comparer les performances des trois implémentations. Bien évidemment, ces mesures n'ont pas été faites de façon brute, car plusieurs paramètres, comme les invocations distantes, perturbent l'application. En fait, nous souhaitons mesurer le surcoût induit par l'utilisation de Fractal, Kilim ou JAC par rapport à une application en Java "pur". Ces mesures ont donc été réalisées en dehors du service de partage de documents.

Nous avons mis en place un programme de *benchmark* qui permet de mesurer le coûts des invocations de méthodes entre objets. Plusieurs méthodes avec des signatures différentes sont invoquées. Le corps des méthodes est trivial mais non vide pour éviter les effets d'optimisation

[1] séparation des préoccupations (en anglais *separation of concerns*).

du compilateur JIT. Par ailleurs, le test est effectué plusieurs fois avant la prise de mesure pour éviter les phases transitoires dues aux chargements initiaux dans la machine virtuelle. Dans le cas de Kilim et Fractal, ce test consiste donc en des séries d'appels entre un composant client et un composant serveur. Dans ce cas de JAC, le serveur est associé à un aspect afin de mesurer donc le coût de l'exécution d'un aspect. Finalement, le test est effectué en java "pur" afin de fournir une base de comparaison.

Pour une série de un million d'appel sur l'objet serveur, la version Java "pur" fournit un temps de référence de 157 ms. La version Kilim fournit un temps identique. La version Fractal (sans optimisation) s'exécute en 476 ms (facteur 3). La version JAC s'exécute en 1450ms (facteur 9). Ces mesures s'expliquent par la façon dont les frameworks gèrent l'**interception** :

- Kilim ne fait pas d'interception. C'est uniquement un framework de configuration qui crée une architecture logicielle. Il est donc normal qu'une fois les liens mis en place, il n'y a pas de surcoût par rapport à la version Java.
- Fractal installe une couche de contrôle autour des objets Java. Cette couche intercepte les appels pour notamment bloquer les appels lorsque, par exemple, le composant est arrêté. Cette interception a donc un coût qui se reflète dans les performances. Notons que si l'on désactive la fonctionnalité d'interception, on retrouve des mesures comparables à la version Java "pur".
- JAC effectue également de l'interception pour gérer des listes d'aspects tissés autour d'un objet. Cette interception est gérée à l'aide de réécriture de *bytecode* et de réflexion Java. Le coût de ces mécanismes sont évidents. Notons que le même test réalisé avec des frameworks AOP dynamiques utilisant des techniques comparables à celles de JAC donne des résultats identiques : 1676 ms pour JBoss AOP 1.0b3, 1777 ms pour JAsCo Jutta, 1802 ms pour AspectWerkz 1.0b2.

4.2.4 Conclusion sur l'étude

Même si les trois frameworks (Fractal, Kilim et JAC) choisis pour cette étude ont des objectifs techniques différents, ils adressent le problème de la séparation des préoccupations et visent à permettre la construction de logiciels complexes. Fractal aborde le problème via l'assemblage de composants, Kilim via leur configuration et JAC via la programmation par aspects.

L'étude de cas traitée est un service distribué de partage de données selon l'algorithme de cohérence de K. Li & P. Hudak (voir section 4.2.1). Bien que de taille modeste, cette application est suffisamment significative pour mettre en avant six aspects non-fonctionnels à composer avec le code métier (voir section 4.2.2).

La réalisation de cette étude de cas avec les trois frameworks nous a permis d'aboutir aux observations de la section 4.2.3 et aux conclusions suivantes.

La composition est structurelle avec Fractal et Kilim (ADL XML), tandis qu'elle relève plus du domaine comportemental avec JAC (processus de tissage). La séparation des préoccupations étant intimement liée à la conception de l'application, il n'est pas surprenant de constater qu'aucune des trois approches mentionnées ne l'influence. Le code est moins dispersé avec JAC, alors qu'avec Fractal et Kilim, les appels aux services techniques restent dispersés dans l'application. L'évolutivité de l'application est donc moins complexe avec JAC. La vision de l'application est claire avec Fractal alors que la richesse des concepts de Kilim nuit plus à son appréhension. Néanmoins, contrairement à Fractal, Kilim n'impose aucune contrainte au niveau du code de l'application.

Quant à JAC, la complexité du tissage engendrée par l'approche AOP n'est pas à négliger. En terme de code, le source Java Kilim reste "pur" mais la partie ADL XML est plus longue que celle de Fractal. Les codes de Fractal et JAC sont comparables. Finalement, en terme de performance, seul Kilim n'introduit pas d'indirections dans le code, ce qui fournit les meilleures performances. Les indirections de JAC sont nettement plus coûteuses que celles de Fractal. En guise de remarque finale, on peut constater qu'aucune des trois approches ne l'emporte nettement, même si la clarté de Fractal, l'évolutivité de JAC et les performances de Kilim peuvent être mises en avant.

Notons que les critères de comparaison qui nous ont permis d'aboutir à ces observations auraient pu être complétés. Des comparaisons en terme de cycle de vie des applications, de déploiement ou de maintenance auraient pu être abordées. Nous les reportons à une étude future.

En terme de perspectives, nous pouvons souligner que comme toute approche expérimentale, cette étude nécessiterait d'être complétée avec d'autres applications. En particulier, malgré la taille modeste de l'application, le nombre d'aspects est important. Il n'est pas acquis que pour d'autres applications, notamment comportant une partie métier plus significative, les conclusions seraient identiques. D'autre part, les avantages comparés de l'approche assemblage/configuration à la Fractal/Kilim, et de l'approche par aspects à la JAC, plaideraient pour une approche mixte dans laquelle les composants pourraient être assemblés, mais aussi se voir étendus par des services non-fonctionnels développés sous forme d'aspects.

4.3 Complémentarité aspect-composant

Les paradigmes du développement à base de composants (CBSD pour *Component-Based Software Development*) et à base d'aspects (AOSD pour *Aspect-Oriented Software Development*) peuvent être vus tous les deux comme des successeurs de l'approche objet. D'autres approches comme les traits [155] et *classbox* [16] postulent à ce statut de technique post-objet, mais les composants et les aspects semblent être celles qui retiennent le plus l'attention des chercheurs et des développeurs. Loin d'être en concurrence, les composants et les aspects apportent à mon sens des réponses complémentaires aux limites des objets.

Dans cette section, nous présentons deux expériences qui visent à rapprocher aspects et composants. La première (section 4.3.1) est un modèle (FAC pour *Fractal Aspect Component*) dans lequel aspect et composant sont des entités de première classe et peuvent être utilisés conjointement. La seconde expérience (section 4.3.2) montre comment les aspects peuvent être utilisés dans la mise en œuvre d'un modèle de composants (en l'occurrence Fractal [27]). Ces deux expériences sont conduites dans le cadre du contrat de recherche avec France Telecom R&D. La première fait l'objet de la thèse de N. Pessemier dont j'assure l'encadrement. Des résultats préliminaires ont été présentés dans [138] et [137].

Analogies

Comme nous venons de le mentionner, l'objectif commun des composants et des aspects est de répondre à un certain nombre de limites de l'approche objet. Il s'agit dans les deux cas d'offrir une meilleure **structuration** des applications et de mieux traiter la **composition** des entités logicielles. Pour atteindre ces objectifs, les composants et les aspects adoptent deux points de vues complémentaires : les aspects modularisent des fonctionnalités transversales, tandis que les composants spécifient leurs dépendances et fournissent des schémas de "câblage" pour décrire les

applications. Ces objectifs sont en fait complémentaires puisque les composants n'abordent pas le problème de la transversalité, tandis que les aspects ignorent le côté architectural des composants.

La **structuration** passe, dans le cas de l'AOP, par la définition d'aspects et de codes advice. Les composants sont vus, quant à eux, comme des boîtes noires qui ont aussi un comportement sans qu'aucune autre hypothèse ne soit faite sur la nature de ce comportement. À ce niveau, il apparaît donc que rien n'empêche de considérer que le comportement d'un aspect puisse être exprimé de la même façon que celui d'un composant. Rétrospectivement, la même fusion s'est opérée dans les frameworks AOP pour les objets (JAC, JBoss AOP, AspectWerkz, etc.) : les comportements des aspects sont exprimés avec des classes et des méthodes.

De cette constatation, il résulte que l'on peut considérer que les comportements des aspects et des composants peuvent être exprimés avec des entités logicielles identiques que nous appellerons composant d'aspect.

Postulat 4.1 *Un composant d'aspect est un composant qui implémente le comportement d'une fonctionnalité transversale à une application programmée sous forme de composants.*

En ce qui concerne la **composition**, les composants offrent des connecteurs et/ou des liaisons pour exprimer un schéma de câblage. Selon les modèles, les liaisons (ou les connecteurs) peuvent être associées ou non à un comportement. Il s'agit dans tous les cas d'assembler deux entités logicielles et d'exprimer la façon dont elles interagissent. La notion de coupe en AOP a un objectif similaire. Elle permet de désigner les emplacements stratégiques où les aspects sont greffés. Bien que le terme ne fasse pas parti du vocabulaire aspect, une **liaison** est ainsi créée entre un aspect et les entités de base aspectisées. Le modèle FAC que nous présentons dans la section suivante réifie cette liaison. D'autres approches, comme FuseJ [167], exploitent également cette analogie. Cette liaison peut être modifiée dynamiquement dans certains modèles de composant. De même les approches AOP dynamiques autorisent de tels changements entre entités de base et aspects.

Postulat 4.2 *De cette constatation, il résulte que l'on peut considérer que tissage d'aspect et assemblage de composants peuvent être considérés comme deux visions d'une notion plus générale de liaisons entre entités logicielles.*

La liaison que nous venons de mettre en avant est explicite dans le cas des composants. Elle y est défini en général avec des ADL qui permettent de désigner les interfaces (ou les ports, ou les connecteurs, selon la terminologie du modèle de composants) reliées. La définition de l'architecture complète passe par l'énumération exhaustive de toutes les liaisons. Les points d'ancrage des liaisons représentent des services requis et fournis. Ils sont en général fortement typés et doivent respecter une relation de sous-typage (l'extrémité doit être un sous-type de l'origine).

Dans le cas des aspects, cette liaison est implicite. Elle se définit avec un langage de *pattern* pour exprimer des coupes. En quantifiant les éléments de base aspectisés, ce style de définition offre une certaine concision lorsqu'il s'agit de relier de nombreuses localisations. Les points d'ancrage des liaisons sont définis de façon moins formelle qu'avec les composants : à l'origine de la liaison, il s'agit d'un *join point*, et à son extrémité on trouve l'API d'interception du code advice (ou du wrapper, ou de l'intercepteur selon la terminologie du modèle AOP). Les interfaces liées ne sont donc pas de même nature que dans le cas des composants. Ce ne sont pas des interfaces "métier" correspondant au service invoqué, mais des interfaces génériques dépendant du mode de fonctionnement du langage AOP.

4.3.1 Fractal Aspect Component

Fractal Aspect Component (FAC) est un projet mené dans le cadre de la thèse de N. Pessemier en collaboration avec France Telecom R&D. FAC est un modèle qui transpose au niveau composant les principes de la programmation orientée aspect tel qu'ils existent dans les frameworks objet comme JAC.

FAC repose sur le modèle Fractal [27]. Il l'étend pour incorporer les notions d'aspect, de coupe, de point de jonction et de tissage. En fait, cette extension est minimale et repose essentiellement sur les deux analogies énoncées précédemment : les aspects sont des composants et le tissage est une liaison.

La définition actuelle de FAC est basée sur Fractal. Cependant, la légèreté de ce modèle (les concepts sont simples et concis) fait que FAC pourrait être transposé dans d'autres modèles de composants. Le modèle FAC possède deux implémentations réalisées par N. Pessemier : Julius qui est basée sur Julia, l'implémentation de référence de France Telecom R&D, et FACAOKell qui est basée sur AOKell, l'implémentation de Fractal présentée section 4.3.2.

4.3.1.1 Principes

Fractal, comme la plupart des modèles de composants, part du principe que les composants sont des boîtes noires exposant des interfaces requises et fournies. La façon dont le composant est implémenté ne relève pas du modèle. Un même composant peut être implémenté de différentes façons (ensemble de fonctions comme dans COM, une classe, plusieurs classes, etc.) même si souvent, le composant est implémenté avec une classe. On se retrouve donc face à un schéma à deux niveaux : celui du composant et celui de son implémentation (le niveau objet de la figure 4.4).

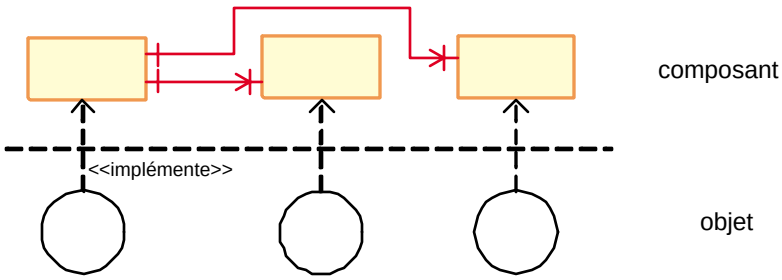

FIG. 4.4 – Niveaux composant et objet.

Le modèle FAC applique au niveau composant les concepts de l'AOP tels qu'ils sont définis au niveau objet. Comme nous le présentons dans la section suivante, des points de jonction sont définis sur les éléments des composants et un modèle d'implémentation des aspects est fourni. Bien que FAC aborde seulement le niveau composant, rien n'interdit d'utiliser les aspects aux deux niveaux : FAC pour le niveau composant et JAC, AspectJ ou tout autre langage au niveau objet. L'unification de ces deux niveaux dans un framework commun est une perspective sur laquelle nous reviendrons en conclusion.

De la même façon que pour les aspects au niveau objet, les aspects au niveau composant capturent des préoccupations transversales, c'est-à-dire des préoccupations dispersées dans différents composants. D'un point de vue architectural, cela revient à définir ce que nous nommons des **domaines de contrôle** (voir figure 4.5). Face à un assemblage initial de composants (en clair sur la

figure), chaque composant d'aspect matérialise un domaine qui se superpose à l'assemblage initial. Les composants inclus dans ce domaine sont ceux pour lesquels l'aspect s'applique. Par exemple pour un aspect de persistance, le domaine associé comprend l'ensemble des composants devant être pris en compte par ce service. Comme pour les objets, un même composant peut se trouver dans plusieurs domaines (c'est le cas du composant C sur la figure qui est partagé par le domaine en trait pointillé et le domaine en trait foncé). Comme les aspects au niveau objet, les domaines de contrôle peuvent implémenter des services système (sécurité, transaction, réplication, etc.) ou tout autre fonctionnalité transversale.

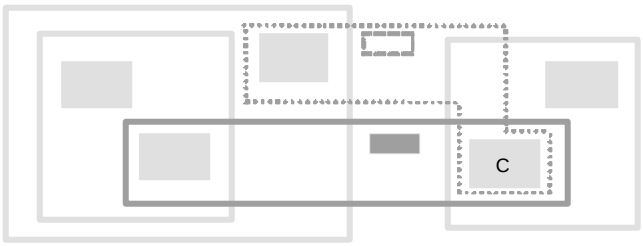

FIG. 4.5 – Différents domaines de contrôle dans une architecture composants.

4.3.1.2 Composant d'aspect et langage de coupe

Cette section présente l'adaptation des concepts aspects au niveau composant telle que la propose FAC.

4.3.1.2.1 Points de jonction Ce sont les points de contact entre l'application de base et les aspects. Nous ne nous intéressons ici qu'aux points relevant du niveau composant. Ce parti pris permet d'obtenir un modèle indépendant de la façon dont les composants sont implémentés. Étant donné la nature des concepts présents dans Fractal, les seuls points de jonction que nous envisageons sont les interactions au niveau des interfaces de composants. FAC gère donc deux types de points de jonction : **invocation** de méthode sur une interface requise et **exécution** de méthode sur une interface fournie. Ces deux types sont comparables aux types `call` et `execution` d'AspectJ.

Bien qu'un langage comme AspectJ offre plusieurs autres types de points de jonction, ces deux types sont suffisants au niveau composant. En effet, AspectJ définit des points correspondant aux lectures et aux écritures d'un attribut. Dans le modèle Fractal, les attributs des composants sont accessibles via des interfaces uniquement : les invocations et les exécutions d'interfaces couvrent donc ces besoins. De même, AspectJ prend en compte les créations d'instance. En Fractal, les composants sont instanciés via un composant fabrique (*factory*) qui fournit une interface de création : encore une fois les invocations et les exécutions d'interfaces suffisent. AspectJ supporte les points de jonction sur les exécutions de code advice. Comme nous le verrons ci-dessous, ceux-ci sont définis avec FAC par un composant et une interface. Finalement, AspectJ gère des points correspondant aux exceptions, aux blocs de code `static` d'initialisation de classe et aux flots de contrôle. Les deux premières notions n'ont pas d'équivalent dans le modèle composant. Quant à la dernière, N. Pessemier a montré qu'elle peut être programmé avec FAC. Finalement, FAC propose

au niveau composant, un modèle de point de jonction couvrant des besoins équivalents à ceux d'AspectJ, tout en étant plus concis.

4.3.1.2.2 Aspects Les aspects dans FAC sont implémentés comme des composants "normaux" : ils sont instanciés et gérés de la même façon que n'importe quel autre composant. Ils doivent implémenter l'interface d'interception `AspectComponent` (voir figure 4.6). Cette interface est liée à tout composant impacté par cet aspect. La figure 4.7 représente un assemblage de composants bien connu dans le monde des ADL, correspondant à d'une station service. Les deux composants fonctionnels `CashRegister` et `Bank` sont reliés à l'interface `AspectComponent` du composant d'aspect `CryptAC`. Ce dernier implémente une fonctionnalité de cryptage/décryptage des communications. La création de la liaison est effectuée lors du tissage à l'aide d'expressions de coupe que nous présentons ci-dessous. L'interface d'interception est invoquée, via cette liaison, chaque fois qu'un point de jonction survient dans le composant aspectisé.

Le composant d'aspect, comme tout autre composant Fractal, peut implémenter un comportement quelconque, fournir ou requérir des services à d'autres composants. Comme mentionné au chapitre 3 section 3.2.4, une bonne pratique de l'AOP consiste à déléguer l'implémentation du comportement à d'autres composants afin que le composant d'aspect ne conserve que la fonctionnalité d'interception. Cette séparation des tâches favorise la réutilisabilité et la clarté des applications.

De façon technique, en termes Fractal, la liaison entre un composant et un composant d'aspect est gérée par un contrôleur d'aspect. Ce contrôleur est attaché à tout composant aspectisable. Le choix de cette propriété est du ressort du développeur. Les composants d'aspect, en tant que composants "normaux", peuvent eux aussi être aspectisés. En plus des types de composant `primitive` et `composite` existant dans l'implémentation de référence de Fractal, deux nouveaux types ont été ajoutés : `FACPrimitive` et `FACComposite` pour désigner respectivement, les composants primitifs aspectisables et les composants composites aspectisables.

4.3.1.2.3 Langage de coupe Comme tout langage de coupe, celui de FAC permet de sélectionner un ensemble de point de jonction. Il offre un langage de *pattern* qui permet d'exprimer des quantifications sur les composants Fractal pour sélectionner les localisations destinées à être aspectisées. Ce langage s'appuie sur la structure des composants Fractal dont un méta-modèle simplifié est présenté figure 4.8.

Les expressions de coupe FAC sont construites sur les noms des composants et des interfaces, sur la signature des méthodes et sur le type (client ou serveur) des interfaces. À titre d'exemple, l'expression $Bank.*\,;\, CLIENTDesk\,;\, get.*() :.*$ sélectionne pour tous les composants dont le nom commence par *Desk*, les méthodes dont la signature commence par *get*, ne comprend pas de paramètre et ont un type de retour quelconque, localisées sur une interface cliente (requise) de nom *Desk*.

Une coupe est donc avant tout un élément structurel de code. Elle se manifeste à l'exécution par un ensemble de points de jonction (comme dans toutes les approches AOP existantes) et par un ensemble de liaisons.

4.3.1.2.4 Tissage Le tissage dans FAC est dynamique. Les liaisons créées entre des composants et un composant d'aspect peuvent être modifiées à tout moment. Aucune hypothèse n'est faite sur les conséquences du tissage/détissage sur la sémantique de l'application. Il est en effet difficile de fournir une solution qui soit valide pour tous les cas d'utilisation : par exemple, on pourrait

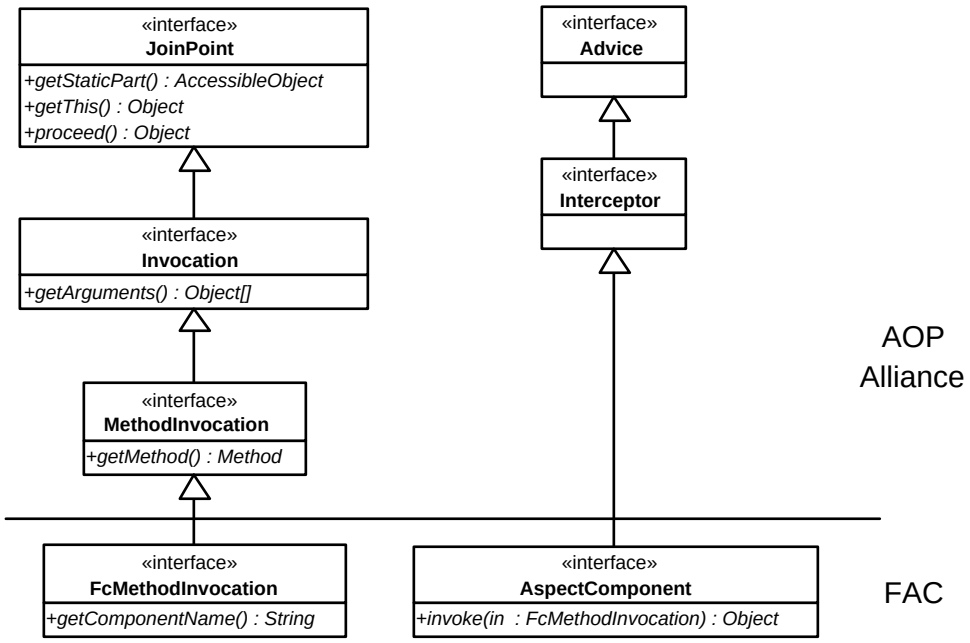

FIG. 4.6 – API d'introspection et de composant d'aspect de FAC.

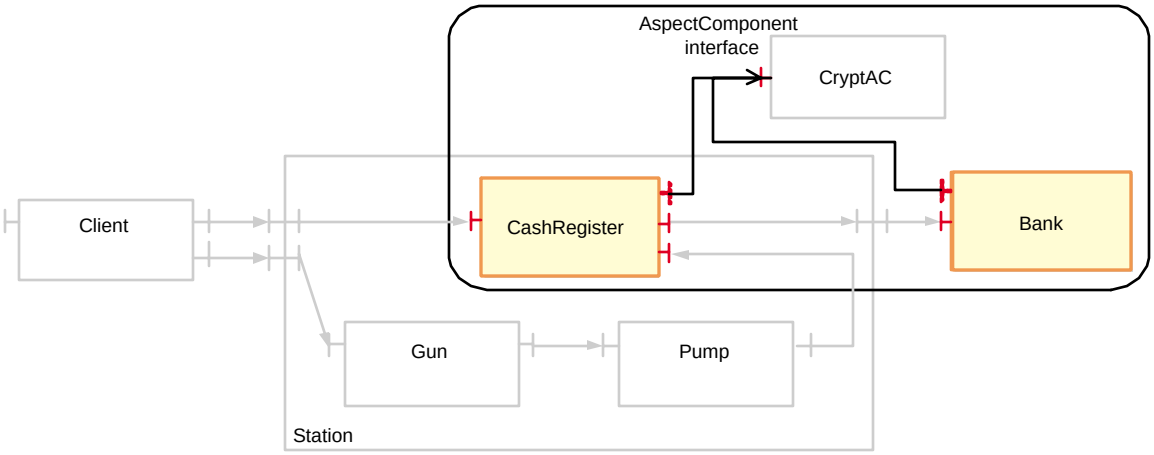

FIG. 4.7 – Liaison entre composants et composant d'aspect.

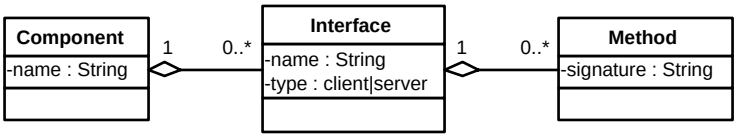

FIG. 4.8 – Méta-modèle simplifié des composants Fractal.

souhaiter que le tissage soit transactionnel afin de pouvoir revenir en arrière en cas de problème. Une telle qualité de service a néanmoins un coût qui peut être jugé prohibitif dans certains cas d'utilisation. Pour cela, la sémantique du tissage est laissée à la charge du développeur.

FAC définit une API pour gérer le tissage (voir figure 4.9). Une caractéristique importante de cette API réside dans la possibilité de créer et détruire des liaisons de tissage. Elle offre également des méthodes d'introspection pour déterminer par exemple, l'ensemble des composants inclus dans une coupe. Le langage de description d'architecture de Fractal a également été étendu afin de pouvoir exprimer de façon plus synthétique qu'avec une API des opérations de tissage. On aboutit ainsi des architectures dont la mise en place se fait à la fois par assemblage et par tissage.

```
interface AspectController {

  /** Méthode de tissage. */
  void weave(
    Component root, AspectComponent ac,
    ItfPointcutExpr pcd, String acName );

  /** Méthode de détissage. */
  void unweave( Component root, Component ac );

  /** Réordonnancement des aspects tissés sur le composant courant. */
  String[] changeACorder( String acName, int newPosition );

  /** Méthodes d'introspection de coupes. */
  String[] listACNames();
  Component[] listAC();
  Component[] listCrosscutComps( Component root, Component ac );
  Pointcut aspectizableComps( Component root, ItfPointcutExp pcd );
}
```

FIG. 4.9 – API de tissage de FAC.

Techniquement, en termes Fractal, lors d'un tissage une liste de références vers des composants d'aspects est gérée au niveau de chaque composant aspectisé. L'ordre dans cette liste définit l'ordre d'application des aspects. Cet ordre peut être changé localement ce qui représente un degré d'intervention plus fin que dans les approches comme AspectJ ou JAC où l'ordre des aspects est global. En plus de cette gestion de référence, un composite est crée pour chaque opération de tissage. Il englobe le composant d'aspect et tous les composants aspectisés, qui sont alors partagés avec leurs composites d'origine et éventuellement avec les composites d'autres domaines

de contrôle. Il matérialise donc un domaine de contrôle (voir section 4.3.1.1). Une double relation existe donc entre un composant d'aspect et ses composants aspectisés (liaison et inclusion).

4.3.1.3 Travaux connexes

Cette section présente quelques travaux proches de FAC ayant mis en œuvre un rapprochement entre les notions de composant et d'aspect.

Fractal-AOP [53] est une extension du modèle Fractal conçue à l'École des Mines de Douai. Comme FAC, Fractal-AOP étend le modèle de composants Fractal avec des aspects. Une première différence entre ce modèle et le notre réside dans la gestion du tissage. Alors que nous envisageons le tissage comme une liaison primitive entre des composants d'aspect et des composants de base (cette liaison peut être multiple lorsque le même aspect impacte plusieurs composants de base), Fractal-AOP introduit une liaison composite qui met en jeu une instance de composant dit de tissage. Le choix d'une liaison primitive limite le nombre d'instances et amène selon nous une architecture plus simple à gérer. Dans [53], il est mentionné que le composant de tissage gère aussi la coupe. En ce qui concerne FAC la coupe est spécifiée dans l'extension de l'ADL. Par ailleurs, les points de jonction envisagés par Fractal-AOP sont différents des notres : nous nous limitons aux éléments architecturaux présents dans Fractal (interfaces, méthodes et donc appels ou réception de méthodes), laissant de côté les points de jonction de type lecture-écriture d'attributs qui appartiennent selon nous au niveau objet. A contrario Fractal-AOP prend en compte ces points de jonction et d'autres (connexion, déconnexion, ajout, suppression de composants, changement d'état, changement de structure). La granularité des points de jonction n'est donc pas tout à fait la même. Finalement, au niveau de l'implémentation Fractal-AOP est implémenté en Smalltalk comme une extension de FracTalk (implémentation des spécifications Fractal en Smalltalk réalisée par N. Bouraqadi).

JAsCo [166] est un modèle de composants conçu à la Vrije Universiteit Brussel. JAsCo étend le modèle Java *bean* en introduisant les notions d'*aspect beans* et de *connectors*. Un *aspect bean* décrit où et quand (les notions de code advice et de coupe en AOP) appliquer un comportement indépendant de tout contexte, en utilisant un type de classe interne appelé *hook*. Les *connectors* déploient les *hooks* dans un contexte spécifique. Un des grands apports de JAsCo réside dans son langage de coupe, ses *connectors* permettant de définir un contrôle plus fin que le langage de coupe d'AspectJ, sur l'ordre des aspects à exécuter. Finalement nous pouvons remarquer que la gestion de *hooks* et de *connectors* est complètement centralisée et gérée par un registre de *connectors* qui a été récemment optimisé grâce au mécanisme HotSwap de Java dans le but de réduire le coût dont souffre toute approche dynamique. Le projet JAsCo se continue actuellement dans le cadre du projet **FuseJ** [167].

DAOP [139] est un modèle de composants conçu à l'Université de Málaga. C'est une plate-forme dynamique distribuée où aspects et composants sont des entités de premier ordre composées à l'exécution par une couche intergicielle. La gestion des composants et aspects est complètement centralisée et toutes les informations sur l'architecture et ses entités sont enregistrées dans la couche intergicielle. Dans une première phrase, aspects et composants sont décrits par un langage spécifique qui permet la définition des interfaces des rôles et des liaisons. Ensuite, à l'exécution les

composants et les aspects sont liés en suivant les spécifications de la couche intergicielle. L'originalité de l'approche DAOP réside dans le nom de rôle unique attribué à chaque entité. De cette manière les communications sont effectuées en précisant ces noms et non des références d'objets.

Jiazzi [110] est un modèle de composants conçu à l'Université d'Utah. Jiazzi étend le langage Java pour des composants binaires compilés séparément et liés de manière externe en tant qu'unités. Les unités sont des conteneurs de classes Java pré-compilées et peuvent être de deux types : les *atoms* (construits depuis des classes Java) ou les *compounds* (construits depuis d'autres *atoms* ou *compounds*). De nouveaux comportements peuvent être ajoutés aux méthodes et attributs sans modifier le code source, grâce à des mécanismes de classe ouverte (*open classes*) et de signature ouverte (*open signature*) basées sur l'utilisation des *mixins*. Pour résumer, Jiazzy effectue une séparation des préoccupations au niveau des classes et offre des mécanismes d'enrichissement grâce aux *mixins*.

4.3.2 Un modèle de composant ouvert et sa construction en AspectJ

Dans la section précédente, nous avons montré comment les concepts de composants et d'aspects peuvent être unifiés afin d'aboutir à un modèle de programmation offrant ces deux concepts. Dans cette section, nous présentons une deuxième expérience, baptisée AOKell [159], d'utilisation conjointe des aspects et des composants : il s'agit cette fois d'introduire de la flexibilité et d'"ouvrir" le niveau de contrôle du modèle de composants Fractal à l'aide d'aspects AspectJ. Notons que ces deux expériences sont complémentaires : N. Pessemier a réalisé une implémentation de FAC avec AOKell. Comme pour FAC, le travail sur AOKell est réalisé dans le cadre d'un contrat avec France Telecom R&D.

4.3.2.1 Principes

Une des originalités du modèle de composant Fractal [27, 28] par rapport aux modèles existants est de séparer clairement deux niveaux : métier et contrôle. Le premier correspond aux services fournis par le composant, tandis que le second représente un niveau de contrôle et de supervision du composant qui s'attachent à certaines de ses propriétés non fonctionnelles : cycle de vie, liaison, nommage, etc. Cette séparation aboutit à un modèle au pouvoir d'expression élevé dans lequel un niveau de contrôle, que nous désignerons sous le terme de membrane, peut être appliqué à l'identique pour différents composants, et réciproquement, dans lequel un même composant peut être associé alternativement (mais pas dynamiquement) à différents types de membrane. La membrane joue donc le rôle d'un niveau méta vis-à-vis du composant qu'elle contrôle.

L'implémentation de référence de Fractal, Julia, fournit en standard douze types de membranes. Le framework Dream [90, 91] étend Julia et complète cet ensemble avec d'autres politiques de contrôle. Cette approche est séduisante puisqu'elle permet de ne pas être limité à une seule politique de contrôle, mais d'adapter celle-ci aux besoins des développeurs. On retrouve ici une démarche analogue à celle mise en œuvre dans le protocole à méta-objet Reflex [170] dans lequel le degré de réification peut varier en fonction des besoins.

Néanmoins, bien que différentes membranes puissent être fournies pour personnaliser le contrôle, leur implémentation reste ad-hoc et doit se conformer au cadre défini par Julia : il s'agit de programmer des mix-in fournissant des fonctionnalités de contrôle et collaborant avec les autres mix-in de la membrane.

Face à ce constat, notre objectif, dans l'expérience AOKell est double : réduire la complexité de programmation de nouveaux contrôleurs en remplaçant la technique des mix-ins par des aspects AspectJ [82] et de standardiser l'écriture des membranes à l'aide d'assemblage de composants.

4.3.2.2 Aspects de contrôle

La première contribution d'AOKell repose sur l'utilisation d'aspect pour le développement de contrôleurs Fractal. Elle part de la constatation qu'un contrôleur a besoin d'étendre les fonctionnalités du composant contrôlé et d'intercepter ses fonctionnalités primaires pour en modifier le comportement. En termes AOP, ces deux points correspondent respectivement aux notions d'introduction et de *code advising* (voir chapitre 3, section 3.2.3).

Le schéma de principe des aspects dans AOKell est illustré figure 4.10. Chaque aspect correspond à un contrôleur qui étend et contrôle le composant. L'aspect n'implémente pas lui-même la logique de contrôle, mais selon la bonne pratique de l'AOP mentionnée chapitre 3 section 3.2.4, elle la délègue à une instance.

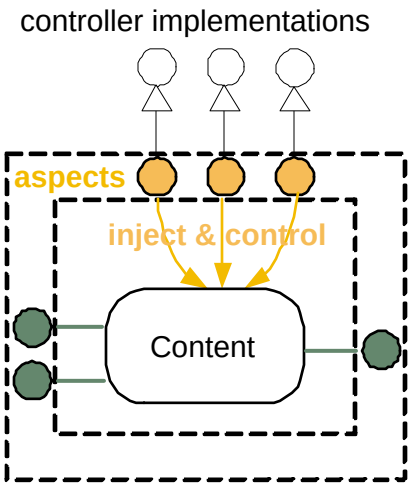

FIG. 4.10 – Principe des aspects dans AOKell.

Nous pensons que l'utilisation d'aspects pour le développement de contrôleurs présente plusieurs avantages : facilité de développement accrue et plus haut niveau d'abstraction par rapport à un mécanisme de mix-in, réduction du temps d'apprentissage pour le développement de nouveaux contrôleurs et donc expérimentation plus facile de nouvelles politiques de contrôle, meilleure cohabitation avec les autres outils du génie logiciel (test unitaire, génération de documentation, etc.), meilleure intégration avec les IDE courants (par exemple Eclipse), débuggage des contrôleurs plus facile. Par ailleurs, dans les cas standards, les performances d'AOKell sont identiques à celles de l'implémentation de référence, Julia. Par contre, Julia implémente des politiques d'optimisation agressives qu'AOKell ne supporte pas. Finalement, il s'agit, comme souvent, d'un choix entre haut niveau d'abstraction et performance.

4.3.2.3 Membranes sous forme de composants

La deuxième contribution d'AOKell porte sur la standardisation de l'écriture des membranes. Il s'agit de bénéficier pour l'ingénierie des membranes des mêmes avantages de ceux que l'on retire des composants pour l'écriture des fonctionnalités de base. Nous avons donc mis en place dans AOKell un mécanisme qui permet d'écrire les membranes (des composants Fractal), à l'aide de composants Fractal (voir figures 4.11 et 4.12).

AOKell est ainsi un système dans lequel le niveau de base et le niveau méta sont écrits avec les mêmes concepts. L'extension du niveau méta, donc la définition d'une nouvelle politique de contrôle, passe par l'écriture de nouveaux composants de contrôle que l'on introduit dans une membrane. Pour l'instant cette introduction n'est pas dynamique au sens où le type de la membrane est fixé au moment de la création du composant et ne peut pas être changé. Nous comptons profiter des fonctionnalités à venir de la version 5 d'AspectJ, notamment en terme de tissage à l'exécution pour fournir cette fonctionnalité.

4.4 Conclusion

Ce chapitre a présenté nos résultats de nos travaux sur les composants. Le rapprochement des notions de composant et d'aspect a été le fil conducteur de ces travaux.

La section 4.2 présente une étude comparative de deux frameworks composants (Fractal et Kilim) et d'un framework AOP (JAC). Elle a permis de concevoir et de réaliser une application de gestion de documents partagés dans ces trois environnements. Les conclusions de l'étude, fournies en détail à la section 4.2.4, soulignent les points forts de chacun des trois frameworks : clarté de l'architecture pour Fractal, évolutivité pour JAC et performances pour Kilim. Selon le point à privilégier, cette conclusion peut servir de grille de choix.

La section 4.3.1 présente le modèle Fractal Aspect Component (FAC) unifiant les notions de composants et d'aspects. Ce modèle a été conçu et implémenté comme une extension du modèle Fractal. Les deux résultats majeurs obtenu consistent à avoir obtenu un modèle dans lequel les aspects s'implémentent comme des composants, et où le tissage d'aspects est une forme de liaison entre composants. De façon connexe, nous avons défini la notion de domaine de contrôle qui matérialise au sein d'un composant composite, l'ensemble des composants impactés par un aspect. Les domaines de contrôle définissent donc des plans architecturaux qui se superposent au plan de base de l'application. À la manière de calques, les domaines de contrôle fournissent différentes vues de l'application selon l'aspect auquel ils sont associés : sécurité, persistance, transactions, etc. Finalement, FAC a montré qu'il est possible de transposer et d'adapter au niveau composant, les résultats acquis en terme d'AOP au niveau objet. Cette transposition ne remet nullement en cause le niveau objet. En effet, il est tout à fait possible d'envisager des applications où le niveau composant est aspectisé avec FAC et le niveau objet (i.e. les objets implémentant les composants) est aspectisé avec AspectJ, JAC ou tout autre approche AOP de niveau objet.

La section 4.3.2 présente une expérience baptisée AOKell, qui consiste à rendre fléxible en terme de contrôle, le modèle de composants Fractal. Fractal est un modèle dans lequel un niveau méta contrôle et supervise le fonctionnement des composants de base. Jusqu'à présent, ce niveau méta était conçu de façon ad-hoc avec des techniques objet. Nous avons montré que ce niveau méta pouvait être lui même conçu en terme de composants et intégré au niveau de base avec des aspects. Nous obtenons ainsi un modèle de composants réflexif dans lequel le niveau méta

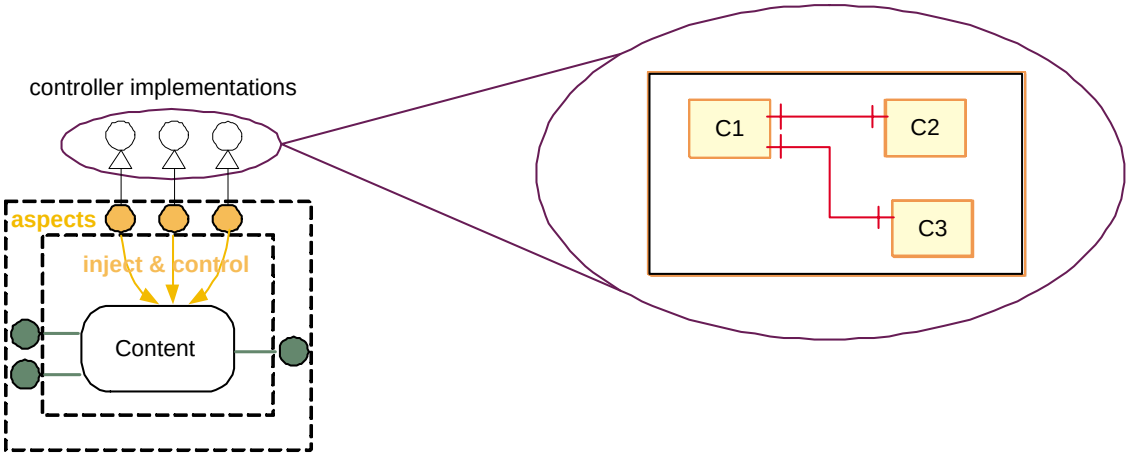

FIG. 4.11 – Principe des membranes à base de composants.

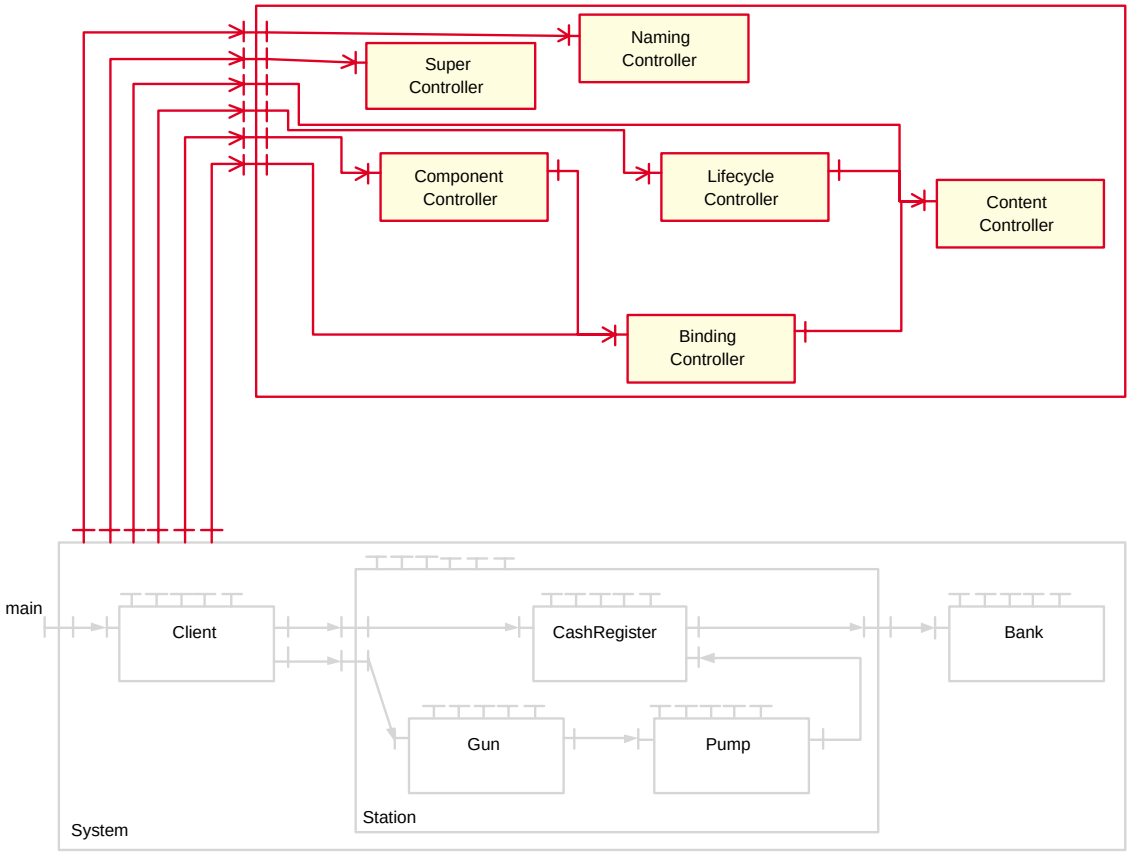

FIG. 4.12 – Association entre un composant et sa membrane.

est conçu et architecturé avec les mêmes concepts que le niveau de base. Ce résultat original ouvre des perspectives en terme d'ingénierie des propriétés non fonctionnelles. Sous ce terme, nous rassemblons l'ensemble des services systèmes tels que le nommage, la persistance, les transactions, etc. qui sont offerts aux applications. Ces services sont typiquement offerts par la niveau méta des composants. Avec AOKell, nous espérons faire progresser l'ingénierie de ces services et offrir aux concepteurs des moyens plus simples et puissants pour l'intégration de leurs services non fonctionnels.

Chapitre 5

Conclusion & perspectives

Sommaire

Ce mémoire présente les travaux de recherche que j'ai effectué depuis 1999 au sein du thème "Systèmes Répartis et Coopératifs" (SRC) du LIP6 à l'Université Pierre & Marie Curie (Paris 6), et simultanément, à partir de septembre 2003, dans le projet Jacquard de l'INRIA Futurs à Lille. Ces travaux portent sur la réflexivité, les aspects et les composants. Ils sont appliqués au domaine des intergiciels (*middleware*). Il s'agit de proposer des nouveaux concepts, de nouvelles méthodes et de nouveaux outils pour développer des intergiciels et des applications réparties. Ce domaine est caractérisé par un nombre élevé de fonctionnalités à faire cohabiter pour obtenir un logiciel fini.

L'intégration de ces fonctionnalités est un problème d'autant plus difficile que chaque fonctionnalité prise séparément est elle-même parfois complexe. Elle constitue le fil conducteur de ce mémoire. Je me suis attaché à proposer diverses solutions pour intégrer des services dans une application CORBA (chapitre 2, section 2.2.2), pour intégrer des aspects afin de construire un intergiciel ou une application répartie (chapitre 3) ou pour intégrer des aspects dans des applications à base de composants (chapitre 4).

Les sections suivantes résument les travaux présentés dans ce mémoire et proposent pour chacun d'eux des perspectives spécifiques. La section 5.4 présente sous forme de projet de recherche, des perspectives globales.

5.1 Construction de services pour les intergiciels

Le chapitre 2 présente une expériences de conception et de réalisation d'un service réflexif pour un intergiciel CORBA. Il s'agit d'un service d'observations d'exécutions réparties.

Notre contribution sur cette expérience est double : nous obtenons des traces d'exécutions des communications, de la concurrence, de la synchronisation et du partage de données au sein de

l'application et, deuxièmement, l'intégration de ce service est rendue transparente à l'aide de la réflexivité. En terme de réalisation, ce service repose sur le langage réflexif à la compilation OpenJava [171] et sur l'ORB CORBA JacORB [26]. Néanmoins, rien n'est spécifique à cet ORB dans ce service qui peut être utilisé avec n'importe quel intergiciel CORBA.

Sur le premier point, nous sommes reparti de la relation d'ordre causale dite *Happened before* [86] de Lamport. Cette relation permet un ordonnancement causal d'événements distribués à partir des envois et réceptions de messages asynchrones. Nous avons étendu cette relation afin de l'adapter à un environnement objet réparti à base de communications de type RPC (requête/réponse) avec gestion de la concurrence, des synchronisations et des ordonnancements transactionnels des lectures/écritures sur les données partagées. Nous avons défini un mécanisme d'estampillage qui permet de dater de façon logique les événements observés. Un outil de visualisation interactif du graphe des événements a été réalisé.

Sur le second point, la transparence d'intégration, nous avons eu comme objectif de pouvoir instrumenter de façon transparente n'importe quelle application CORBA. Nous nous sommes appuyés sur les capacités réflexives du langage OpenJava. Les événements à observer sont réifiés et capturés à l'aide méta-programmes. L'intérêt de l'approche réside donc dans le fait que les programmes de base restent inchangés et que l'on sépare donc clairement la logique d'observation du reste de l'application.

Cette expérience a fait l'objet de plusieurs publications, dont [49], qui en présente un bilan global.

Perspectives

Cette étude se caractérise par une transparence plus élevée que dans le cas d'une approche par librairie où l'insertion du service est intrusive. Néanmoins cette transparence ne dispense pas d'une étape de configuration : par exemple, dans le cas du service d'observation, il est nécessaire de paramétrer le méta-programme pour lui indiquer quels objets doivent être observés et selon quelles modalités, et dans le cas du service de réplication, quels composants doivent être répliqués et avec quelles options. Ce travail indispensable pour ne pas être confronté à un volume ingérable de traces ou de composants répliqués, n'est néanmoins pas adressé de façon satisfaisante par les approches réflexive ou intégrée. Aucun concept n'est disponible pour décrire cette liaison qui reste noyée au sein du méta-programme ou du service. Les aspects et les composants apportent sur ce point une avancée en externalisant, avec un langage de coupe dans un cas, avec la notion d'architecture logicielle dans l'autre, cette information.

La deuxième problématique concerne le caractère pervasif des services adjoints aux intergiciels : il n'est pas simple de délimiter précisément les frontières du service. Par exemple, le service d'observation a un impact non seulement sur le langage d'implémentation (Java en l'occurrence), mais aussi sur le langage IDL de description des interfaces. Dans d'autres cas, comme par exemple la réplication de composants EJB de type (*entity bean*), le service impacte à la fois sur le serveur d'application J2EE mais également le SGBD stockant les données. On constate donc que l'adjonction d'un service n'impacte pas seulement un langage ou un système mais est profondément multi-langages et/ou multi-systèmes. D'autres services peuvent illustrer ce propos : par exemple la gestion des droits dans les applications client/serveur de type J2EE. Elle n'est pas confinée au langage Java, mais concerne également la configuration du serveur HTTP, du conteneur EJB

et du SGBD. Il n'existe pas à ce jour d'approche permettant d'exprimer cette problématique de façon globale et intégrée. Il apparaît donc qu'une démarche holistique[1] de la configuration des services constitue une voie de recherche qui permettrait une meilleure intégration, composition et configuration des services pour les intergiciels.

5.2 Aspects

Le chapitre 3 présente mes contributions au domaine de la programmation orientée aspect (AOP pour *Aspect-Oriented Programming*) [84]. Elles concernent la plate-forme orientée aspect dynamique JAC (section 3.3) et le développement de services à l'aide d'aspects pour les plates-formes à composants J2EE et CCM (section 3.4).

La plate-forme JAC (*Java Aspect Component*) offre un environnement complet de développement d'applications orientées aspect. Rappelons que les aspects sont une technique indépendante des styles de développements (i.e. ils peuvent être appliqués aussi bien aux objets, qu'aux procédures ou aux fonctions) qui vise à lutter contre les phénomènes de dispersion (*scattering*) et d'entremêlage (*tangling*) du code. Un aspect est ainsi une entité logicielle qui modularise une fonctionnalité transversale et définit des règles d'intégration avec le reste de l'application. Les aspects permettent ainsi d'aboutir à des applications plus modulaires et donc plus facilement développables, maintenables et plus évolutives.

La contribution principale du projet JAC a été de mettre en œuvre la technique dite de tissage dynamique. Alors que AspectJ [83], qui est le langage de référence du domaine est un compilateur, JAC est un framework dans lequel les aspects sont des entités qui ont une existence propre à l'exécution. De plus, la liaison entre un aspect et les objets qu'il impacte, peut être créée, modifiée ou supprimée à chaud fournissant ainsi un tissage dynamique. JAC a apporté une contribution majeure à ce domaine en étant l'une des premières plates-formes à proposer au niveau international, ce concept d'AOP dynamique. Elle a été rejointe par d'autres plates-formes comme JBoss AOP, AspectWerkz ou Spring AOP.

Une deuxième contribution concerne la problématique des conteneurs ouverts. En effet, avec JAC les services techniques (persistance, transaction, sécurité, etc.) fournis aux applications sont programmés sous forme d'aspects. JAC est un conteneur ouvert au sens où seuls les aspects nécessaires à l'application sont chargés. Ces aspects peuvent être déchargés ou rechargés dynamiquement en fonction des besoins. Cette caractéristique est un apport majeur par rapport à des solutions comme J2EE ou .Net dans lesquelles les services techniques sont figés.

Une troisième contribution du projet JAC porte sur la notion d'aspect distribué. Nous avons ainsi proposé la notion de coupe distribuée qui permet d'appliquer un même aspect sur un ensemble de serveurs JAC coopérant via un réseau. Cette notion de coupe distribuée est une contribution à l'unification des notions de tissage et de déploiement : tisser un aspect distribué revient à déployer un ensemble d'instances d'aspects sur des objets applicatifs hébergés par différents serveurs JAC.

Finalement, JAC aborde également d'autres problématiques importantes comme la configuration ou la composition. L'utilisation de services techniques requiert la gestion de nombreux paramètres. Dans les approches comme J2EE ou CCM, cette problématique est résolue à l'aide de fichiers XML. Cette approche n'est valable que parce que les services sont figés. Dans une approche

[1] Holistique : qui présente le caractère de la totalité.

ouverte comme celle de JAC, ces paramètres sont par définition, quelconques. JAC propose donc un mécanisme flexible et ouvert dans lequel les paramètres de configuration sont définissables en fonction des besoins. En ce qui concerne la composition, JAC promeut une approche par ordonnancement. Il s'agit de déterminer, lorsque plusieurs aspects interviennent aux mêmes localisations d'un programme, quel est leur ordre d'exécution. JAC part du principe qu'il n'y a pas de solution universelle et que cet ordre dépend de la sémantique des aspects : deux aspects identiques, par exemple trace et contrôle d'accès, peuvent être ordonnés différents selon que l'on souhaite tracer toutes les tentatives d'accès ou seulement seules qui ont abouti. JAC propose donc via un mécanisme de configuration et via une API, des moyens pour définir cet ordre d'exécution.

La plate-forme JAC a été développée en partie dans le cadre de la thèse de R. Pawlak [128] dont j'ai participé à l'encadrement. [134] est la publication majeure sur JAC. Les ouvrages [148, 149] et [132] comprennent un chapitre de présentation générale de JAC.

Les résultats présentés à la section 3.4 partent du constat que les applications EJB et CCM partagent de nombreux concepts et que leur programmation suit le même schéma. Nous avons donc proposée une couche abstraite à base d'aspects qui matérialise ces concepts et qui permet de développer de nouveaux services pour composants. Ces services peuvent alors être utilisés indifféremment avec l'une des deux technologies en sélectionnant la personnalité adéquate (EJB ou CCM) de la couche abstraite. Ce travail a été réalisé dans le cadre de la thèse de F. Legond-Aubry [94] dont j'ai assuré l'encadrement sous la direction de G. Florin.

Perspectives

Du fait de sa relative jeunesse, l'AOP ou plus généralement l'AOSD (*Aspect-Oriented Software Development*), est un domaine dans lequel beaucoup reste à faire. Je présente ci-dessous quelques perspectives de recherche qui me semblent significatives.

Vers une AOP plus symétrique Jusqu'à présent, l'AOP telle qu'elle est prônée par AspectJ tend à faire en sorte qu'aspects et classes soient deux types d'entités logicielles distinctes. Les classes définissent des données et des méthodes et les aspects définissent des coupes et des codes advice. D'un point de vue pédagogique, cette dichotomie a permis de bien faire comprendre la raison d'être des aspects. Cependant, à partir du moment où il est dit qu'un aspect modularise une fonctionnalité transversale, il est difficile de voir pourquoi cette fonctionnalité ne se développerait pas avec les mêmes concepts que les fonctionnalités "normales". Deuxièmement, on peut faire remarquer que si un code advice correspond bien à un comportement, une coupe adresse plus une problématique d'intégration de l'aspect avec le reste de l'application. Une bonne séparation des préoccupations consisterait à différencier clairement ces deux notions. Il ressort de ces deux constatations qu'une application devrait pouvoir être développée de façon modulaire à partir d'un ensemble de comportements, qu'ils soient transversaux ou non, et d'une politique d'intégration. Il faudrait donc re-symétriser l'AOP en atténuant la dichotomie classe-aspect. À long terme, cette perspective unifiante me semble garante d'une meilleure ingénierie logicielle, plus claire, plus facilement abordable et outillable, et donc au final, plus efficace.

Notons que le succès d'AspectJ a, momentanément, mis un frein à d'autres approches plus symétriques comme la programmation orientée sujet [67] (SOP), la séparation multi-dimensionnelle des préoccupations [124] (MDSoC) ou Hyper/J. Notons également que les nouveaux styles

de développements d'aspects à base d'annotations d'AspectWerkz ou d'AspectJ 5 sont un premier pas vers une re-symétrisation de l'AOP. Néanmoins beaucoup reste à faire. En ce qui nous concerne, nous pensons que cette symétrisation passe par un changement de granularité : les composants offrent une abstraction de plus haut niveau, se prêtant mieux à une symétrisation car ils séparent de façon claire les comportements, implémentés dans les composants, de la logique d'intégration, exprimée dans des langages de description d'architecture (ADL) ou des connecteurs. Nous défendons ce point au chapitre suivant, notamment dans la section 4.3.1 avec le modèle FAC.

Composition d'aspects La composition d'un aspect avec une application de base est traitée par les langages de coupe et le processus de tissage. Par contre, la composition de deux ou plusieurs aspects intervenant sur le même point de jonction est un problème difficile. Dans toutes les approches existantes, il se ramène à une question d'ordre : quels aspects doivent être exécutés avant quels autres. Cependant, cet ordre est dépendant de la sémantique des aspects ou de la sémantique que l'intégrateur souhaite donner à sa composition. Par exemple dans le cas d'un aspect de trace et d'un aspect de contrôle d'accès par *login* et mot de passe, l'intégrateur peut souhaiter tracer tous les accès, y compris ceux ayant échoué, ou il peut souhaiter ne tracer que les accès ayant abouti. Selon le cas, l'ordre des aspects n'est pas le même et il n'est pas possible d'ordonnancer automatiquement.

Face à ce problème, la piste que nous proposons est basée sur une contractualisation du comportement des aspects. Ainsi, dans [131], avec R. Pawlak, nous avons proposé le langage CompAr. Ce langage est basé sur une spécification du comportement des aspects avec des primitives pour exprimer des choix, des exécutions d'actions ou d'invocations, et des contraintes de post-exécution. À l'aide de ces primitives, le développeur d'aspect est invité à donner une vision comportementale macroscopique de ses aspects. En présence d'un ensemble d'aspect, un solveur calcule tous les chemins d'exécutions possibles et signale les conflits potentiels entre aspects. Cette étude doit être poursuivie et éprouvée sur d'autres aspects que l'étude de cas traitée dans [131].

Aspects dans les phases amont Les aspects se sont d'abord matérialisés en terme de programmation. La communauté s'est alors aperçu que les principes mis en œuvre pouvaient être utilisés dans d'autres phases du génie logicielle. Par exemple, [76] et [135] étudient les aspects dans le cadre des diagrammes de cas d'utilisation. Néanmoins, si plusieurs autres propositions existent pour d'autres phases du génie logiciel, force est de constater qu'il n'existe pas de méthode intégrée, comme c'est le cas pour l'objet, permettant de conduire un projet avec des aspects, depuis les phases d'analyse jusqu'à la génération finale de code, en passant par la conception et les tests. La mise au point d'une telle démarche globale nécessite un investissement important. Néanmoins, c'est peut-être la clé qui permettra aux aspects de franchir le pas des grands projets industriels. En effet, pour l'instant, les plus grandes applications utilisant les aspects sont des applications issues d'une refactorisation d'applications objets existantes : on peut citer l'aspectisation de IBM WebSphere [40] ou d'ORB CORBA [183]. Il n'y a pas, à ma connaissance, de grand projet informatique qui ait dès le départ pris le parti des aspects. Les raisons en sont certainement multiples, mais l'absence de démarche globale intégrée est en peut-être une.

Face à ce défi, qui nécessitera sans doute plusieurs thèses, nous nous sommes lancés dans le cadre de la thèse CIFRE de D. Diaz en partenariat avec la société de services Norsys, dans un projet qui vise à répertorier et tracer l'évolution des aspects dans les différentes phases d'analyse, conception et codage. Le but est de fournir un atelier permettant de lier les éléments d'analyse,

conception et code faisant partie d'un aspect pour pouvoir suivre et comprendre les évolutions d'un aspect au cours de son évolution.

Coupes comportementales Au niveau des concepts, nous avons constaté, comme d'autres, que la réutilisabilité des aspects est limitée par le caractère ad-hoc des coupes. Celles-ci fournissent les localisations où les aspects doivent être intégrés. Ces localisations sont définies syntaxiquement et sont donc dépendantes de l'application : en changeant d'application, il faut changer les coupes. Une piste envisagée pour lutter contre cette fragilité est de s'appuyer sur le comportement, plutôt que la syntaxe, des applications. En définissant des *patterns* de comportements, on espère que la coupe pourra s'abstraire de la syntaxe concrète des applications et s'appliquer à un plus grand nombre d'applications. Cette thématique, coupe comportementale, peut typiquement faire l'objet d'un sujet de thèse.

***Patterns* de coupe** Toujours au niveau des concepts de l'AOP, les points de jonction entre les aspects et l'application sont des entités au couplage assez faible. En effet, un aspect s'applique sur un ensemble de points de jonction, mais ces points ne présentent aucun lien logique entre eux. Dans certaines situations, il pourrait être intéressant de lier deux ou plusieurs points de jonction, par exemple les points de jonction correspondant au départ et à l'arrivée d'un appel de méthode. Cela permettrait d'envisager ces deux points comme faisant partie d'une structure commune, i.e. la liaison, plutôt que comme deux points séparés. Plutôt que des points, on aboutirait à des *patterns* de jonction qui permettrait d'obtenir un vue de plus haut niveau et donc plus architecturale du comportement des applications. Cette idée, qui apparaît dans le cadre de la thèse de O. Barais [11], est également en cours d'exploitation dans la cadre de la thèse de N. Pessemier que j'encadre.

Un langage de point de jonction ouvert Chaque langage AOP offre un ensemble de points de jonction permettant la greffe d'aspects. Face à un aspect donné, le développeur est contraint par les types de points de jonction offerts par le langage. Ces types peuvent ne pas correspondre exactement aux besoins de l'aspect. Il pourrait donc être souhaitable de personnaliser le modèle de points de jonction pour l'adapter à un problème particulier. Cette voie, qui a été explorée par ailleurs dans [37] avec le langage Josh, fait partie des pistes qui selon nous, permettrait de mieux adapter les aspects à des problèmes dédiés.

Dissémination Finalement, notons que d'un point de vue pratique, l'adoption de l'AOP pour les développements industriels tarde à se faire sentir clairement. On est certainement en présence du syndrome "encore une nouvelle technologie à intégrer" qui freine de façon tout à fait compréhensible les chefs de projets. On peut aussi arguer du manque d'outillage de l'AOP bien que AspectJ, le leader du domaine, fournisse plusieurs plug-ins pour les IDE Java Eclipse, JBuilder, JDeveloper et NetBeans, ce qui rend l'AOP plus abordable. Face à ce constat, on ne peut que militer pour des actions pédagogiques de formation et d'explications. C'est ce que, modestement, je m'efforce de faire au travers de mes enseignements de l'AOP et de mes publications à destination de la communauté industrielle. Cet effort doit être poursuivi.

5.3 Composants

Le chapitre 4 présentent mes travaux sur les composants. La ligne directrice suivie est celle de l'unification des notions de composant et d'aspect. Je défends l'idée que composants et aspects sont complémentaires. Les deux se positionnent comme des techniques apportant plus de modularité aux objets. Alors que les composants permettent d'architecturer le plan fonctionnel d'une application, les aspects modularisent les dimensions transversales et non fonctionnelles. C'est en ce sens que ces deux techniques me semblent complémentaires. Leur unification est donc de nature à faire progresser l'ingénierie logicielle. Le chapitre 4 fait le point sur deux expériences mettant en œuvre conjointement aspects et composants : Fractal Aspect Component (section 4.3.1) et AOKell (section 4.3.2).

Fractal Aspect Component (FAC) est une extension du modèle de composants Fractal [27] conçue et développée dans le cadre de la thèse de N. Pessemier, qui a débuté en 2004 en collaboration avec France Telecom R&D. La contribution majeure de FAC porte sur l'unification des concepts de l'AOP et du développement à base de composants. Ainsi, dans FAC, un aspect est implémenté sous la forme d'un composant et le tissage s'apparente à une liaison entre un composant d'aspect et des composants de base. Cette liaison est associée à une expression de coupe qui désigne, à l'aide d'un langage de patterns, les composants de base à lier au composant d'aspect. FAC introduit également la notion de domaine de contrôle. Il s'agit de superposer à l'architecture de base de l'application, les différents plans résultant du tissage des aspects. Ces plans, ou domaines de contrôle, sont matérialisés par des composites partageant les composants avec le reste de l'application. On obtient ainsi une architecture à plusieurs dimensions.

AOKell est une réalisation du modèle Fractal dans lequel le niveau de contrôle des composants est défini à l'aide d'aspects et de composants. On obtient ainsi un modèle réflexif dans lequel le contrôle et la base s'expriment avec les mêmes notions : composants, interfaces, liaisons. Les aspects réalisent l'intégration des deux niveaux. Il est possible de modifier le niveau de contrôle en lui ajoutant ou en retirant des composants ou en modifiant ses liaisons. On aboutit ainsi à une ingénierie du niveau de contrôle similaire à l'ingénierie du niveau de base. Cette expérience ouvre des perspectives intéressantes en terme d'extensibilité du contrôle. Par ailleurs, nous avons montré que les performances du modèle obtenu sont équivalentes à celles du modèle où le contrôle est défini de manière ad-hoc. Le gain en terme d'unification des concepts n'est ainsi donc pas hypothéqué par une perte de performances.

Perspectives

Préciser le modèle aspect-composant Les expériences précédentes ont permis d'esquisser un rapprochement entre composants et aspects. En guise de perspective, nous pouvons mentionner que cette voie de recherche n'en est qu'à ses prémices. Beaucoup reste à faire pour stabiliser une vision cohérente de ce rapprochement. Par exemple, nous avons pris le parti de considérer que tissage d'aspects et assemblage de composants sont deux visions d'une même relation plus générale. Il reste à déterminer comment cette relation peut être définie formellement et à bien expliciter la différence de nature entre tissage et assemblage.

Contractualiser le modèle aspect-composant La contractualisation des activités d'assemblage et de tissage est un domaine quasiment vierge. Alors que des solutions existent pour tester, vérifier ou valider le comportement d'un composant pris individuellement, il est difficile de pouvoir vérifier la validité d'un assemblage ou d'un tissage. Cette difficulté résulte d'un caractère intrinsèquement réparti : un assemblage met en jeu une coordination de comportements entre plusieurs composants. Il semble primordial pour faciliter et sécuriser l'emploi de telles techniques, que des garanties puissent être émises sur le comportement résultant d'un assemblage. Cette voie mérite d'être explorée et peut typiquement faire l'objet d'un sujet de thèse.

Une vision unifiée de l'architecture logicielle Nous avons établi que les aspect peuvent s'envisager à deux niveaux dans les modèles à composants : au niveau composant et au niveau objet. Un troisième niveau peut être ajouté, celui des architectures logicielles. Ces trois niveaux, architecture, composant et objet, peuvent indépendamment intégrer des notions issues de l'AOP. Il reste à mettre en place un framework unifié dans lequel les trois niveaux peuvent être adressés de concert.

5.4 Perspectives générales

En guise de perspectives, je propose ci-dessous quelques grandes lignes pour un projet de recherche sur l'ingénierie des architectures logicielles à base de composants et d'aspects.

Passage à l'échelle Les applications actuelles à base de composants sont encore de faible taille (quelques centaines de composants) et fortement couplées (au mieux communications distantes en mode asynchrone via des réseaux LAN, rarement des réseaux WAN). Il y a donc un besoin de passage à l'échelle pour des applications comportant plusieurs milliers de composants faiblement couplés. Le calcul scientifique haute performance est typiquement un domaine applicatif cible pour cela.

En ce qui concerne les aspects, les expérimentations actuelles (aspectisation de serveurs Web ou du serveur d'applications WebSphere d'IBM) ont permis d'identifier au mieux une demi-douzaine d'aspects différents. Par ailleurs, les études que nous avons pu mener sur le projet JAC (chapitre 3) nous ont amené à développer une librairie de quinze aspects. Il est néanmoins encore rare de trouver une application qui utilise simultanément de nombreux aspects. Il est donc nécessaire de poursuivre l'étude et le développement d'applications conséquentes pour compléter la couverture fonctionnelle des aspects, en améliorer leur compréhension et celle des mécanismes d'interactions inter aspects.

Sûreté La question de la sûreté de fonctionnement des applications à base de composants commence tout juste à être abordée par la communauté scientifique et est cruellement ignorée dans les aspects. Des tentatives de formalisation des mécanismes de tissage ou de définition d'assertions pour du code ajouté par des aspects ont été proposées par la communauté, mais elles sont loin de fournir les outils nécessaires pour pouvoir vérifier par exemple qu'un aspect peut se tisser sur une application, qu'un aspect ne remet pas en cause le bon fonctionnement d'une application ou que deux aspects ne sont pas conflictuels. Un travail de formalisation et de contractualisation des mécanismes d'assemblage de composants et de tissage d'aspects est nécessaire. Il s'agit d'atteindre dans ce domaine des résultats comparables à ce que les techniques de *model checking*, de preuve

automatique de programmes ou de logiciels de type *proof carrying* code ont pu amener ces vingt dernières années.

L'AOP et les composants sont arrivés à un point où les environnements de programmation disponibles sont utilisables dans des projets de taille industrielle et où des sociétés proposent leurs services pour cela. Néanmoins, de nombreuses étapes du génie logiciel restent à couvrir pour améliorer la qualité et la sûreté des développements. Les phases amont de conception et d'analyse sont encore peu couvertes et la formalisation et la vérification ne sont quasiment pas abordées. En particulier, les processus de tissage d'aspects et d'assemblage de composants restent une activité manuelle (i.e. programmée par un développeur). Les environnements de développement ne proposent à ce jour aucun outil pour vérifier qu'un aspect peut se tisser à une application, que le tissage d'aspects ne remet pas en cause le bon fonctionnement de l'application ou qu'un assemblage de composants fournit le service escompté qui est potentiellement différent des services rendus individuellement par les composants assemblés.

L'objectif que nous nous fixons ici consiste donc à contractualiser les composants et les aspects. Il s'agit à la fois de caractériser leur structure, par exemple en termes de typage, et leur comportement, par exemple en en donnant une représentation abstraite dans une algèbre de processus. La vérification consistera à partir de la définition d'opérateurs représentant l'assemblage et le tissage, de vérifier que les propriétés globales attendues d'une application sont respectées. La difficulté de cet objectif réside dans l'éventail des compétences à rassembler : en plus d'une expertise en tissage et en assemblage, il est nécessaire de rassembler des compétences en vérification. C'est certainement un objectif fédérateur pour un groupe de recherche particulier autour d'équipes possédant ces compétences. Le travail sur cet objectif a commencé dans le cadre de la thèse de F. Loiret qui a débuté en 2004 financée par le CEA Saclay et dont j'assure l'encadrement.

Élargir les domaines d'applications des architectures logicielles Comme tout paradigme d'ingénierie logicielle ou système, les succès (ou l'échec) des composants, de l'AOP ou des ADL se mesure à l'aune des applications qui sont développées. La "*killer*" application qui démocratisera l'utilisation de ces paradigmes reste à inventer. Rétrospectivement, on constate que le développement d'IHM a certainement joué ce rôle pour les objets. Ceux-ci ont permis de gérer de façon élégante la complexité des librairies graphiques pour les IHM, tout en les laissant suffisamment ouvertes pour que le développement de nouveaux *widgets* par réutilisation de l'existant soit aisé. Un des facteurs qui explique ce succès est certainement la symbiose atteinte entre le paradigme et le domaine applicatif : les IHM ont permis de faire évoluer les objets et réciproquement les objets ont progressé grâce aux IHM. Partant de là, l'utilisation des objets s'est répandue dans d'autres domaines.

L'objectif fixé ici consiste à appliquer l'architecture unifiée à base de composants et d'aspects proposée précédemment, à des domaines porteurs comme l'informatique ubiquitaire ou le calcul haute performance pour les grilles. Il est important de porter des efforts sur ces domaines, car il s'agit dans un cas (ubiquitaire) des applications grands publics de demain et dans l'autre (calcul) des applications scientifiques actuelles et à venir. Une des caractéristiques majeures communes à ces deux domaines est la variabilité. Dans un cas, nous avons affaire à des applications s'exécutant dans des contextes matériels (*desktop*, *laptop*, PDA, *smart phone*, etc.) et de ressources (réseau, mémoire, CPU, etc.) pouvant varier pendant l'exécution de l'application ; il s'agit de faire en sorte que l'application s'y adapte. Dans l'autre cas, les applications de calcul doivent pouvoir se redéployer dynamiquement en fonction du nombre de nœuds ou de la bande passante réseau

disponible. Dans les deux cas, il s'agit donc de disposer de supports d'exécution adaptables au contexte, de paradigmes permettant aux développeurs d'exprimer leurs politiques d'adaptation, de compilateurs, d'outils de test et de vérification prenant en compte cette adaptation. Cet objectif consiste donc à avoir une chaîne complète de développement et d'exécution prenant en compte l'adaptation pour les applications ubiquitaires et de calcul haute performance.

Bibliographie

[1] T. Abdellatif. Enhancing the management of a J2EE application server using a component-based architecture. In *Proceedings of the Component-based Software Engineering Track at the 31th IEEE Euromicro Conference (Euromicro CBSE'05)*, Août 2005.

[2] B. Achauer. The DOWL distributed object-oriented language. *Communications of the ACM*, 36(9):48–55, Septembre 1993.

[3] M. Ahuja, T. Carlson, A. Gahlot, and D. Shands. Timestamping events for inferring affects relation and potential causality. In *Proceedings of the IEEE International Computer Software and Applications Conference (COMPSAC'91)*, pages 606–611, 1991.

[4] M. Aksit, K. Wakita, J. Bosch, and L. Bergmans. Abstracting object interactions using composition filters. volume 791, pages 152–184, 1994.

[5] J. Aldrich, C. Chambers, and D. Notkin. ArchJava: Connecting software architecture to implementation. In *Proceedings of the 24th International Conference on Software Engineering (ICSE'02)*, pages 187–197. ACM Press, Mai 2002.

[6] R. Allen and D. Garlan. Formalizing architectural connection. In *Proceedings of the 16th International Conference on Software Engineering (ICSE'94)*, Mai 1994.

[7] A. Andersen, G. Blair, V. Goebel, R. Karlsen, T. Stabell-Kulo, and W. Yu. Arctic beans: Configurable and re-configurable enterprise component architectures. *IEEE Distributed Systems Online*, 2(7), 2001.

[8] IST NoE Aspect Oriented Software Development. Survey of Aspect-oriented Middleware, 2005.
`www.aosd-europe.net/documents/index.htm`.

[9] As-2 Embedded Computing Systems Committee SAE. *Architecture Analysis & Design Language (AADL)*, Novembre 2004. SAE Standards AS5506.

[10] R. Balter, J. Bernadat, D. Decouchant, A. Duda, A. Freyssinet, S. Krakowiak, P. Le Dot, M. Meysembourg, H. Nguyen, E. Paire, M. Riveill, C. Roisin, X. R. de Pina, R. Scioville, and G. Vandome. Architecture and implementation of GUIDE, an object-oriented distributed system. *Computing Systems*, 4(1):31–67, 1991.

[11] O. Barais. Construire et maîtriser l'évolution d'une architecture logicielle à base de composants. Thèse de Doctorat de l'Université Lille 1, Décembre 2005.

[12] M. Bartorello, H. Maguin, M. Blay-Fornarino, A.-M. Dery, and M. Riveill. Adjonction de services au sein d'un serveur d'EJB. In *Journées Composants 2001 (JC'01)*, Octobre 2001. `lifc.univ-fcomte.fr/~philippe/composants/papiers/`.

[13] L. Bass, P. Clements, and R. Kazman. *Software Architecture in Practice*. Addison-Wesley, 1998.

[14] T. Basten, T. Kunz, J. Black, M. Coffin, and D. Taylor. Vector time and causality among abstract events in distributed computations. *Distributed Computing*, 11, 1997.

[15] F. Baude, D. Caromel, and M. Morel. From distributed objects to hierarchical grid components. In *Proceedings of the International Symposium on Distributed Objects and Applications (DOA'03)*, Novembre 2003.

[16] A. Bergel, S. Ducasse, and R. Wuyts. Classboxes: A minimal module model supporting local rebinding. In *Proceedings of the Joint Modular Languages Conference (JMLC'03)*, volume 2789 of *Lecture Notes in Computer Science*, pages 122–131. Springer-Verlag, Juin 2003.

[17] A. Birrell and B. Nelson. Implementing remote procedure calls. *ACM Transactions on Computer Systems*, 2(1):39–59, Février 1984.

[18] G. Blair, G. Coulson, P. Robin, and M. Papathomas. An architecture for next generation middleware. In *Proceedings of Middleware'98*, pages 191–206, Septembre 1998.

[19] C. Bockisch, M. Haupt, M. Mezini, and K. Ostermann. Virtual machine support for dynamic join points. In *Proceedings of the 3rd International Conference on Aspect-Oriented Software Development (AOSD'04)*, pages 83–92. ACM Press, 2004.

[20] S. Bodoff, E. Armstrong, J. Ball, and D. Carson. *The J2EE Tutorial*. Addison-Wesley, Juin 2004. 2nd edition.
`java.sun.com/j2ee/1.4/docs/tutorial/doc/index.html`.

[21] K. Bohrer, V. Johnson, A. Nilsson, and B. Rubin. Business process components for distributed object applications. *Communications of the ACM*, 41(6):43–48, 1998.

[22] J. Bonér, J. Dahlstedt, and A. Vasseur. JRockit JVM support for AOP, Août 2005.
`dev2dev.bea.com/lpt/a/438`.

[23] G. Booch. *Software Components with Ada: Structures, Tools and Subsystems*. The Benjamin/Cummings Publishing Company, 1987.

[24] J.-P. Briot. Modélisation et classification de langages de programmation concurrente à objets : l'expérience actalk. In *Actes de Langages et Modèles à Objets (LMO'94)*, 1994.

[25] J.-P. Briot, R. Guerraoui, and K. Lohr. Concurrency and distribution in object-oriented programming. *ACM Computing Surveys*, 30(3):291–329, Septembre 1998.

[26] G. Brose. JacORB: Implementation and design of a Java ORB. In *Proceedings of the International Working Conference on Distributed Applications and Interoperable Systems (DAIS'97)*. Chapman & Hall, Septembre 1997.
`www.inf.fu-berlin.de/~brose/jacorb/ftp/doc/dais97.ps.gz`.

[27] E. Bruneton, T. Coupaye, M. Leclercq, V. Quema, and J.-B. Stefani. An open component model and its support in Java. In *Proceedings of the 7th International Symposium on Component-Based Software Engineering (CBSE-7)*, volume 3054 of *Lecture Notes in Computer Science*, pages 7–22. Springer, Mai 2004.

[28] E. Bruneton, T. Coupaye, and J.-B. Stefani. *The Fractal Component Model*. ObjectWeb, Février 2004. Version 2.0.3.
`fractal.objectweb.org/specification/index.html`.

[29] M. Buchi and W. Weck. Generic wrappers. In *Proceedings of the 14th European Conference on Object-Oriented Programming (ECOOP'00)*, volume 1850 of *Lecture Notes in Computer Science*, pages 201–225. Springer, Juin 2000.

[30] J. Bézivin, F. Jouault, and P. Valduriez. First experiments with a ModelWeaver. In *OOPSLA & GPCE Workshop on Best Practices for Model Driven Software Development*, 2004.

[31] R. Campbell, N. Islam, D. Raila, and P. Madany. Designing and implementing Choices: An object-oriented system in C++. *Communications of the ACM*, 36(9):117–126, Septembre 1993.

[32] O. Caron, B. Carré, A. Muller, and G. Vanwormhoudt. A framework for supporting views in component oriented information systems. In *Proceedings of the International Conference on Object-Oriented Information Systems (OOIS'03)*, Septembre 2003.

[33] K. Chandy and L. Lamport. Distributed snapshots: Determining global states of distributed systems. *ACM Transactions on Computer Systems*, 3(1):63–75, Février 1985.

[34] C. Chase and V. Garg. Detection of global predicates: Techniques and their limitations. *Distributed Computing*, 11, 1998.

[35] S. Chiba. A metaobject protocol for C++. In *Proceedings of the 10th Annual Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA'95)*, volume 30 of *SIGPLAN Notices*, pages 285–299. ACM Press, Octobre 1995.

[36] S. Chiba. Load-time structural reflection in Java. In *Proceedings of the 14th European Conference on Object-Oriented Programming (ECOOP'00)*, volume 1850 of *Lecture Notes in Computer Science*, pages 313–336. Springer, Juin 2000.

[37] S. Chiba and K. Nakagawa. Josh: An open AspectJ-like language. In *Proceedings of the 3rd International Conference on Aspect-Oriented Software Development (AOSD'04)*, pages 102–111. ACM Press, 2004.

[38] S. Chiba, Y. Sato, and M. Tatsubori. Using HotSwap for implementing dynamic AOP systems. In *Workshop on Advancing the State-of-the-Art in Run-Time Inspection at ECOOP'03*, Juillet 2003.
`www.st.informatik.tu-darmstadt.de/pages/workshops/ASARTI03/index.html`.

[39] M. Clarke, G. Blair, G. Coulson, and N. Parlavantzas. An efficient component model for the construction of adaptive middleware. In *Proceedings of Middleware'01*, 2001.

[40] A. Colyer and A. Clement. Large-scale AOSD for middleware. In *Proceedings of the 3rd International Conference on Aspect-Oriented Software Development (AOSD'04)*, pages 56–65. ACM Press, 2004.

[41] C. Consel and R. Marlet. Architecturing software using a methodology for language development. In *Proceedings of the 10th International Symposium on Programming Language Implementation and Logic Programming*, volume 1490 of *Lecture Notes in Computer Science*, pages 170–194. Springer-Verlag, Septembre 1998.

[42] G. Coulson, G. Blair, M. Clarke, and N. Parlavantzas. The design of a configurable and reconfigurable middleware platform. *Distributed Computing*, 15(2):109–126, 2002.

[43] M. Dahm. Byte code engineering. In *Proceedings of JIT'99*, 1999.
`bcel.sourceforge.net`.

[44] P.-C. David. Développement de composants Fractal adaptatifs : un langage dédié à l'aspect d'adaptation. Thèse de Doctorat de l'École des Mines de Nantes, Juillet 2005.

[45] M. Desertot. JOnAS 5 and dynamic services with OSGi. ObjectWeb Architecture Meeting, Juin 2005.
`www.objectweb.org/phorum/read.php?f=16&i=226&t=226`.

[46] J. Dowling and V. Cahill. The K-Component architecture meta-model for self-adaptative software. In *Proceedings of Reflection'01*, volume 2192 of *Lecture Notes in Computer Science*, pages 81–88. Springer-Verlag, Septembre 2001.

[47] L. Duchien and E. Jeury. Observation in CORBA Java applications. In *Proceedings of the Session on Coordination in Parallel and Distributed Applications and Activities at PDPTA'99*, Juin 1999.

[48] L. Duchien and L. Seinturier. Reflective observation of CORBA applications. In *Proceedings of 11th IASTED International Conference on Parallel and Distributed Computing and Systems (PDCS'99)*, volume 1, pages 311–316. ACTA Press, Novembre 1999.

[49] L. Duchien and L. Seinturier. Observation of distributed computations: A reflective approach for CORBA. *International Journal of Parallel and Distributed Systems and Networks*, 4(1):17–25, 2001.

[50] B. Dumant, F. Horn, F. Dang Tran, and J.-B. Stefani. Jonathan: an open distributed processing environment in Java. In *Proceedings of Middleware'98*, 1998.
`jonathan.objectweb.org`.

[51] J.-C. Fabre. Tolérance aux fautes par protocole à métaobjets. *L'Objet*, 3(1):7–27, 1997.

[52] J. Fabry and T. Cleenewerck. Aspect-oriented domain specific languages for advanced transaction management. In *Proceedings of 7th International Conference on Enterprise Information Systems (ICEIS'05)*, 2005.

[53] H. Fakih and N. Bouraqadi. Les aspects et les composants logiciels. In *1ère Journée Francophone sur Développement de Logiels Par Aspects (JFDLPA'04)*, Septembre 2004.

[54] J.-P. Fassino, J.-B. Stefani, J. Lawall, and G. Muller. Think: A software framework for component-based operating system kernels. In *Proceedings of the USENIX Annual Technical Conference*, pages 73–86, Juin 2002.
`www.usenix.org/events/usenix02/full_papers/fassino/fassino_html/index.html`.

[55] J. Ferber. Computational reflection in class based object oriented languages. In *Proceedings of the 4th Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA'89)*, volume 24 of *SIGPLAN Notices*, pages 317–326. ACM Press, Octobre 1989.

[56] R. Filman and D. Friedman. *Aspect-Oriented Software Development*, chapter Aspect-Oriented Programming is Quantification and Obliouvness, pages 97–122. Addison-Wesley, Septembre 2004. ISBN 0-321-21976-7.

[57] R. Gabriel, M. Devos, B. Foote, G. Steele, and J. Noble. Objects have failed, Novembre 2002. Object-Oriented Programming, Systems, Languages and Applications (OOPSLA'02). Seattle, USA.

[58] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[59] B. Garbatino, R. Guerraoui, and K. Mazouni. Implementation of the GARF replicated objects platform. *Distributed System Engineering*, 2:14–27, Février 1995.

[60] V. Garg. Observation and control for debugging distributed computations. In *Proceedings of the 3rd International Workshop on Automated and Algorithmic Debugging (AADEBUG'97)*, 1997.

[61] D. Garlan, R. Monroe, and D. Wile. Acme: Architectural description of component-based systems. In *Foundations of Component-Based Systems*, pages 47–68. Cambridge University Press, 2000.

[62] M. Golm and J. Kleinöder. MetaXa: A platform for adaptable operating system mechanisms. In *Workshop on Object Orientation and Operating Systems at ECOOP'97*, Juin 1997.

[63] M. Golm and J. Kleinöder. MetaXa: A platform for adaptable operating system mechanisms. Technical Report TR-I4-97-10, Computer Science Department, Friedrich Alexander University, Erlangen-Nürnberg, Germany, Avril 1997.

[64] C. Gransart, P. Merle, and J. Geib. GoodeWatch: Supervision of CORBA applications. In *Workshop on Object-Orientation and Operating Systems at ECOOP'99*, Juin 1999.

[65] A. Hachichi, C. Martin, G. Thomas, B. Folliot, and S. Patarin. A generic language for dynamic adaptation. In *Proceedings of the 11th International Euro-Par Conference (Euro-Par 2005)*, pages 40–49, Août 2005.

[66] J. Hannemann and G. Kiczales. Design pattern implementation in Java and AspectJ. In *Proceedings of the 17th Conference on Object-Oriented Programming: Systems, Languages and Applications (OOPSLA'02)*, volume 37 of *SIGPLAN Notices*, pages 161–173. ACM Press, Octobre 2002.

[67] W. Harrison and H. Ossher. Subject-oriented programming (A critique of pure objects). In *Proceedings of OOPSLA'93*, volume 28 of *SIGPLAN Notices*, pages 411–428. ACM Press, Octobre 1993.

[68] D. Hart, E. Kraemer, and D. Roman. Interactive visual exploration of distributed computations. In *Proceedings of the 11th International Parallel Processing Symposium*, 1997.

[69] G. Heineman and W. Councill. *Component-based Software Engineering*. Addison-Wesley, 2001.

[70] W. Ho, J.-M. Jézéquel, F. Pennaneac'h, and N. Plouzeau. A toolkit for weaving aspect oriented UML designs. In *Proceedings of the 1st International Conference on Aspect-Oriented Software Development (AOSD'02)*, pages 99–105. ACM Press, 2002.

[71] F. Horn, F. Delpiano, and B. Dumant. Kilim 2 tutorial, 2003. `kilim.objectweb.org/doc-k2/index.html`.

[72] IEC. *Utility Communications Specification Working Group. TASE.2 Services and Protocol*, 1996. Version 1996-08. IEC870-6-503. ICCP Inter-Control Centre Communications Protocol Version 6.1.

[73] Inprise. Inprise AppCenter. `www.inprise.com/appcenter`, 1999.

[74] Y. Ishikawa, A. Hori, M. Sato, M. Matsuda, J. Nolte, H. Tezuka, H. Konaka, M. Maeda, and K. Kubota. Designing and implementation of metalevel architecture in C++: MPC++ approach. In *Proceedings of Reflection'96*, 1996.

[75] ITU-T X.901, ISO 10746. *RM-ODP*, 1995.

[76] I. Jacobson. Use cases and aspects - working seamlessly together. *Journal of Object Technology*, 2(4):7–28, Juillet 2003.

[77] R. Johnson, J. Hoeller, A. Arendsen, T. Risberg, and C. Sampaleanu. *Professional Java Development with the Spring Framework*. Wiley, Juillet 2005.

[78] P. Kenens, S. Michiels, F. Matthijs, B. Robben, E. Truyen, B. Vanhaute, W. Joosen, and P. Verbaeten. An AOP case with static and dynamic aspects. In *Workshop on Aspect-Oriented Programming at ECOOP'98*, Juillet 1998. `trese.cs.utwente.nl/aop-ecoop99/aop98.html`.

[79] G. Kiczales. Aspect-oriented programming. *ACM Computing Surveys*, 28(4):154, Décembre 1996.

[80] G. Kiczales. Beyond the back box: Open implementation. *IEEE Software*, 13(1):8–11, Janvier 1996.

[81] G. Kiczales, J. des Rivieres, and D. Bobrow. *The Art of the Metaobject Protocol*. MIT Press, 1991.

[82] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. Griswold. Getting started with AspectJ. *Communications of the ACM*, 44(10):59–65, 2001.

[83] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. Griswold. An overview of AspectJ. In *Proceedings of the 15th European Conference on Object-Oriented Programming (ECOOP'01)*, volume 2072 of *Lecture Notes in Computer Science*, pages 327–353. Springer, Juin 2001.

[84] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J. Loingtier, and J. Irwin. Aspect-oriented programming. In *Proceedings of the 11th European Conference on Object-Oriented Programming (ECOOP'97)*, volume 1241 of *Lecture Notes in Computer Science*, pages 220–242. Springer, Juin 1997.

[85] U. Kulesza and D. Silva. Reengineering of the JaWS web server design using aspect-oriented programming. In *Workshop on Aspects and Dimensions of Concerns at ECOOP'00*, Juillet 2000.

[86] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, Juillet 1978.

[87] D. Lantim. *.Net.* Eyrolles, 2003.

[88] R. Lea, C. Jacquemot, and E. Pillevesse. COOL: System support for distributed object-oriented programming. *Communications of the ACM*, 36(9):37–46, Septembre 1993.

[89] T. Leblanc and J. Mellor-Crummey. Debugging parallel programs with instant replay. *IEEE Transactions on Computers*, 36(4):471–482, Avril 1987.

[90] M. Leclercq, V. Quéma, and J.-B. Stefani. DREAM: a component framework for the construction of resource-aware, reconfigurable MOMs. In *Proceedings of the 3rd Workshop on Reflective and Adaptative Middleware (Middleware'04)*, Octobre 2004.

[91] M. Leclercq, V. Quéma, and J.-B. Stefani. DREAM: a component framework for the construction of resource-aware, configurable middleware. *IEEE Distributed Systems Online*, 6(9), 2005.

[92] T. Ledoux. Réflexion dans les systèmes répartis : application à CORBA et Smalltalk. Thèse de Doctorat de l'École des Mines de Nantes, Mars 1998.

[93] T. Ledoux. OpenCorba: A reflective open broker. In *Procedings of Reflection'99*, volume 1964 of *Lecture Notes in Computer Science*, pages 197–214. Springer, Juillet 1999.

[94] F. Legond-Aubry. Un modèle d'assemblage de composants par contrat et programmation orientée aspect. Thèse de Doctorat du Conservatoire National des Arts et Métiers, Paris, Juillet 2005.

[95] F. Legond-Aubry, G. Florin, and L. Seinturier. An AOP layer to abstract programming with distributed components. In *International Workshop on Aspect-Oriented Software Development (WAOSD'04)*, Septembre 2004.

[96] K. Li and P. Hudak. Memory coherence in shared virtual memory systems. *ACM Transactions on Computer Systems (TCOS)*, 7(4):321–359, Novembre 1989.

[97] F. Loiret. Composants logiciels & programmation par aspects - application aux applications réparties. Mémoire de DEA SIR, Université Pierre & Marie Curie, Paris, Septembre 2003. `www.scanx.org/acpoa/`.

[98] F. Loiret, L. Seinturier, and E. Gressier-Soudan. Fractal, Kilim, JAC : une expérience comparative. In *Journées Composants 2004 (JC'04)*, Mars 2004. `www.lifl.fr/jc2004/articles/loiret-seinturier-gressier-soudan.ps`.

[99] C. Lopes. AOP: A historical perspective. Technical Report UCI-ISR-02-5, Institute for Software Research, University of California, Irvine, Décembre 2002. `www.isr.uci.edu/tech_reports/UCI-ISR-02-5.pdf`.

[100] C. Lopes. *Aspect-Oriented Software Development*, chapter AOP: A Historical Perspective, pages 97–122. Addison-Wesley, Septembre 2004. ISBN 0-321-21976-7.

[101] C. Lopes and G. Kiczales. D: A language framework for distributed programming. Technical Report SPL97-010 P9710047, Xerox Palo Alto Research Center, Février 1997. `www.parc.xerox.com/csl/groups/sda/publications/papers/PARC-AOP-D97/for-web.pdf`.

[102] D. Luckham and J. Vera. An event-based architecture definition language. *IEEE Transactions on Software Engineering*, 21(9):714–734, Septembre 1995.

[103] P. Maes. Concepts and experiments in computational reflection. In *Proceedings of the 2nd Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA'87)*, volume 22 of *SIGPLAN Notices*, pages 147–155. ACM Press, Décembre 1987.

[104] J. Magee and J. Kramer. Dynamic structure in software architectures. In *Proceedings of the 4th ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 3–14, Juin 1996.

[105] R. Marvie, P. Merle, J.-M. Geib, and M. Vadet. OpenCCM: une plate-forme pour composants CORBA. In *Actes de la seconde conférence française sur les systèmes d'exploitation (CFSE-2)*, Avril 2001.

[106] R. Marvie and M.-C. Pellegrini. Modèles de composants, un état de l'art. *L'Objet*, 8(3):61–89, 2002.

[107] Introduction to MAScOTTE, Esprit project 20804. White paper, Mai 1997.

[108] H. Masuhara, S. Matsuoka, T. Watanabe, and A. Yonezawa. Object-oriented concurrent reflective languages can be implemented efficiently. In *Proceedings of the 7th Conference on Object-Oriented Programming: Systems, Languages and Applications (OOPSLA'92)*, volume 27 of *SIGPLAN Notices*, pages 127–147. ACM Press, Octobre 1992.

[109] J. McAffer. Meta-level programming with CodA. In *Proceedings of the 9th European Conference on Object-Oriented Programming (ECOOP'95)*, volume 952 of *LNCS*, pages 190–214. Springer-Verlag, Août 1995.

[110] S. McDirmid and W. Hsieh. Aspect-oriented programming with Jiazzi. In *Proceedings of the 2nd International Conference on Aspect-Oriented Software Development (AOSD'03)*, pages 70–79. ACM Press, 2003.

[111] D. McIlroy. Mass produced software component. Technical report, Report on the NATO Software Engineering Conference, 1968.

[112] N. Medvidovic and R. Taylor. A classification and comparison framework for software architecture description languages. *IEEE Transactions on Software Engineering*, 26, Mai 2000.

[113] A. Mendhekar, G. Kiczales, and J. Lamping. RG: A case study for aspect-oriented programming. Technical Report SPL-97-009, Xerox Palo Alto Research Center, 1997.

[114] P. Merle, D. Sevilla Ruiz, H. Böhme, S. Leblanc, M. Vadet, T. Ritter, and J. Scott Evans. *CORBA Component Model Tutorial*. OMG, 2002. Document ccm/02-06-01. `www.omg.org/cgi-bin/doc?ccm/2002-06-01`.

[115] B. Meyer. Applying desing by contract. *IEEE Computer*, 25(10):40–52, Octobre 1992.

[116] B. Miller and D. Choi. Breakpoints and halting in distributed programs. In *Proceedings of the 8th International Conference on Distributed Computing Systems*, pages 316–323, 1988.

[117] Emerging technologies and their impact - ten technologies that will change the world, Janvier 2001. Repris par Le Monde Interactif du 28/02/01. `www.techreview.com/articles/mag_toc_jan01.asp`.

[118] M. Nishizawa, S. Chiba, and M. Tatsubori. Remote pointcut - a language construct for distributed AOP. In *Proceedings of the 3rd International Conference on Aspect-Oriented Software Development (AOSD'04)*, pages 7–15. ACM Press, 2004.

[119] F. Ogel, B. Folliot, and I. Piumarta. On reflexive and dynamically adaptable environments for distributed computing. In *3rd International Workshop on Distributed Auto-Adaptive and Reconfigurable Systems at ICDCS'03*, Mai 2003.

[120] F. Ogel, G. Thomas, A. Galland, and B. Folliot. MVV : une plate-forme à composants dynamiquement reconfigurables. *Techniques et Sciences Informatiques*, 23(10):1269–1299, 2004.

[121] H. Okamura and Y. Ishikawa. Object location control using meta-level programming. In *Proceedings of the 8th European Conference in Object-Oriented Programming (ECOOP'94)*, volume 821 of *Lecture Notes in Computer Science*, pages 299–319. Springer-Verlag, Juillet 1994.

[122] OSGi Alliance. *OSGi Technical Whitepaper*, Juin 2004. Revision 3.0. `www.osgi.org`.

[123] H. Ossher, K. Kaplan, W. Harrison, A. Matz, and V. Kruskal. Subject-oriented composition rules. In *Proceedings of the 10th Annual Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA'95)*, volume 30 of *SIGPLAN Notices*, pages 235–250. ACM Press, 1995.

[124] H. Ossher and P. Tarr. Multi-dimensional separation of concerns and the hyperspace approach. In *Software Architectures and Component Technology*. Kluwer Academic Publishers, 2000.

[125] H. Ossher and P. Tarr. Using multidimensional separation of concerns to (re)shape evolving software. *Communications of the ACM*, 44(10):43–49, 2001. Special issue on Aspect-Oriented Programming.

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

[127] L. Pautet. Intergiciels schizophrènes : une solution à l'interopérabilité entre modèles de répartition. Habilitation à diriger des recherches, Université Pierre & Marie Curie, Paris, Décembre 2001.

[128] R. Pawlak. Contruction d'applications réparties orientées aspect. Thèse de Doctorat du Conservatoire National des Arts et Métiers, Paris, Décembre 2002.

[129] R. Pawlak, L. Duchien, G. Florin, F. Legond-Aubry, L. Seinturier, and L. Martelli. An UML notation for aspect-oriented software design. In *Workshop on Aspect-Oriented Modeling with UML at AOSD'02*, Avril 2002.

[130] R. Pawlak, L. Duchien, G. Florin, L. Martelli, and L. Seinturier. Distributed separation of concerns with aspects components. In *Proceedings of the European Conference on Technology of Object-Oriented Languages and Systems (TOOLS Europe 2000)*. Prentice Hall, 2000.

[131] R. Pawlak, L. Duchien, and L. Seinturier. Ensuring safe around advice composition. In *Proceedings of the 7th IFIP International Conference on Formal Methods for Open Object-Based Distributed Systems (FMOODS'05)*, pages 163–178, Juin 2005.

[132] R. Pawlak, L. Duchien, L. Seinturier, G. Florin, F. Legond-Aubry, and L. Martelli. *Aspect-Oriented Software Development*, chapter JAC: An Aspect-Based Distributed Dynamic Framework, pages 343–369. Addison-Wesley, Septembre 2004. ISBN 0-321-21976-7.

[133] R. Pawlak, L. Seinturier, L. Duchien, and G. Florin. JAC: A flexible solution for aspect-oriented programming in Java. In *Proceedings of Reflection'01*, volume 2192 of *Lecture Notes in Computer Science*, pages 1–24. Springer, Septembre 2001.

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

[135] R. Pawlak and H. Younessi. On getting use cases and aspects to work together. *Journal of Object Technology*, 3(1):15–26, Janvier 2004.

[136] F. Peschanski. Comet: A component-based reflective architecture for distributed programming. In *1st Workshop on Object-Orientation for Software Engineering at OOPSLA'99*, Novembre 1999.

[137] N. Pessemier, O. Barais, L. Seinturier, T. Coupaye, and L. 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*, Juillet 2005. `wcat05.unex.es/wcat05/index.htm`.

[138] N. Pessemier, L. Seinturier, and L. Duchien. Components, adl & aop: Towards a common approach. In *Workshop on Reflection, AOP and Meta-Data for Software Evolution (RAM-SE) at ECOOP'04*, Juin 2004.
`www.disi.unige.it/person/CazzolaW/RAM-SE04.html`.

[139] M. Pinto, L. Fuentes, M. Fayad, and J. Troya. Separation of coordination in a dynamic aspect oriented framework. In *Proceedings of the 1st International Conference on Aspect-Oriented Software Development (AOSD'02)*, pages 134–140. ACM Press, 2002.

[140] P. Placide, L. Duchien, G. Florin, and L. Seinturier. A consistent global state algorithm to debug distributed object-oriented applications. In *Proceedings of AADEBUG'95*, Mai 1995.

[141] F. Plasil, D. Balek, and R. Janecek. SOFA/DCUP: Architecture for component trading and dynamic updating. In *Proceedings of the 4th IEEE International Conference on Configurable Distributed Systems (ICCDS'98)*, Mai 1998.

[142] R. Pratap, R. Cytron, D. Sharp, and E. Pla. Transport layer abstraction in event channels for embedded systems. In *Proceedings of the ACM SIGPLAN Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES'03)*, Juin 2003.

[143] E. Putrycz and G. Bernard. Using aspect oriented programming to build a portable load balancing service. In *Proceedings of the International Workshop on Aspect Oriented Programming for Distributed Computing Systems at ICDCS'02*, Juillet 2002.

[144] I. Rammer and M. Szpuszta. *Advanced .NET Remoting*. APress, 2nd edition, Mars 2005.

[145] A. Rashid. *Aspect-Oriented Database Systems*. Springer-Verlag, 2004.

[146] A. Rashid and R. Chitchyan. Persistence as an aspect. In *Proceedings of the 2nd International Conference on Aspect-Oriented Software Development (AOSD'03)*, pages 120–129. ACM Press, 2003.

[147] B. Redmond and V. Cahill. Iguana/J: Towards a dynamic and efficient reflective architecture for Java. In *Workshop on Reflective Object-Oriented Programming and Systems at ECOOP'00*, Juillet 2000.
`www.disi.unige.it/RMA2000.html`.

[148] J.-P. Retaillé, R. Pawlak, and L. Seinturier. *La programmation orientée aspect pour Java/J2EE*. Eyrolles, Mai 2004. ISBN 2-212-11408-7.

[149] J.-P. Retaillé, R. Pawlak, and L. Seinturier. *Foundations of AOP for J2EE Development*. APress, Septembre 2005. ISBN 1-59059-507-6.

[150] B. Robben, W. Joosen, F. Matthijs, B. Vanhaute, and P. Verbaeten. A metaobject protocol for Correlate. In *Workshop on Reflective Object-Oriented Programming at ECOOP'98*, Juillet 1998.

[151] R. Rouvoy. Canevas pour le transactionnel dans les plates-formes à composants. Mémoire de DEA, Université Lille 1, Juin 2003.
`www.lifl.fr/jacquard/modules/biborb/bibs/jacquard/papers/rouvoy-dea-03.pdf`.

[152] D. Sabbah. AOSD - from promise to reality. 3rd International Conference on Aspect-Oriented Software Development (AOSD'04), Mars 2004.
`aosd.net/2004/archive/AOSD-FromPromiseToReality.ppt`.

[153] D. Schmidt and C. Cleeland. Applying a pattern language to develop application-level gateways. In *Design Patterns in Communications Software*, pages 315–355. Cambridge University Press, 2001.
`www.cs.wustl.edu/~schmidt/PDF/ORB-patterns.pdf`.

[154] R. Schwarz and F. Mattern. Detecting causal relationships in distributed computations: In search of the holy grail. Technical Report SFB 124 - 15/92, University of Kaiserslautern, Décembre 1992.

[155] N. Schärli, S. Ducasse, O. Nierstrasz, and A. Black. Traits: Composable units of behaviour. In *Proceedings of the 17th European Conference on Object-Oriented Programming (ECOOP'03)*, volume 2743 of *Lecture Notes in Computer Science*, pages 248–274. Springer-Verlag, Juillet 2003.

[156] L. Seinturier. Intégration de liaisons ATM dans l'ORB CORBA Jonathan. Technical Report NT/CNET/6125, CNET, Janvier 1999.
`www-src.lip6.fr/homepages/Lionel.Seinturier/cnet/rapport.html`.

[157] L. Seinturier, L. Duchien, and G. Florin. A meta-object protocol for distributed OO applications. In *Proceedings of TOOLS 23*, Août 1997.

[158] L. Seinturier, L. Duchien, and G. Florin. CAOLAC : un protocole à méta-objets pour la synchronisation d'objets concurrents. *L'Objet*, 4(3):241–272, Septembre 1998.

[159] L. Seinturier, N. Pessemier, and T. Coupaye. AOKell: An aspect-oriented implementation of the Fractal specifications. Objectweb Fractal Workshop, Grenoble, France, Juin 2005.
`www.lifl.fr/~seinturi/aokell/javadoc/overview.html`.

[160] M. Shapiro. A binding protocol for distributed shared objects. In *Proceedings of the International Conference on Distributed Computing Systems (ICDCS'94)*, Juin 1994.

[161] M. Shapiro, Y. Gourhant, S. Habert, L. Mosseri, M. Ruffin, and C. Valot. SOS: An object-oriented operating system - assessment and perspectives. *Computing Systems*, 4(2):287–338, Décembre 1989.

[162] J. Siegel. *CORBA 3 Fundamentals and Programming.* Wiley, 2nd edition, 2000.

[163] B. Smith. Reflection and semantics in a procedural language. Technical Report 272, Laboratory for Computer Science, MIT, 1982.

[164] B. Smith. Reflection and semantics in Lisp. In *Proceedings of ACM Symposium on Principles of Programming Languages*, pages 23–35, 1984.

[165] S. Soares, E. Laureano, and P. Borba. Implementing distribution and persistence with AspectJ. In *Proceedings of the 17th Conference on Object-Oriented Programming: Systems, Languages and Applications (OOPSLA'02)*, volume 37 of *SIGPLAN Notices*, pages 174–190. ACM Press, Octobre 2002.

[166] D. Suvée, W. Vanderperren, and V. Jonckers. 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, 2003.

[167] D. Suvée, W. Vanderperren, and V. Jonckers. FuseJ: An architectural description language for unifying aspects and components. In *Workshop Software-engineering Properties of Languages and Aspect Technologies (SPLAT) at AOSD'05*, 2005. `ssel.vub.ac.be/Members/dsuvee/papers/splatsuvee2.pdf`.

[168] C. Szyperski. *Component Software - Beyond Object-Oriented Programming*. Addison-Wesley, 2nd edition, 2002.

[169] M. Ségura-Devillechaise, J.-M. Menaud, G. Muller, and J. Lawall. Web cache prefetching as an aspect: Towards a dynamic-weaving based solution. In *Proceedings of the 2nd International Conference on Aspect-Oriented Software Development (AOSD'03)*, pages 110–119. ACM Press, 2003.

[170] E. Tanter, N. Bouraqadi, and J. Noyé. Reflex - towards an open reflective extension of Java. In *Procedings of Reflection'01*, volume 2192 of *Lecture Notes in Computer Science*, pages 25–43. Springer, Septembre 2001.

[171] M. Tatsubori and S. Chiba. Programming support of design patterns with compile-time reflection. In *Workshop on Reflective Programming in C++ and Java at OOPSLA'98*, Juillet 1998.

[172] L. Teboul. Programmation par aspect appliquée à une messagerie industrielle. Mémoire de DEA SIR, Université Pierre & Marie Curie, Paris, Septembre 2002.

[173] L. Teboul, R. Pawlak, L. Seinturier, E. Gressier-Soudan, and E. Becquet. AspectTAZ: A new approach based on aspect-oriented programming for object-oriented industrial messaging services design. In *Proceedings of the 4th IEEE International Workshop on Factory Communication Systems (WFCS'02)*, Août 2002.

[174] P. Tessier, S. Gérard, C. Mraidha, F. Terrier, and J.-M. Geib. A component-based methodology for embedded system prototyping. In *Proceedings of the 14th IEEE International Workshop on Rapid System Prototyping (RSP'03)*, pages 9–15. IEEE Computer Society Press, Juin 2003.

[175] K. Thini. Service de tolérance aux pannes pour les composants EJB. Mémoire de DEA SIR, Université Pierre & Marie Curie, Paris, Septembre 2002.

[176] C. Treanor. IIOP protocol analyser. `www-inf.int-evry.fr/~defude/analyseur-iiop.html`, 1999.

[177] M. Vadet. Un modèle de services logiciels pour la spécialisation des intergiciels à composants. Thèse de Doctorat de l'Université Lille 1, Novembre 2004.

[178] A. van Deursen, P. Klint, and J. Visser. Domain-specific languages: An annotated bibliography. *SIGPLAN Notices*, 35(6):26–36, 2000.

[179] T. Watanabe and A. Yonezawa. Reflection in an object-oriented concurrent language. In *Proceedings of OOPSLA'88*, volume 23 of *SIGPLAN Notices*. ACM Press, Septembre 1988.

[180] I. Welch and R. Stroud. From Dalang to Kava - the evolution of a reflective Java extension. In *Procedings of Reflection'99*, volume 1964 of *Lecture Notes in Computer Science*, pages 2–21. Springer, Juillet 1999.

[181] Y. Yokote. The Apertos reflective operating system: The concept and its implementation. In *Proceedings of the 7th Conference on Object-Oriented Programming: Systems, Languages and Applications (OOPSLA'92)*, volume 27 of *SIGPLAN Notices*, pages 414–434. ACM Press, Octobre 1992.

[182] G. Zelesnik. The UniCon language reference manual. Technical report, Carnegie Mellon University, 1996.
`www-2.cs.cmu.edu/afs/cs/project/vit/www/unicon/`.

[183] C. Zhang and H.-A. Jacobsen. Quantifying aspects in middleware platforms. In *Proceedings of the 2nd International Conference on Aspect-Oriented Software Development (AOSD'03)*, pages 130–139. ACM Press, 2003.

Index

Annexe A

Liste sélective de publications et d'encadrements

Cette annexe présente chapitre par chapitre, une liste sélective de publications majeures et d'encadrements.

Une liste complète de publications est disponible sur la page web :
`www-src.lip6.fr/homepages/Lionel.Seinturier/publis.html`

A.1 Réflexivité

Publications majeures

1. L. Duchien and L. Seinturier. Observation of distributed computations: A reflective approach for CORBA. *International Journal of Parallel and Distributed Systems and Networks*, 4(1):17–25, 2001.

2. L. Duchien and L. Seinturier. Reflective observation of CORBA applications. In *Proceedings of 11th IASTED International Conference on Parallel and Distributed Computing and Systems (PDCS'99)*, volume 1, pages 311–316. ACTA Press, Novembre 1999.

3. P. Placide, L. Duchien, G. Florin, and L. Seinturier. A consistent global state algorithm to debug distributed object-oriented applications. In *Proceedings of AADEBUG'95*, Mai 1995.

Encadrements

DEA

1. En 2001, j'ai encadré (avec P. Sens) le stage de DEA de Kamal Thini. Ce stage s'est déroulé dans le contexte du projet RNTL IMPACT (projet de type plate-forme visant à développer des briques logicielles de base pour le développement des intergiciels). K. Thini a développé un service de réplication de componsants EJB pour le serveur J2EE JOnAS.

A.2 Programmation orientée aspect

Publications majeures

Les quatre publications les plus significatives sur JAC et l'AOP sont les suivantes (voir CV pour une liste complète). Les deux premières références, conférence Reflection 2001 et journal Software Practise and Experience en 2004, sont les deux articles de référence qui présentent la plate-forme JAC.

Par la suite, l'ouvrage paru en 2004 chez Eyrolles présente un panorama de l'AOP. Il contient une présentation générale du domaine et de ses concepts, une présentation de AspectJ, JAC, JBoss AOP, AspectWerkz et une comparaison de ces outils. L'ouvrage se poursuit par des exemples d'utilisation des aspects pour la programmation de patrons de conception (*designs pattern*), de contrat et pour l'administration des applications. Finalement, l'étude de cas de l'aspectisation d'une application J2EE est présentée. En septembre 2005, une version en anglais mise à jour (notamment avec les nouveautés de la version 5 d'AspectJ) et complétée (avec une présentation de Spring AOP) est paru chez APress.

1. R. Pawlak, L. Seinturier, L. Duchien, and G. Florin. JAC: A flexible solution for aspect-oriented programming in Java. In *Proceedings of Reflection'01*, volume 2192 of *Lecture Notes in Computer Science*, pages 1–24. Springer, Septembre 2001.

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

3. J.-P. Retaillé, R. Pawlak, and L. Seinturier. *La programmation orientée aspect pour Java/ J2EE*. Eyrolles, Mai 2004. ISBN 2-212-11408-7.

4. J.-P. Retaillé, R. Pawlak, and L. Seinturier. *Foundations of AOP for J2EE Development*. APress, 2005. ISBN 1-59059-507-6.

Encadrements

Thèses

1. De 1998 à 2002, j'ai participé avec G. Florin et L. Duchien à l'encadrement de la thèse de Renaud Pawlak (directeur G. Florin) qui a permis de définir la plate-forme JAC. Cette thèse a montré que les aspects peuvent être utilisés pour construire un intergiciel et a contribué au développement de l'AOP en étant au niveau international une des toutes premières plates-formes AOP dynamiques. Elle continue à faire partie des 5 plates-formes les plus référencées du domaine.

2. De 2001 à 2005 (thèse soutenue le 11 juillet 2005), j'ai encadré la thèse de Fabrice Legond-Aubry (directeur G. Florin). Cette thèse porte sur l'intégration de services dans les applications à base de composants (J2EE et CCM) à l'aide de l'AOP. Elle a permis de montrer que l'AOP est une technique adaptée au développement de services tels que la vérification de contrats ou l'équilibrage de charge pour des applications à base de composants. L'AOP permet de réaliser cette intégration de façon transparente, apportant ainsi un gain en terme de souplesse de développement et de modularité du code.

3. Depuis janvier 2004, j'encadre la thèse de Dolorès Diaz (directrice L. Duchien). Cette thèse s'effectue dans le cadre d'une convention CIFRE avec la société de services Norsys (responsable scientifique P. Flament). Le but de la thèse est de mettre en place une méthode pour gérer l'évolution des logiciels. Il s'agit de pouvoir assurer pour chaque fonctionnalité d'une application, d'une part une traçabilité verticale entre les différentes phases d'un processus de développement, et d'autre part une traçabilité temporelle en fonction de l'évolution des besoins à prendre en compte pour la fonctionnalité. Ce travail doit donner lieu au développement d'une méthode et à sa mise en œuvre à l'aide de l'AOP dans un IDE.

DEA

1. En 2000, Grégory Haïk. Ce stage a porté sur la définition d'un langage d'aspect pour l'observation d'applications réparties. Le candidat a poursuivi son parcours en thèse (Université Paris 6, directeur J.-P. Briot).
2. En 2001, Thibault Garcia-Fernandez (encadrement avec B. Folliot). Ce stage a porté sur la définition d'aspect de bas niveau pour la gestion de drivers. Le candidat a poursuivi son parcours en thèse (Université de Nantes).
3. En 2002, Luc Teboul (encadrement avec E. Gressier-Soudan). Ce stage a porté sur la définition et l'implémentation d'un aspect de communication en mode message utilisant le protocole réseau de messagerie industrielle TASE.2 pour la plate-forme JAC.

A.3 Composants et ADL

Publications majeures

1. F. Loiret, L. Seinturier, and E. Gressier-Soudan. Fractal, Kilim, JAC : une expérience comparative. In *Journées Composants 2004 (JC'04)*, Mars 2004. `www.lifl.fr/jc2004/articles/loiret-seinturier-gressier-soudan.ps`.
2. N. Pessemier, O. Barais, L. Seinturier, T. Coupaye, and L. 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*, Juillet 2005. `wcat05.unex.es/wcat05/index.htm`.
3. N. Pessemier, L. Seinturier, and L. Duchien. Components, adl & aop: Towards a Common Approach. In *Workshop on Reflection, AOP and Meta-Data for Software Evolution (RAM-SE) at ECOOP'04*, Juin 2004. `www.disi.unige.it/person/CazzolaW/RAM-SE04.html`.

Encadrements

Thèse

1. Depuis octobre 2004, j'encadre la thèse de Nicolas Pessemier (directrice L. Duchien). Cette thèse s'effectue dans le cadre d'un contrat entre l'INRIA Futurs et France Telecom R&D. Le but de la thèse est de travailler sur l'unification des concepts d'aspect et de composant autour du modèle Fractal.

2. Depuis octobre 2004, j'encadre la thèse de Frédéric Loiret (directrice L. Duchien). Cette thèse se déroule en collaboration avec le CEA Saclay (S. Gérard et D. Servat). Elle porte sur la définition d'un modèle de composants pour les applications temps-réel embarquées.

DEA

1. En 2003, Frédéric Loiret (encadrement avec E. Gressier-Soudan). Ce stage a porté sur l'implémentation d'une application de gestion de données répliquées avec JAC. Deux autres implémentations de la même application ont été réalisées avec les modèles de composant Fractal et Kilim. Une comparaison des trois implémentations a été réalisée et les résultats ont été publiés lors de la conférence nationale sur les composants logiciels JC 2004.
2. En 2004, Nicolas Pessemier. Ce stage a porté sur l'extension du modèle de composants Fractal avec des notions issues de l'AOP. Les résultats de ce stage ont été présentés en juin 2004 lors du workshop Reflection, AOP and Meta-Data for Software Evolution (RAM-SE) de la conférence ECOOP.