**UNIVERSITE SAAD DAHLEB DE BLIDA**

**Faculté des Sciences**
Département d'Informatique

# MEMOIRE DE MAGISTER

Spécialité : Systèmes d'informations et de connaissances

# IASA, UNE APPROCHE ARCHITECTURE LOGICIELLE ORIENTEE ASPECT

Par

**KHIDER Hadjer**

Devant le jury composé de :

| | | |
|---|---|---|
| N. Benblidia | Maître de conférences A, U.S.D., de Blida | Présidente |
| W. Hidouci | Maître de conférences A, E.S.I. Alger | Examinateur |
| D. Djenouri | Maitre de Recherche, C.E.R.I.S.T. Alger | Examinateur |
| D. Bennouar | Maître de conférences U.S.D., de Blida | Promoteur |

BLIDA, Juin 2012

**ملخص:**

تقنيات البرمجة و الطرائق المرتبطة بها قد عرفت تطورا كبيرا عبر تاريخ الإعلام الآلي مع تطور أنظمة البرمجيات, هذه الأنظمة أصبحت أكثر تعقيدا مع مرور الوقت. أ
البرمجة اعتمادا على القطع المركبة قد أثبتت نجاعتها في التحكم في مدى تعقيدات البرامج المطورة وأصبحت عامل رئيسي في نجاح تطوير المشاريع البرمجية عن طريق تسهيل صيانتها وتطور البرامج و السماح بتطوير أنظمة أكثر ضخامة وأكثر تعقيدا.
هذا النمط من البرمجة يعد بإعادة الاستخدام ولكنه واجه مشاكل التشتت ومزج الخصائص مما استدعى تطبيق ال AOP على القطع البرمجية لمعالجة هذه المشاكل.
البرمجة عن طريق فصل الخصائص الوظيفية عن الخصائص اللاوظيفية هو نموذج البرمجة الحديثة فهو امتداد للتقنيات البرمجية الحالية ولكن بشكل أكثر بساطة وأسهل للتغيير.
اليوم الجوانب اللاتقنية والقطع البرمجية هما نموذج واعد الذي يعزز من إعادة الاستخدام وتبسيط وتطوير البرامج حتى الآن, التنفيذ المتزامن لهذين النموذجان هو حقل بحث واستكشاف مهمش. حتى الآن لا يوجد نموذج للقطع المركبة يدعم صراحة فصل الجوانب اللاوظيفية عن الجوانب الوظيفية.
نقدم في هذه الأطروحة IASA-AOP, امتدادا للنموذج IASA المطور في المخبر LRDSI.
الهدف من هذا العمل هو تدعيم نموذج IASA بفصل الجوانب اللاوظيفية بجميع جوانبها.

كلمات مفتاح: البرمجيات المركبة, البرمجة الموجهة اللاوظيفية, القطع المركبة, المنافذ,القطع اللاوظيفية, نقاط الوصل, نقاط العمل,النسج.

## RESUME

Les techniques de programmation et les méthodologies liées ont fortement évoluées tout au long de l'histoire de l'informatique avec l'évolution des systèmes logiciels, ces systèmes ont en effet tendance à devenir de plus en plus complexes. La programmation à base de composants logiciels a prouvé ses intérêts dans la maitrise de la complexité des logiciels conçus et devenu un facteur critique dans la réussite de développement des projets logiciels en facilitant la maintenance et l'évolution du logiciel et autorisant le développement des systèmes volumineux en termes de taille mais aussi de complexité.

Ce style de programmation promet la réutilisation, mais est confronté aux problèmes de dispersion et de mélange de code représentant des propriétés transversales. L'application de la programmation par aspects (AOP) sur les composants logiciels permet de faire face à ces problèmes. La programmation dite par aspect permettant de gérer, de manière modulaire, ces préoccupations en les séparant du code de base.

La Programmation orienté aspect , un nouveau paradigme de la programmation étendant l'existant qui fait parties des techniques de programmation qui ont permis de simplifier l'écriture des programmes informatiques, en les rendant plus modulaire et plus faciles a faire évoluer.

Aujourd'hui, les Aspects et les composants logiciels sont deux paradigmes très prometteurs ; qui favorisent la réutilisation et simplifient le développement logiciel. A ce jour, la mise en œuvre simultanée de ces deux paradigmes reste un champ de recherche très faiblement explorée. A ce jour aucun modèle de composant ne supporte de manière explicite les aspects et plusieurs questions restent ouvertes. Parmi elles : Comment intégrer la représentation des aspects dans les composants logiciels ? Comment gérer les interactions et chevauchements entre aspects ?

Nous présentons dans ce mémoire IASA-AOP, une extension du modèle de composant IASA[1] définie au laboratoire LRDSI qui supporte la programmation par aspects. Cette extension consiste à doter l'approche IASA des composants orienté aspect et des ports orienté aspect.

L'objectif du travail est de faire supporte au modèle de composant IASA le concept d'aspect dans toute sa dimension : Une fois ce concept supporté, un architecte pourrait définir ses propres composants Aspect qu'il instancierait dans la partie contrôle d'un composant.



**Mots clés :** Architecture logicielle, Programmation Orienté Aspect, Composant, Port, Aspect, Point de jonction, Point d'action, Tissage, Advice.


[1] IASA : Integrated Architecture Software Approche développé au sein de laboratoire LRDSI, Blida

## ABSTRACT

The techniques of programming and methodologies strongly evolved throughout the history of data processing with the evolution of the software systems, these systems indeed tend to become increasingly complex. Component-Based Software Development proved its interests in the control of the complexity of the conceived software, and became a critical factor in the success of development of the software projects by facilitating the maintenance and the evolution of the software and authorizing the development of the bulky systems in terms of size but also of complexity.

This style of programming promises the re-use, but is confronted with the problems of *code scattering and tangling.* The application of Aspect-Oriented Programming on the software components makes it possible to face these problems. Programming called by aspect allowing managing, in a modular way, these concerns by separating them from the basic code.

Aspect-Oriented Programming, a new paradigm of the programming which made possible to simplify the writing of the programs data-processing, while making them more modular and easier has to make evolve.

Today, the software Aspects and components are two very promising paradigms which support the re-use and simplify the software development. To date, implementation the simultaneous of these two paradigms remains a field of research very slightly explored. To date no model of component supports in an explicit way the aspects and several questions remain open. Among them: How to integrate the representation of the aspects in the software components? How to manage the interactions and overlappings between aspects?

We present in this dissertation IASA-AOP, an extension of the model of component IASA defined in the laboratory LRDSI which supports the Aspect-Oriented Programming. This extension consists in equipping approach IASA with the aspect components and aspect ports.

The objective of work is to make supports to the model of component IASA the concept of aspect in its entire dimension: Once this concept supported, an architect could define his own Aspect components which it instantiated in the part controls of a component.



Key Words: Software architecture, Aspect-Oriented Programming, Component, Aspect, JoinPoint, PointCut, Weaving, Advice.

**REMERCIEMENTS**

Je voudrais tout d'abord remercier les membres du jury pour m'avoir fait l'honneur d'accepter de juger ce travail de thèse.

Je tiens à remercier très chaleureusement Dr. BENNOUAR DJAMEL sans qui rien de tout cela ne serait arrivé. J'ai énormément apprécié les années de travail sous votre direction. Merci pour votre soutien sans lequel je n'aurais jamais réussi à aller au bout, vos conseils toujours lumineux et votre patience. Merci aussi pour le temps que vous m'as consacré au jour le jour pendant ces années.

Je remercie tout particulièrement mes chers parents pour leurs patiences, encouragement et leurs soutiens précieux tout au long ma carrière.

Je souhaite remercier vivement mes collègues de l'école doctorale Nadjet Zerf, Bou Jabbour Karim et mon ami Mohamed Zouggar.

Enfin, par ces remerciements, je tiens à ne pas oublier les membres de l'école doctorale SIC particulièrement la présidente de l'école doctorale Mlle N. Benblidia Maitre de conférences, U. de Blida ainsi que Mme S. Oukid, Maitre de conférences, U. de Blida.

Merci a tous ceux qui de prés ou de loin m'ont aidé et soutenu pour que ce travail soit réalisé.

**TABLE DES MATIERES**

## LISTE DES ILLUSTRATIONS, GRAPHIQUES ET TABLEAUX

LISTE DES FIGURES

LISTE DES TABLEAUX

# INTRODUCTION

La séparation des préoccupations est un concept présent [7] depuis de nombreuses années dans l'ingénierie des logiciels. En effet, les différentes préoccupations des concepteurs apparaissent comme les motivations premières pour organiser et décomposer une application en un ensemble d'éléments compréhensibles et facilement manipulables La séparation en préoccupations apparait dans les différents étapes du cycle de vie du logiciel et sont donc de différents ordres. Il peut s'agir de préoccupations d'ordre fonctionnel (séparations des fonctions de l'application), technique (séparation des propriétés du logiciel système). Par ces séparations, le logiciel n'est plus abordé dans sa globalité, mais par partie. Cette approche réduit la complexité de conception, de réalisation, mais aussi de maintenance d'un logiciel et en améliore la compréhension, la réutilisation et l'´evolution.

En l'absence d'une approche dans laquelle les préoccupations ne sont pas séparées de manières très nette, nous serons face à un problème d'entrecroisement des préoccupations qui impliquera un certain nombre d'inconvénient au niveau d'un système :

- **Mauvaise traçabilité du code** : implanter simultanément dans un seul module plusieurs préoccupations rend difficile l'étude de la préoccupation
- **Plus faible réutilisation du code** : du fait de l'enchevêtrement de plusieurs besoins dans un même module, il est très difficile d'extraire les parties du code correspondant à un besoin pour les réutiliser dans d'autres modules entraînant de nouveau une diminution de la productivité à moyen terme.
- **Pauvre qualité du code** : en s'occupant de plusieurs préoccupations à la fois, il est très probable de ne pas accorder à une d'elles toute l'importance qu'elle mérite. On risque alors de produire un code de mauvaise qualité.
- **Difficulté à faire évoluer le code** : La modification d'une certaine préoccupation, est difficile lorsque le code correspondant est éparpillé dans une multitude de modules.

- **Evolution difficile** : une vue limitée, et des ressources limitées produisent généralement un modèle de conception qui ne répond qu'aux problèmes actuels. Répondre aux problèmes futurs nécessite souvent de retravailler toute l'implantation. Et comme dans l'implantation les préoccupations ne sont pas isolées, nous devons modifier de nombreux modules. Modifier chaque module pour répercuter les modifications apportées à une préoccupation peut conduire à des incohérences difficilement maitrisables.

La séparation des préoccupations fut réellement prise en charge dans un premier temps au niveau implémentation. La Programmation Orientée Aspect [1] qui a été introduite pour répondre à certaines insuffisances de l'Orienté Objet permet d'exprimer de manière très claire cette séparation des préoccupations. Les fondements de l'orienté Aspect furent donc introduits par la Programmation Orienté Aspect et AspectJ, le premier langage Orienté Aspect représente aujourd'hui l'endroit ou sont concrétisés les derniers concepts, technique et mécanismes de l'orienté Aspect.

Généralement, nous distinguons deux grandes catégories de préoccupations dans une application informatique : le métier pur et le technique. Le métier pur se charge de la réalisation des fonctionnalités fondamentales d'un logiciel. Le technique se concentre sur les préoccupations souvent dite non-fonctionnelles. Les préoccupations techniques peuvent être vues comme des fonctionnalités de support qui permettent d'exploiter le métier pur selon des exigences bien précises [2] [3] . La sécurité la gestion des erreurs, la persistance des données, les IHM, la journalisation et le traçage sont souvent classées comme préoccupations technique. De manière plus générale, toute préoccupation non métier est considérée copmme technique. La figure 1 donne une brève illustration de l'intégration dans un même systeme des diverses préoccupations métier et techniques dans le cadre de la Programmation Orientée Aspect.

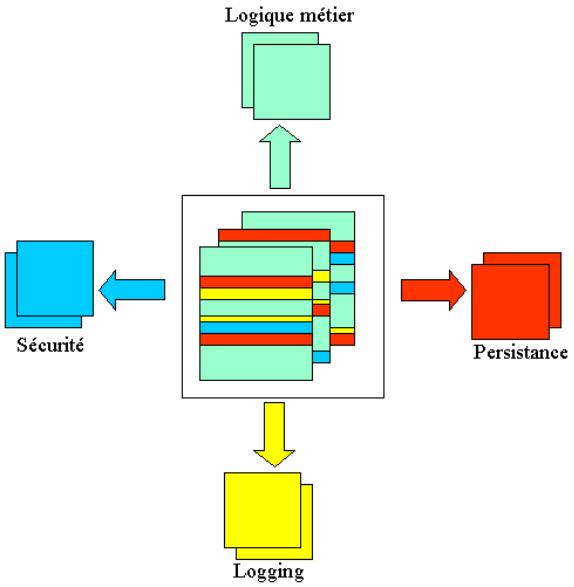



Figure 1 : Les Préoccupations d'un système

A titre d'exemple, l'abstraction d'une carte de crédit admet une préoccupation fonctionnelle qui concerne le processus de payement et des préoccupations non fonctionnelles assurant la gestion de la connexion, l'authentification, l'intégrité de la transaction de paiement, la sécurité, etc.

La programmation par aspects (AOP, Aspect Oriented Programming) introduite
en 1997 par la socité Xerox, permet de faire la séparation explicite des fonctionnalités métier des fonctionnalités dites transversales que nous trouvons dans tout logiciel. Les fonctionnalités transverses (ou préoccupations techniques) sont mise chacune dans un module appelé *aspect* dans AspectJ qui est le premier langage Orienté Aspect. Une fois les différents aspects définis, ils sont assemblés avec le métier pour produire l'application. Ce processus d'intégration, essentiellement automatique, est appelé *tissage* (weaving) des aspects dans le métier. Le tissage des aspects est réalisé en des points particuliers du métier. Ces points sont appelés des *points de jonction*.

En plus de l'amélioration de la qualité du code et de rendre plus efficace la réutilisation et la maintenance, la POA permet de résoudre les problèmes dus à l'enchevêtrement et l'éparpillement du code. Elle permet aussi de modulariser l'implantation des problématiques transverses, de créer des systèmes plus évolutifs. Les études montrent que la surcharge introduite par les approches

orientées aspect est relativement faible. Les implantations orientées aspect ont des niveaux d'adaptabilité et de réutilisation plus élevés que les implantations uniquement objet. Une étude récente [1] a mis en évidence le fait que les programmeurs trouvent plus facilement la cause du problème en cas de bug dans le code source d'un langage orienté aspect. Les développeurs passent moins de temps à essayer de comprendre la sémantique des instructions et gagnent donc un temps considérable.

Du fait des grands avantage observé au niveau de la POA, aujourd'hui il y'a un accord croissant que les concepts introduits par la POA doivent être généralisé à tout le cycle de développement de logiciel en commençant par l'analyse de besoins et en passant par la définition de l'architecture du logiciel et sa conception [5] . Aujourd'hui nous parlons de plus en plus de Conception Orientée Aspects ou AOSD (Aspect Oriented Software Design) que de POA.

La spécification architecturelle est une étape importante dans le cycle de vie de développement d'un logiciel et un facteur critique dans sa réussite, c'est pour cette raison qu'elle doit être spécifiée de manière précise et commune. Aujourd'hui l'architecture logicielle représente la discipline du génie logiciel qui permet de prendre en charge de manière formelle toutes les activités relative à la phase spécification de l'architecture d'une application. Dans cette nouvelle discipline la spécification d'architecture est réalisé à l'aide des langages de Description d'Architecture Logicielle souvent appelé ADL (Architrecture Description Language). L'architecture logicielle se base sur deux éléments fondamentaux : Le composant et le connecteur. L'architecture logicielle vise à construire des systèmes par assemblages de composants ayant des propriétés certifiées à l'aide de connecteurs.

Un composant exprime de manière explicite les services requis pour son fonctionnement et les services qu'ils peuvent offrir (Figure 2). Cette expression se fait à travers les interfaces du composant. Un connecteur lie souvent une interface exposant un service requis à une interface qui fournit le service demandé.

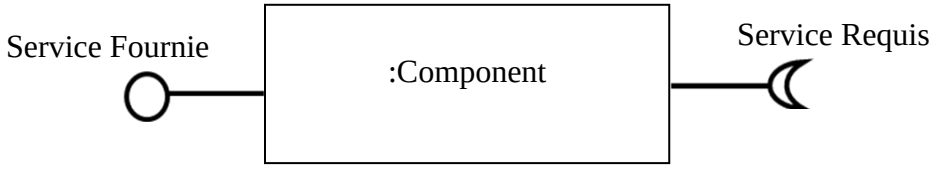



Figure 2 : Modele général d'un composant en Architecture Logicielle

Problématique

Comme les composants, les aspects répondent à une problématique [6] [8] de séparation des préoccupations. Il s'agit de regrouper dans une entité logicielle distincte du code correspondant à une préoccupation donnée. La séparation des préoccupations fournit un support méthodologique de modélisation et de programmation. Elle doit bien sur être accompagnée d'un processus d'intégration des différents composants générés pour les différentes préoccupations.

L'Orienté Aspect a été introduit en programmation pour permettre de résoudre un problème de réutilisation qui s'est posé dans l'orienté objet. C'est ainsi que même après avoir mis les diverses préoccupation dans des modules distincts, les classes d'objet qui maintenant ne contiennent plus de code transverses, faisaient référence de manière explicites aux préoccupations transverses. Ainsi lors de l'instanciation d'un objet, il devient nécessaire de résoudre la référence. L'instanciation d'une classe nécessite obligatoirement l'instanciation des aspects technique référencés. Il n'est pas possible d'instancier une classe indépendamment des aspects référencé. Cette situation a aussi été observée en Architecture Logicielle. C'est ainsi que les composants exposaient de manière explicite leurs besoins en aspect technique (sécurité et persistances de la figure 3) et nécessitaient pour leur instanciation la présence obligatoire des composants techniques. Ainsi, de tels composants ne peuvent pas être réutilisés indépendamment d'un certain nombre d'aspects techniques bien précis.

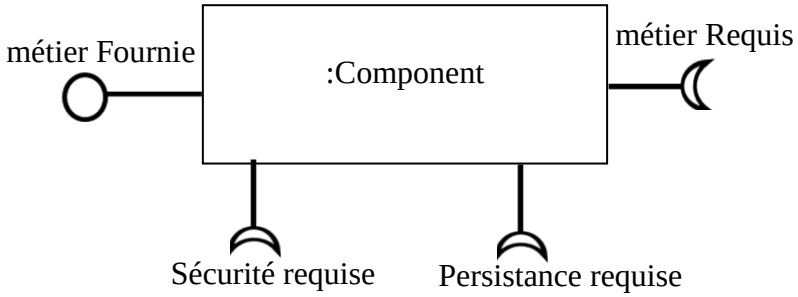



Figure 3 : Présentation d'un composant

Diverses approches d'architecture logicielle ont essayé de résoudre ce problème de réutilisation de composant par l'introduction de l'Orienté Aspect en Architecture logicielle. La quasi-totalité des approches ont introduit l'Orienté Aspect par extension pur et simple d'approches existante d'Architecture logicielle. Cette manière, d'introduire les aspects, et que nous qualifierons de brutale, ne s'est pas faite sans conséquence sur les approches Architecture Logicielles

étendue et sur les concepts d'aspects. C'est ainsi qu'à titre d'exemple que l'approche ACME supporte le concept d'aspect au prix fort de permettre l'introduction d'un connecteur spécial permettant de lier deux services fourni, ce qui n'est pas de coutume en Architecture Logicielle.

Objectif :

Notre travail s'inscrit dans le contexte du projet IASA (Integrated Approach to Software Architecture) dont l'objectif premier est de mettre sur place une approche architecture logicielle qui doit supporter de manière native les concepts d'aspect. Cette approche dispose d'un langage de description d'Architecture logiciel qui doit permettre de spécifier la structure, le comportement et l'orienté aspect.

Actuellement l'approche IASA ne supporte pas les aspects. Elle se base sur un modèle de composant, d'interface et de connecteur très flexibles. La flexibilité dans les modèles d'IASA est due à l'objectif premier de l'approche IASA. Cet objectif de base est de permettre à IASA la spécification d'architecture mixte hardware/software, dans laquelle certains composants sont des composants software alors que les autres sont composant Hardware. C'est ainsi que l'interface d'un composant IASA est appelé port au lieu d'interface. Contrairement aux interfaces traditionnelles (Interface UML, Interface Java, Interface d'ADL etc..) qui sont atomique, les éléments structurels et comportementaux d'un port IASA sont accessible individuellement et manipulable indépendamment. C'est à travers cette flexibilité que nous essayerons d'atteindre notre objectif, a savoir transformer IASA en une approche native de conception orientée aspect.

Un autre aspect flexible de IASA c'est son langage de description d'Architecture logicielle appelé SEAL (Simple and Extensible Architecture an Action Language). Le concept d'action de SEAL est pris de Précise Action Semantic d'UML 2.0. L'ADL SEAL est facilement extensible dans deux directions : Ajout de nouvelles actions spécifique et leur intégration au niveau des bibliothèques du langage et le support du concept d'alias d'action. L'extension par alias est un mécanisme qui permet l'extension du vocabulaire dans une description d'Architecture. Grace au concept d'alias, il est possible d'adapter certaines actions associées à des éléments de modélisation au vocabulaire propre du concepteur.

Un autre objectif de notre travail est le respect de l'exigence suivante : lors de l'introduction de l'orienté aspect dans IASA, aucun concepts fondamental de l'Architecture logicielle, notamment ceux définis par IASA ne doivent être remis en cause. De cette manière l'orienté aspect devrait paraitre comme s'il était un des objectifs fondamentaux lors de la définition d'IASA. Le support de l'Orienté Aspect Devra apparaitre comme natif.

Organisation du mémoire :

Pour atteindre nos objectifs, nous avions dans un premier temps essayé de maitriser l'orienté aspect. C'est ainsi que le premier chapitre est dédié à une étude de l'état de l'art de l'orienté aspect, que ce soit au niveau de la programmation, des cadres de travail (Framework) ou de l'Architecture logicielle.

La transformation de l'approche IASA en une approche native d'Architecture Logicielle Orientée Aspect est l'objectif fondamental de notre étude. La présentation de IASA dans sa première version est ainsi nécessaire. Cette présentation sera réalisée au niveau du deuxième chapitre.

Le troisième chapitre sera consacré aux concepts, techniques et mécanismes que nous avons définis pour rendre IASA une approche native d'Architecture Logicielle Orientée Aspect. C'est au niveau de ce chapitre que nous montrerons comment les concepts de port a été exploité pour faire supporter à IASA le concept d'Aspect. Dans ce même chapitre nous avons présenté une nouvelle vue sur l'organisation interne des composants IASA et nous avons proposé l'introduction d'une nouvelle clause dans le langage SEAL. Le Langage SEAL aura après introduction des aspects un nouveau nom : C'est 3ADL pour Architecture, Action an Aspect Description Language.

Dans le chapitre 4 on va présenter notre outil IASA studio, le chapitre 5 sera concerné par la validation de notre approche et le chapitre 6 conclura ce mémoire.

# CHAPITRE 1*:*ETAT DE L'ART

## 1.1 Introduction :

Le principe de séparation des préoccupations est un concept présent depuis de nombreuses années en génie logiciel. Il consiste à identifier séparément l'ensemble des préoccupations d'un système logiciel et à les adresser relativement d'une façon indépendante, aussi bien à la phase de conception qu'à la phase d'implantation.

Grâce à ces nombreux atouts, ce principe est reconnu essentiel au développement des logiciels : il améliore la lisibilité et la flexibilité du système permettant ainsi l'évolution, l'adaptation et la réutilisation de toutes ou une partie de l'application.

La Programmation Orientée Aspect est la première discipline ayant permis la mise en œuvre et l'appréciation du principe de la séparation des préoccupations qui entrecoupent les fonctionnalités principales d'un système. Grâce au paradigme aspect, le code des préoccupations transversales peut être regroupé au sein de modules au lieu d'être dispersé à travers les autres modules d'un système.

Au cours des dernières années le concept d'aspect attire de plus en plus l'attention des chercheurs pour sa généralisation à toutes les phases du cycle de développement logiciel. Aujourd'hui il est plus question de Conception Orientée Aspect que de Programmation Orientée Aspect. Les résultats ont montré les avantages de l'application de l'orienté Aspect dans le processus de développement d'applications notamment en ce qui concerne l'organisation, la réutilisation et la capacité d'adaptation.

Nous présentons dans ce que suis les concepts et mécanismes fondamentaux de l'orienté aspect à travers la POA (Programmation Orientée Aspect). Par la suite nous présenterons les trois grandes orientations ou sont mis en œuvre les aspects : Le niveau programmation, le niveau cadre de travail basé sur le concept de composant et le niveau Architecture logicielle.

## 1.2 La programmation orientée aspect :

La Programmation Orienté Aspects (POA) a vu le jour officiellement en 1996 à Xerox PARC (*Palo Alto Research Center*) comme résultats des travaux Gregor Kiczales et son équipe [9]. C'est un paradigme qui fut défini au départ pour le niveau programmation et qui s'est concrétisé par le langage AspectJ qui représente le premier langage orienté aspect. Tout comme l'a été la Programmation Orientée Objet (POO), la POA est une nouvelle orientation en programmation, née suite aux diverses limites de la POO. La programmation orientée aspect se base sur un certain nombre de concepts et mécanismes : les plus importants sont la séparation des préoccupations, le conseil, le point de jonction, le type de conseil, la coupe et le tissage des préoccupations pour former l'application à exécuter.

**Le conseil** (*advice* en Anglais) parfois appellé **greffon** : Le conseil représente une fonctionnalité transverse et correspond souvent à une exigences ou propriété non fonctionnelle. C'est un support nécessaire au fonctionnement correct du métier.

**Le point de jonction** (*joinpoint* en Anglais): C'est l'endroit dans le métier ou sera inséré le conseil. Le point de jonction pourrait être un appel de fonction, la modification d'une variable, la production d'une exception.

**Le type de conseil** : Par rapport à un point de jonction, il indique à quel moment le conseil doit entre en action. Nous distinguons globalement trois type de conseil : avant, après et autour.

**La coupe** (*pointcut* en Anglais) : Une coupe est un ensemble de point de jonction. Cet ensemble est souvent concerné par l'insertion d'un même conseil avec un même type de conseil. La coupe est souvent spécifiée sous forme d'une expression régulière. L'évaluation de l'expression régulière fait apparaitre tous les points de jonction ciblée par une insertion de conseil.

**Le tissage** (weaving en anglais): C'est l'opération de production du système à partir des divers modules contenu le métier et des modules contenant les conseils (module représentant les aspects techniques). Le tissage est accompli en se basant sur la coupe et les type de conseil. La coupe indiquera les endroits ou mettre un conseil et le type préciseras le moment ou le conseil doit être effectif au niveau des points de jonctions de la coupe. Le tissage peut être statique ou

dynamique. Le tissage statique est réalisé durant la compilation. Le tissage dynamique prend effet durant l'exécution.

La POA est encore relativement jeune comparée à la POO. Il reste de nombreuses zones d'ombres à éclaircir notamment en ce qui concerne la portabilité des aspects. Même si un effort de standardisation de la POA semble être en cours pour la plateforme Java, le temps où les aspects seront complètement portables d'un langage à un autre semble encore bien éloigné. Peut être faudrait il se pencher sur un sorte de méta approche de l'AOP en utilisant ce que l'on pourrait appeler des **'Platform indépendant Aspects'**, qui serait projeté façon automatique sur des **'Platform dépendant Aspects'**, et de se rapprocher ainsi de la philosophie du Model Driven Architecture mise en avant par l'OMG.

## 1.3 L'orienté aspect et les composants logiciel:

L'étude de la programmation orientée objet et les diverses expériences réalisées en utilisant le langage AspectJ, nous a permis d'apprécier et de comprendre manière très précise les divers concepts de l'orienté aspect et de l'importance de l'orienté aspect dans un processus qui veut produire du logiciel de qualité. Nous avons aussi été très sensibilisés à l'importance de la généralisation du concept d'aspect à toutes les phases d'un processus de conception et aux modèles utilisé pour raisonner sur une solution logicielle. Notre objectif est d'introduire les aspects dans une approche d'architecture logicielle. Dans ces approches, le composant logiciel représente l'élément fondamental. Le concept de composant logiciel n'est pas seulement utilisé en Architecture logicielle, mais il est utilisé dans certains langages qui se basent sur le composant logiciel tel que les Java BEAN et dans le cadre de travail qui basent sur les composants EJB. Vis-à-vis du composant de l'Architecture logiciel, les Java BEAN ou les EJB sont appelés des composants industriels.

Ainsi notre étude sur l'état de l'art ne s'est pas restreinte aux aspects dans les approches Architecture Logicielle uniquement, mas elle a aussi porté sur les aspect dans toutes les approches basé sur le concept de composant qui est un concept fondamental en Architecture Logicielle. L'étude réalisée nous a permis d'identifier trois grandes approches dans lesquelles les concepts d'aspect ont été exploités et mis en œuvre de manière intéressante. Ces trois tendances sont : [8] [58] [65] [66] [67] :

- Les cadres de travail basé sur les composants logiciels (*Component Based Software Engineering frameworks)*: Les grands représentant de cette tendance sont SPring-AOP, JBOSS-AOP et JAC. Les composants logiciels sont des composants dits industriel tels que les EJB (Entreprise Java Bean). La grande différence entre un composant de langage de programmation comme le Java BEAN et un composant industriel comme l'EJB réside dans le fait que le composant industriel nécessite un conteneur pour l'exécuter alors que le composant de langage de programmation est exécutable directement sans conteneur.
- L'approche aspect au niveau des langages de programmation basé sur le composant: Le composant dans ces approches ne correspond pas à un modèle de composant de l'architecture logicielle. Le composant est en fait un objet tel que les Java BEAN. Les représentants de cette approche sont JAsco, Aspectual Component, CAESAR et Open Module
- L'Architecture logicielle : Le composant en Architecture Logicielle se base un modèle abstrait. L'orienté aspect en Architecture Logicielle est ainsi traité à un haut niveau d'abstraction qui est indépendant du langage de programmation. Les approches architecture logicielle muni de l'orienté aspect sont appelé des Architecture Logicielles Orientées Aspect.

Dans ce qui suit nous présenterons une par une ces trois tendances. Une comparaison de ces tendances avec notre approche sera présentée au chapitre 4. L'évaluation de ces tendance sera faite en considérant les points suivants :le type de conseil supporté, le type de point de jonction ciblé par un conseil, la forme de spécification des coupes, l'indépendance de la spécification des coupe et des conseil, les technique de spécification et d'application du tissage, la réutilisation d'aspect, la composition et l'ordonnancement des aspects, le type de tissage (dynamique ou statique) et les traces comme point de jonction.

### 1.3.1 Les cadres de travail basé sur les composants logiciels (CBSE[2]) (*Component Based Software Engineering frameworks):*

Nous étudierons dans cette section les Framework basé sur les composants logiciels, à savoir *SPring-AOP, JBOSS-AOP et JAC*

#### 1.3.1.1 Spring AOP:

##### 1.3.1.1.1 Introduction :

Spring est un Framework open source J2EE pour les applications n-tiers, dont il facilite le développement et les tests. Il est considéré comme un conteneur léger qui permet une intégration des services J2EE. Spring s'appuie principalement sur l'intégration de trois concepts clés [52] :

1. l'inversion de contrôle ou injection de dépendance (IoC).
2. la programmation orientée aspect (AOP).
3. une couche d'abstraction.

Le Framework Spring est organisé en modules [51], reposant tous sur le module Spring Core (voir Figure 1.1):

- **Spring Core :** implémente notamment le concept d'inversion de contrôle (injection de dépendance). Il est également responsable de la gestion et de la configuration du conteneur.
- **Spring Context: Ce** module étends Spring Core. Il fournit une sorte de base de données d'objets, permet de charger des ressources (telles que des fichiers de configuration) ou encore la propagation d'évènements et la création de contexte comme par exemple le support de Spring dans un conteneur de Servlet.
- **Spring AOP :** Permet d'intégrer de la programmation orientée aspect.
- **Spring DAO :** Ce module permet d'abstraire les accès à la base de données, d'éliminer le code redondant et également d'abstraire les messages d'erreur spécifiques à chaque vendeur. Il fournit en outre une gestion des transactions.
- **Spring ORM :** Cette partie permet d'intégrer des frameworks de mapping Object/Relationnel tel que Hibernate, JDO ou iBatis avec Spring. La quantité de

[2] Component based software engineering

code économisé par ce package peut être très impressionnante (ouverture, fermeture de session, gestion des erreurs).

- **Spring Web :** Ensemble d'utilitaires pour les applications web. Par exemple une servlet qui démarre le contexte (le conteneur) au démarrage d'une application web. Permet également d'utiliser des requêtes http de type multi part.
- **Spring Web MVC[3] :** Implémentation du modèle MVC.

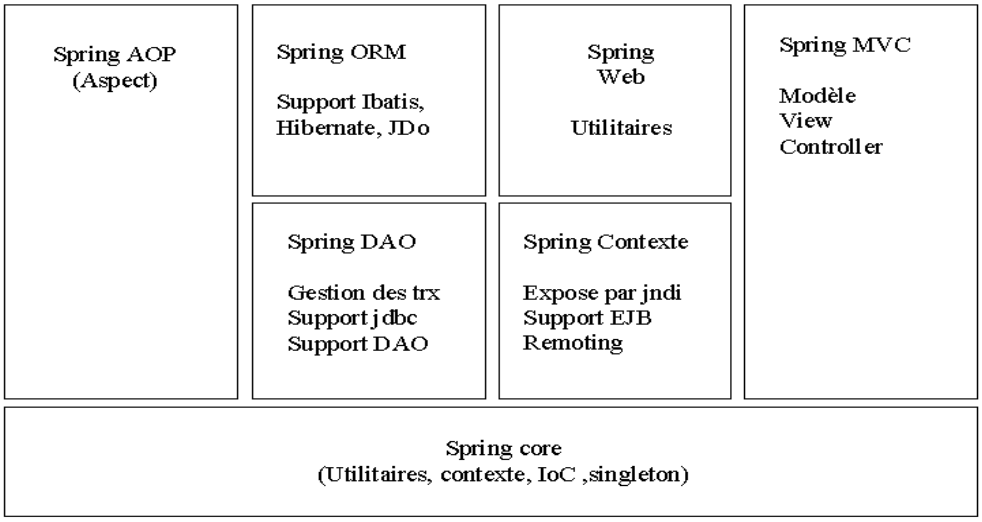



Figure 1.1- Schéma représentant l'organisation des modules et fonctionnalités à l'intérieur du Framework SPRING

1.3.1.1.2 Les aspects dans Spring AOP:

La partie aspect de Spring [51] est en pur Java et implante donc des interfaces du canevas Spring. Ces interfaces existent pour les coupes ou point d'actions et les codes advice, ou encore les *advisor* qui encapsulent les deux à la fois. Spring AOP emploie le terme *advisor,* en référence à la notion d'advice. Spring-AOP permet d'ajouter, par configuration, du "comportement" à une méthode de classe sans modifier son code.

Les aspects Spring [50] peuvent s'appliquer sur les composants *beans* qui sont de toute manière des objets. Les aspects peuvent à la fois atteindre les objets hébergés par un conteneur à inversion de contrôle ou des objets extérieurs. Contrairement à JBoss AOP, Spring AOP supporte uniquement l'*interception* et non l'*introduction*.

[3] *Model-View-Controller* est une architecture qui organise l'interface homme-machine (IHM) d'une application logicielle. Ce paradigme divise l'IHM en un modèle (modèle de données), une vue (présentation, interface utilisateur) et un contrôleur (logique de contrôle, gestion des événements, synchronisation).

```
1@Aspect
2public class AnAspect {
3 @Pointcut("execution(* com.xyz.someapp.service.*.*(..))")
4 public void  businessService() {}
5}
```

Figure 1.2 – Spring AOP : exemple d'aspect utilisant les annotations et AspectJ.

De plus, l'interception se limite aux méthodes, car l'accès aux attributs de classe est considéré comme une violation de l'encapsulation par les auteurs de Spring.

Les coupes ou les points d'actions (pointcut en anglais) sont définies en implantant des interfaces fournies par le canevas, par annotations[4] (voir **Figure 1.1.2**), ou alors par le biais de descripteurs XML à l'aide d'expressions régulières (voir illustration du **Figure 1.1.3**).

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

Figure 1.3– Spring AOP : exemple d'aspect utilisant les descripteurs XML.

Dans le fichier de configuration : On déclare le Bean *myAspect* et on spécifie la configuration AOP.

La première étape pour déclarer les aspects, pointcuts et les advisors on utilisant des descripteurs XML, est de créer un fichier Spring XML. Dans se fichier on peut déclarer des multiples balises <aop:config> dans un ou plusieurs fichiers XML. La balise <aop:config> peut contenir les balises suivante:

• <aop:aspect>: Permet de créer des aspects dans des fichiers XML qui sont comparable aux annotations.

[4] Les Annotations permettent de marquer différents éléments du langage Java avec des attributs particuliers, dans le but d'automatiser certains traitements et même d'ajouter des traitements avant la compilation grâce au nouvel outil du JDK : Annotation Processing Tool (APT). Les nouvelles annotations Java permettent de simplifier et de structurer l'utilisation des Méta-données.

• <aop:advisor>: Permet de créer des objets de type advisor avec les coupes (pointcuts) classiques de AspectJ.

• <aop:pointcut>: Permet de déclarer et réutiliser des pointcuts dans aspects utilisant les descripteurs XML.

1.3.1.1.2.1 Point de jonction et coupe:

Les coupes sont spécifiées sous forme d'expressions régulières selon un format propre à AspectJ. Le support d'AspectJ proposé par Spring AOP ne comprend qu'un seul type de point de jonction, ***execution***, qui intercepte les exécutions de méthodes. À ce point de jonction est associée une expression régulière spécifiant les méthodes à intercepter.

Voici quelques exemples d'expressions courantes pour une coupe.

- **execution(public * *(..))** : Toutes les méthodes public
- **execution(* set*(..))** : Toutes les méthodes commencant par 'set'
- **execution(* com.xyz.service.IAccountService.*(..))** : Toutes les méthodes de l'interface 'IAccountService'
- **execution(* com.xyz.service.*.*(..))** : Toutes les méthodes du package 'com.xyz.service'
- **execution(* com.xyz.service..*.*(..))** : Toutes les méthodes du package 'com.xyz.service' et de ses sous-packages.

On peut réutiliser les pointcuts déclarer par le biais de descripteurs XML ou ceux déclarés par annotations (Style @AspectJ**).**

1.3.1.1.2.2 Les type de conseils supporté :

SPRING-AOP supportes les types de conseil suivants :

- **Before advice**: Exécuté avant le join point.
- **After returning advice**: Exécuté après le join point (Si la méthode interceptée s'exécute normalement – sans retourner d'exception).
- **After throwing advice**: Exécuté si la méthode interceptée retourne une exception.
- **After (finally) advice**: Exécuté en sortie du join point (exécution normale de la méthode ou sortie en exception).
- **Around advice**: Il englobe l'exécution de la méthode interceptée. Il permet de paramétrer le comportement avant et après exécution de la méthode. Il

permet ainsi d'exécuter ou non la méthode, de définir le retour souhaité, voir de lancer une exception

Le **Figure 1.1.4** propose un exemple de code advice de type ***around*** utilisant les annotations et AspectJ.

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

Figure 1.4– Spring AOP : exemple d'aspect utilisant les annotations etAspectJ.

Sur le **Figure 1.5** nous voyons le même aspect écrit en Spring AOP sans AspectJ.

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

Figure 1.5 – Spring AOP : exemple d'aspect sans les annotations.

1.3.1.1.2.3 Composition et ordonnancement d'aspect

La composition d'aspect se fait par le classique mécanisme de précédence d'advice d'AspectJ, c'est-à-dire qu'il est possible de spécifier globalement sur l'application que tel ou tel aspect sera appliqué avant tel autre. Le **Figure 1.6** propose un exemple de séquencement qui suit tout simplement l'ordre de déclaration des codes advice.

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

Figure 1.6– Spring AOP : mécanisme de séquencement d'advice.

Lorsque deux advices qui sont déclarés dans les différents aspects sont exécutés pour le même point se joindre, l'ordre est déterminé par l'interface org.springframework.core.Ordered, comme dans la Figure 1.7.

```
package org.springframework.core;
public interface Ordered {
int getOrder();
}
```

Figure 1.7. L'interface Ordered de Spring

Le framework Spring utilise l'interface *Ordered* à chaque fois une liste d'objets doivent être traitées dans un ordre particulier. On implémentant l'interface *Ordered* pour les aspects, on pout placer les advices dans un endroit bien précis de l'ordre d'exécution de ces advices pour les points de jonction. Les règles d'ordonnancement pour les aspects sont les suivants:

• Les Aspects qui n'implémentent pas l'interface *Ordered* sont dans un ordre indéterminé et venir après les aspects qui implémentent l'interface.

• Les Aspects qui implémentent l'interface *Ordered* sont classés en fonction de la valeur de retour de la GetOrder () méthode. Les valeurs les plus faibles de se retrouver première.

• Deux ou plusieurs aspects qui ont la même valeur de retour pour la GetOrder () sont dans un ordre indéterminé.

1.3.1.1.2.4 Le tissage ou Aspect Weaving :

Le tissage des aspects avec Spring AOP s'effectue grâce à la création dynamique [55] de proxy pour les Beans dont les méthodes doivent être interceptées. Les utilisateurs de ces méthodes obtiennent ainsi une référence à ces proxys (dont les méthodes reproduisent rigoureusement celles des Beans qu'ils encapsulent), et non une référence directe aux Beans tissés. La génération de ces proxys est contrôlée au niveau d'un fichier de configuration.

Spring AOP propose des modes de tissage automatique matérialisés par des Beans spécifiques à instancier [56][57]. Le mode le plus simple utilise la classe *DefaultAdvisorAutoProxyCreator*, qui doit être instanciée sous forme de Bean dans le conteneur léger. Ce mode se fonde sur les coupes gérées par l'ensemble des advisors de l'application pour identifier les proxys à générer automatiquement.

Le second mode permet de spécifier la liste des Beans à encapsuler dans des proxys. Ce mode utilise la classe *BeanNameAutoProxyCreator,* qui doit être instanciée sous forme de

Bean dans le conteneur léger.

Le tisseur prend en paramètre la liste des Beans à tisser sous forme d'un tableau de chaînes de caractères *via* la propriété beanNames et la liste des advisors à y appliquer *via* la propriété *interceptorNames*. Cette propriété accepte également les advices dans sa liste. Dans ce cas, les advices interceptent l'ensemble des méthodes des Beans concernés par le tissage.

1.3.1.1.3 Exemple illustratif :

Dans cet exemple [53], nous allons montrer comment tracer les appels d'une méthode dans une application basée sur Spring. Pendant le développement ou la maintenance d'une application, le développeur n'a pas forcément envie de mettre dans son code des log.debug() partout pour mieux comprendre le cheminement de l'application ou tracer un bug. Avec Spring, il est possible de tracer les appels aux méthodes objets pendant l'exécution et sans modifier le code de l'application. Nous allons voir avec cet exemple l'intérêt de *Spring AOP*.

- **La classe MessageService (Figure 1.8) :** Elle dispose d'un service envoie un message d'un expéditeur sur la sortie par défaut du système.

```
public class MessageService {
                    public String envoiMsg(String expediteur, String msg){
                    String message ="-> " + expediteur + " dit : " + msg + "\n";
                    System.out.print(message);      return message;}}
```

Figure 1.8– La classe MessageService.

- La classe du test unitaire **MessageServiceTest (Figure 1.9)** est défini pour permettre de tester le petit service de MessageService

```
public class MessageServiceTest {
        private ApplicationContext context;
        /**
         * Méthode appelée à l'initialisation de la classe de test Unitaire.
         * @throws Exception
         */
        @Before
        public void setUp() throws Exception {
        context = new ClassPathXmlApplicationContext( new String[] {"spring.context.xml"} );
        }
        @Test
        public void testEnvoiMsg(){
                // On récupère le service de message instancié avec spring
                MessageService service = (MessageService)
context.getBean("MessageService");
                String expediteur = "Nico";
                String msg1 = "Bienvenu dans le tutoriel de Spring AOP!";
                // On envoie un premier message
                String msgEnvoye1 = service.envoiMsg(expediteur, msg1);
```

Figure 1.9 – La classe MessageServiceTest.

- **La classe de génération des logs *MessageLogger*** : Elle contient deux méthodes :

- *logMethodEntry()* qui affiche dans la console le nom de la méthode et ses paramètres .
- *logMethodExit()* qui affiche dans la console le nom de la méthode et ce qu'elle envoie.

```
public class MessageLogger {
        public void logMethodEntry(JoinPoint joinPoint) {

                Object[] args = joinPoint.getArgs();
                // Nom de la méthode interceptée
                String name = joinPoint.getSignature().toLongString();
                StringBuffer sb = new StringBuffer(name + " appelé avec en paramètre : [");
                // Liste des valeurs des arguments reçus par la méthode
                for (int i = 0; i < args.length; i++) {
                        Object o = args[i];
                        sb.append("'" + o + "'");
                        sb.append((i == args.length - 1) ? "" : ", ");
                 }
        public void logMethodExit(StaticPart staticPart, Object result) {
                // Nom de la méthode interceptée
                String name = staticPart.getSignature().toLongString();
                System.out.println (name + " retourne : [" + result + "]");          }}
```

Figure 1.10 – La classe MessageLogger

- **Configuration de Spring**

Se fait à travers un fichier XML dans lequel nous trouvons :

- La déclaration les beans *MessageService* et *MessageLogger*.
- La spécification de la configuration AOP.

Dans le fichier XML (voir **Figure 1.8**) au niveau de la configuration AOP, on déclare :

- **<aop:pointcut id="servicePointcut" expression="execution(* com.scub.foundation.samples.spring.aop.service.*.*(..))" />** : permet de définir des points d'interception sur les objets suivants :

  **com.scub.foundation.samples.spring.aop.service.*.*** signifie que toutes les méthodes des objets qui sont dans le package com.scub.foundation.samples.spring.aop.service seront interceptées

- **<aop:aspect id="loggingAspect" ref="MessageLogger">** : les appels aux méthodes seront renvoyés vers le bean Spring MessageLogger définit auparavant.
- **<aop:before method="logMethodEntry" pointcut-ref="servicePointcut" />** :

avant l'exécution de n'importe quelle méthode d'une classe du package com.scub.foundation.samples.spring.aop.service, la méthode logMethodEntry() est appelée

**<aop:after-retunrning method="logMethodExit" returning="result" pointcut-ref="servicePointcut" />** : après l'exécution de n'importe quelle méthode d'une classe du package com.scub.foundation.samples.spring.aop.service, la méthode logMethodExit() est appelée et le résultat retourné lui sera passé en argument.

```
<bean id="MessageLogger"
                    class="com.scub.foundation.samples.spring.aop.service.MessageLogger" />
    <bean id="MessageService"
                    class="com.scub.foundation.samples.spring.aop.service.MessageService" />
    <!-- Début de la configuration AOP -->
    <aop:config>
            <aop:pointcut id="servicePointcut"
                    expression="execution(*
    com.scub.foundation.samples.spring.aop.service.*.*(..))" />
            <aop:aspect id="loggingAspect" ref="MessageLogger">
                    <aop:before method="logMethodEntry" pointcut-ref="servicePointcut" />
                    <aop:after-returning method="logMethodExit"
                            returning="result" pointcut-ref="servicePointcut" />
            </aop:aspect>
    </aop:config>
    <!-- Fin de la configuration AOP -->
    </beans>
```

Figure 1.11 - Exemple illustratif d'aspect utilisant les descripteurs XML.

On constate bien que pour chaque appel de méthode des classes du package *com.scub.foundation.samples.spring.aop.service* dans notre test unitaire, des logs avant et après appel sont produits dans la console. Et tout cela sans avoir apporté la moindre modification à notre classe de service *MessageService* initiale.

1.3.1.1.4 Evaluation :

Spring intègre les fonctionnalités majeures d'AspectJ. Cependant, le support de la POA proposé par Spring est plus limité que celui d'AspectJ en termes de points de jonction pour définir les coupes (Spring ne supporte qu'un seul type de point de jonction '***execution'***).

Par contre, la composition d'aspects sur un même point de jonction reste de séquencement simple d'aspects, sans possibilité de manipuler les autres aspects en collision sur ce même point. Le Tableau 2 résume les caractéristiques de Spring AOP vis-à-vis de ces critères.

Grâce à la richesse de Spring AOP, le développeur dispose d'une large palette d'outils pour définir les aspects qui lui sont nécessaires. Cependant, Spring AOP n'offre pas toute la richesse fonctionnelle des outils de POA spécialisés. Il ne peut, par exemple, intercepter que les exécutions de méthodes des beans gérés par le conteneur léger.

Spring-AOP est une autre approche qui introduit l'AOP dans la plateforme Spring, qui supporte les composants EJB. Un aspect dans Spring-AOP [8] correspond à un ensemble d'advices défini dans une seule classe. La définition de tissage et des coupes se fait à l'aide de balises XML d'une manière séparée de la définition des aspects. Cependant, l'approche adoptée par Spring-AOP consiste à voir le code de base comme des objets. Le concept de composant n'est pas directement pris en compte.

| | |
|---|---|
| **Point de jonction sur trace (trace pointcut)** | **NON** |
| **Type de tissage : dynamique/ Statique** | **dynamique** |
| **Composition d'aspect** | **(possible) L'implémentation de l'interface (org.springframework.core.Ordered) pour spécifier l'ordre** |
| **Réutilisation aspect** | **OUI** |
| **Spécification du tissage (Application)** | **A travers des proxys (JDK dynamic proxy)** |
| **Indépendance coupe / aspect** | **OUI** |
| **Spécification Coupe** | **(d'expressions régulières Java) Par annotations /fichiers de configuration XML** |
| **Type de Point de jonction** | **Exécution** |
| **Type de Conseil** | **Before, After returning, After throwing, After (finally), Around** |
| **Gestion des Aspects au niveau d'un point de Jonction : Ordre d'exécution des aspects** | **Spécifié explicitement en implémentant l'interface org.springframework.core.Ordered** |
| **Type d'advice par point de jonction** | **Hétérogène** |
| **Nombre d'advice supporté par un point de jonction** | **illimité** |

**Tableau 1.1– Bilan de Spring AOP**

#### 1.3.1.2 JBOSS-AOP:

##### 1.3.1.2.1 Introduction :

**JBoss** [62] [61] est un serveur d'applications J2EE Libre entièrement écrit en Java, publié sous licence GNU LGPL. **JBoss AOP** [61] est un Framework pour la programmation orientée aspect en Java. Les aspects sont décrits en XML, mais même si la terminologie et la syntaxe diffèrent, les concepts sont globalement les mêmes qu'avec AspectJ. Une différence importante est que le déploiement des aspects peut se faire dynamiquement pendant l'exécution de l'application.

##### 1.3.1.2.2 Les aspects dans JBoss AOP:

Un aspect dans JBOSS-AOP correspond à un ensemble d'advices défini dans une seule classe. La définition de tissage et des coupes se fait à l'aide de balises XML d'une manière séparée de la définition des aspects. Cependant, l'approche adoptée par JBOSS-AOP consiste à voir le code de base comme des objets. Le concept de composant n'est pas directement pris en compte. Le mot intercepteur est utilisé dans JBoss AOP, il est l'équivalent de code advice dans AspectJ. JBoss AOP utilise un fichier XML : il s'agit de **jbossaop.xml.**

###### 1.3.1.2.2.1 Point de jonction et coupe

Les expressions de point de coupe sont associées aux Advices soit à l'aide d'annotations [60] [61], soit à l'aide de fichiers de configuration XML. JBoss AOP utilise un langage de point de coupe spécifique proche de celui d'AspectJ. JBoss AOP fournit un ensemble de balises XML et d'attributs pour définir des coupes et leurs intercepteurs associés.

Les deux balises principales utilisées par JBoss AOP pour définir des coupes sont **<bind>** et **<interceptor>** .

- **<bind>** permet de désigner les points de jonction appartenant à la coupe,
- **<interceptor>** indique l'intercepteur (code advice) associé à ces points de jonction.

```
<aop>
    <bind (1)
        pointcut="execution( (2)
        public void aop.jboss.Order->addItem(java.lang.String,int))"
      >
  <interceptor class="aop.jboss.TraceInterceptor" /> (3)
</aop>
```

Figure 1.12 - JBoss AOP : exemple d'aspect utilisant les descripteurs XML.

Chaque application est associée à un fichier XML de définition de coupe appelé habituellement **jboss-aop.xml.** Plusieurs coupes peuvent être définies dans un même fichier **jboss-aop.xml.**

La balise <bind> (ligne 1) débute la définition d'une coupe. L'attribut *pointcut* fournit l'expression de coupe. Cette expression est constituée à l'aide de mot-clé et de valeur. Le mot-clé cité dans l'exemple (ligne 2) correspond aux points de jonction de type exécution de méthodes. La balise <interceptor> (ligne 3) fournit, à travers l'attribut class, le nom de la classe implémentant l'intercepteur. Il s'agit ici de la classe aop.jboss.TraceInterceptor.

JBoss AOP fournis plusieurs types de point de jonction, ***execution*** qui sont les suivant:

- **Les coupes de types exécutions de méthode :** JBoss AOP fournit le mot-clé **execution** pour les coupes de types exécutions de méthode. Par exemple, la coupe suivante :**<bind pointcut="execution(public void aop.jboss.Order->addItem(..))" >** : Désigne les exécutions de toutes les méthodes addItem, quel que soit leur profil de paramètres.
- **Les coupes de type constructeur :** Le mot-clé correspondant est, comme pour les exécutions de méthodes, **execution**. Par exemple, l'expression suivante **: <bind pointcut="execution(public aop.jboss.Order->new(..))" > :** Désigne les exécutions de tous les constructeurs de la classe aop.jboss.Order.
- **Les coupes de type attribut:** concernent les lectures et les écritures d'attributs. Trois mots-clés sont fournis par JBoss AOP : **get, set** et **field**. Le premier désigne les opérations de lecture d'attributs, le deuxième les écritures et le troisième à la fois les lectures et les écritures. Par exemple, l'expression suivante :**<bind pointcut="set(private * aop.jboss.Order->articles)" >** : Dsigne les opérations d'écriture de l'attribut privé articles défini dans la classe aop.jboss.Order.
- **Les coupes de type classe :** regroupent les trois types vus précédemment : exécution de méthodes, constructeur et attribut. Par exemple, l'expression suivante :**<bind pointcut="all(aop.jboss.O*)" >** :Désigne toutes les exécutions de méthodes, de constructeurs et toutes les opérations de lecture et d'écriture d'attributs situées dans toutes les classes du package aop.jboss dont le nom commence par O.

- **Les coupes de type appel de méthode :** Il s'agit donc d'inclure dans la coupe tous les points de jonction qui correspondent à l'appel d'une ou de plusieurs méthodes. Par exemple, la coupe suivante : **<bind pointcut="call(* aop.jboss.Order->*(..))" >** : Désigne tous les appels à une des méthodes de la classe aop.jboss.Order.

1.3.1.2.2.2 Les type de conseils supporté :

Dans JBoss AOP [60], Les codes advice sont des méthodes java classiques. Les concepteurs de JBoss AOP parlent d'***intercepteur*** plutôt que de code advice. Le code d'un intercepteur, comme un code advice, s'exécute avant ou après un point de jonction.

Les intercepteurs de JBoss AOP fournissent du code qui s'exécute avant ou après les points de jonction. Le code d'un intercepteur est fourni dans une classe qui doit implémenter l'interface org.jboss.aop.advice.Interceptor. La balise **<interceptor>** annonce un intercepteur. Cette balise est associée à un attribut class qui fournit le nom de la classe qui implémente l'intercepteur.

À titre d'exemple, La classe TraceInterceptor suivante (voir Figure 1.13) fournit le code de l'intercepteur associé à la coupe précédente :

```
public class TraceInterceptor implements Interceptor {
public String getName() { return "TraceInterceptor"; }    (1)
public Object invoke(Invocation invocation)      (2)
throws Throwable {
MethodInvocation mi = (MethodInvocation) invocation;      (3)
String methodName = mi.method.getName();
System.out.println("Avant "+methodName);
Object rsp = invocation.invokeNext();          (4)
System.out.println("Après "+methodName);
return rsp;} }
```

Figure 1.13- JBoss AOP : exemple du code Advice ou d'intercepteur

Plusieurs intercepteurs peuvent être associés à une coupe. JBoss AOP parle en ce cas de **pile d'intercepteurs**. Les intercepteurs sont exécutés dans leur ordre de définition dans la pile. Un même intercepteur peut apparaître plusieurs fois dans une pile. Il est alors exécuté plusieurs fois. L'appel de la méthode invokeNext dans le code d'un intercepteur permet de passer à l'exécution de l'intercepteur suivant ou à l'exécution du code correspondant au point de jonction s'il n'y a plus d'intercepteur. La balise <stack> pour définir une pile d'intercepteurs est suivie de la liste des intercepteurs faisant partie de la pile.

1.3.1.2.2.3 Composition d'aspect et ordonnancement d'aspect:

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

Il se peut qu'un même point de jonction soit désigné par plusieurs coupes. Dans ce cas, il est associé à plusieurs intercepteurs ou à plusieurs piles. L'ordre d'exécution des intercepteurs et des piles correspond à l'ordre de déclaration de la coupe dans le fichier **jboss-aop.xml.** JBoss AOP utilise la balise <precedence> dans le fichier jboss-aop.xml XML pour spécifier l'ordre. Dans tous les cas, l'ordre est global, et il n'ya aucune garantie que certains ordres ne seront pas respectées.

1.3.1.2.2.4 Tissage:

JBossAOP est un tisseur dual que l'on peut utiliser pour tisser soit statiquement à l'AspectJ, soit dynamiquement à la JAC.

Le tissage des aspects avec JBoss AOP peut être dynamique [59] et s'effectue par la génération des classes de proxies lors du déploiement. Le langage de tissage est très puissant et s'exprime dans un fichier de configuration XML. Il permet de tirer profit soit des zones de greffe banalisées offertes par le langage Java (début et fin de méthodes, lecture et écriture d'attributs...), soit des annotations que l'on pourrait placer dans le code cible. Le tissage dynamique intervient pendant l'exécution. Il permet donc une plus grande flexibilité (ajout, retrait changement d'aspects) qui a cependant un coût car le tissage dynamique intègre en plus une phase d'adaptation qui consiste à préparer l'application pour quelle puisse recevoir les nouveaux aspects.

La version statique du tisseur JBossAOP a un temps de tissage court et son résultat offre des performances du même ordre de grandeur que AspectJ.

1.3.1.2.3 Exemple illustratif :

Le premier aspect que nous allons écrire avec JBoss AOP trace les exécutions des méthodes de la classe Order. Le code de cet aspect comporte deux fichiers, **jboss-aop.xml** et **TraceInterceptor.java.** Le premier est un fichier XML qui définit une coupe à tisser au moment de l'installation de l'application.

- **Le descripteur XML :**

```
<?xml version="1.0" encoding="UTF-8"?>
<aop>
<bind pointcut="execution(Order->add*(..))" >
<interceptors>
<interceptor class="aop.jboss.TraceInterceptor" />
</interceptors>
</bind>
</aop>
```

Figure 1.14 -Exemple illustratif d'aspect utilisant les descripteurs XML.

- **Le code intercepteur :**

Le second un fichier Java qui fournit le code advice associé à cette coupe (voir Figure 1.15):

• intercepteur: classe implements *Interceptor*

• aspect: classe avec des méthodes de profil (Invocation) Object

```
public class TraceInterceptor implements Interceptor {
public String getName() { return "TraceInterceptor"; }
public InvocationResponse invoke(Invocation invocation)
throws Throwable {
MethodInvocation mi = (MethodInvocation) invocation;
String methodName = mi.method.getName();
System.out.println("Avant "+methodName);
InvocationResponse rsp = invocation.invokeNext();
System.out.println("Après "+methodName);
return rsp;} }
```

Figure 1.15- Exemple illustratif d'un intercepteur

- **Création de coupe :**

```
AdviceBinding binding =
new AdviceBinding("execution(POJO->new(..))",null);
binding.addInterceptor(SimpleInterceptor.class);
AspectManager.instance().addBinding(binding);
```

Figure 1.16- Exemple illustratif d'une coupe

1.3.1.2.4 Evaluation :

JBoss AOP, en plus d'offrir un mécanisme d'interception classique des approches par spects, offre également un mécanisme d'introduction[5], ainsi que quelques mécanismes de mixins. Les points de jonctions pris en compte par JBoss AOP sont les accès aux attributs, les invocations de méthodes, ou encore un constructeur d'objet. Lors d'une interception de méthode, il est possible d'intervenir du côté de l'appelant ou de l'appelé. L'interception ne s'arrête pas aux

[5] **Introduction :** Tissage spécial consistant à rajouter de nouvelles méthodes ou de nouveaux attributs à une classe cible, ou encore à rendre disponibles de nouveaux types (interfaces, classes,...) dans un projet cible.

frontières d'une classe, il est possible d'atteindre les éléments publics tout comme privés ou protégés.

Les coupes sont décrites indépendamment d'un aspect sous forme d'expressions régulières portées au niveau des descripteurs de l'application ou sous forme d'annotations. En ce sens, JBoss AOP suit la philosophie employée avec les EJB, qui consiste à conFigure 1.r les services techniques dans des descripteurs XML ou à annoter directement le code des composants. JBoss AOP vise principalement l'intégration des services techniques à l'aide d'aspects, comme l'exemple classique de la journalisation, ou encore les services transactionnels. Pour relier une coupe indépendante d'un code advice, JBoss AOP utilise un mécanisme de liaison qui encore une fois fonctionne soit par annotations soit dans les descripteurs XML. Les définitions de coupes et les liaisons sont ensuite résolues à l'exécution.

| Point de jonction sur trace (trace pointcut) | Type de tissage : dynamique/Statique | Composition d'aspect | Spécification du tissage (Application) | Indépendance coupe / aspect | Réutilisation Aspect | Spécification Coupe | Type de Point de jonction | Type de Conseil | Gestion des Aspects au niveau d'un point de Jonction : Ordre d'exécution des aspects | Type d'advice par point de jonction | Nombre d'advice supporté par un point de jonction |
|---|---|---|---|---|---|---|---|---|---|---|---|
| NON | dual dynamique/ statique | Non supporté (Juste des mécanismes de Détection de conflits) La balise <precedence> dans les fichiers XML | génération de proxies | OUI | OUI | (d'expressions régulières Java) Par annotations /fichiers de configuration XML | Exécution méthode, constructeur, attribut, all, et méthode call | Around Avant ou après un point de jonction | (il n'y a pas de stratégie) Juste des mécanismes de Détection de conflits | Non supporté | Non supporté |

Tableau 1.2– Bilan de JBoss AOP

#### 1.3.1.3 JAC:

##### 1.3.1.3.1 Introduction:

JAC (Java Aspect Components) est un serveur d'applications open-source orienté aspects. Contrairement à d'autres serveurs d'applications dans lesquels les services techniques sont codés en dur, ne peuvent pas être changés, retirés ou étendus, JAC s'appuie sur un modèle de programmation par aspects (AOP) pour adapter et sélectionner les services techniques nécessaires aux applications. Ces services sont définis dans des composants d'aspects (AC) qui sont des entités propres manipulées par le Framework JAC.

JAC présente deux niveaux de détails pour manipuler ces aspects : Un niveau configuration et Un niveau programmatif.

Le niveau configuration concerne la configuration d'aspect qui est un concept essentiel de JAC. Alors qu'AspectJ ne l'aborde pas du tout, c'est une des clés qui permettent d'adapter les aspects de JAC à des applications nouvelles et donc d'augmenter de façon importante leur réutilisabilité. Chaque aspect JAC est associé à un fichier de configuration. Le fichier de configuration d'aspect fournit les paramètres qui permettent d'adapter l'aspect à l'application sur laquelle il sera tissé.

Pour une application donnée, chaque aspect est associé à un fichier de configuration. Le contenu de ce fichier peut varier d'une exécution à une autre afin de tester différentes configurations de l'application. Un fichier de configuration peut également être modifié au cours d'une exécution. Il est alors possible de demander au Framework JAC de recharger ce fichier afin que les nouvelles valeurs soient prises en compte.

Le niveau programmatif concerne la façon de créer des aspects. La création de nouvel aspect se fait via l'extension d'une classe Aspect Component du Framework JAC (voir Figure 1.26). Les aspects sont entièrement décris en JAVA par des méthodes et des classes. Un aspect dans JAC est une instance de la classe AspectComponent, il peut, en cours d'exécution d'une application, être déployé ou retiré.

Les deux éléments principaux [67] de la mécanique interne de *JAC* sont les contrôleurs d'encapsulation (wrapper) et les composants d'aspects.

Le déclenchement des aspects se fait par l'intermédiaire de contrôleurs d'encapsulation qui surveille les objets de base et contrôlent leurs exécutions. Ils

déterminent, en particulier, comment les appels d'aspects doivent être enchaînés. Ils jouent le rôle de composeur d'aspects. Il choisira dans quel ordre appliquer l'ensemble des composants d'aspect sur un objet de base.

D'un point de vue technique, JAC se présent un conteneur d'aspects et de composants métiers. Les deux sont chargés à la demande en fonction des besoins des applications (contrairement aux serveurs d'applications comme les EJB où les services techniques sont codés en dur dans le conteneur).

1.3.1.3.2 Les aspects dans JAC:

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. A titre d'illustration (Figure 1.17), le premier aspect que nous allons écrire avec JAC trace les exécutions de la méthode **addItem** de la classe **Order**. Le code de cet aspect comporte deux classes : *TraceAspect* et *TraceWrapper*.

```
package aop.jac;

import org.objectweb.jac.core.AspectComponent;
public class TraceAspect extends AspectComponent {
public TraceAspect() {
pointcut(
".*", ¨  ligne 1
"aop.jac.Order", ¨ ligne 2
"addItem(java.lang.String,int):void", ¨ ligne 3
"aop.jac.TraceWrapper", ¨ ligne 4
null, false );}}
```

Figure 1.17. Exemple d'aspect de trace dans JAC

1.3.1.3.2.1 Point de jonction et coupe:

Les trois catégories de paramètres suivantes sont associées à la méthode pointcut d'un AspectComponent :

- **Expressions de coupe.** Définissent les points de jonction de l'application qui font partie de la coupe.
- **Wrapper associé à une coupe.** Fournit le code qui s'exécutera avant et après les points de jonction de la coupe. Ce code doit être écrit dans une sous-classe de la classe Wrapper.
- **Gestionnaire d'exception.** L'exécution des points de jonction associés à la coupe ou celle du wrapper peuvent générer des exceptions. Il est possible d'associer un gestionnaire d'exception à la coupe pour récupérer et traiter ces

exceptions. Ce gestionnaire est une méthode de la classe implémentant le wrapper.

La classe *TraceAspec*t (voir Figure 1.17) comporte un constructeur qui appelle la méthode pointcut. Cette méthode est héritée de la classe AspectComponent. Elle permet de déclarer une nouvelle coupe. Six paramètres sont fournis pour cela. Les trois premiers concernent la définition proprement dite de la coupe. Le quatrième, ici aop.jac.TraceWrapper, fournit le wrapper associé à la coupe. Les deux derniers sont des propriétés associées à la coupe. Ils définissent respectivement un gestionnaire d'exception et la façon dont les aspects sont instanciés.

Les trois premiers paramètres de la méthode pointcut définissent les méthodes de l'application qui font partie de la coupe. Il s'agit de définir quelles méthodes (ligne 3) de quels objets (ligne 1) de quelles classes (ligne 2) appartiennent à la coupe.

La coupe définie par l'aspect TraceAspect concerne la méthode addItem, prenant en paramètres une chaîne de caractères et un entier, sur tous les objets de la classe aop.jac.Order.

1.3.1.3.2.2 Advice ou Wrapper

Contrairement 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. Il laisse au programmeur, le choix de la suite du traitement. Le programmeur peut insérer des appels au contrôleur d'aspect (des appels à proceed) où bon lui semble. Il n'y a donc pas de notion de before et d'after. Ce n'est pas une limitation en soit puisque qu'avec la notion d'around, il suffit de placer le code avant ou après un appel à proceed pour recréer la notion de before et d'after.

Avec JAC, les wrappers sont définis dans une classe différente de celle de l'aspect. Cela permet de réutiliser les wrappers indépendamment des aspects et *vice-versa.* 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. 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.

Le nom de la classe implémentant un wrapper est fourni au moment de la définition du pointcut (ligne 4 dans le code de la classe TraceAspect voir Figure 1.17). Ici, il s'agit de la classe TraceWrapper, dont le code est le suivant :

```
package aop.jac;

import org.aopalliance.intercept.ConstructorInvocation;
import org.aopalliance.intercept.MethodInvocation;
import org.objectweb.jac.core.AspectComponent;
import org.objectweb.jac.core.Wrapper;
public class TraceWrapper extends Wrapper {
public TraceWrapper(AspectComponent ac) {
super(ac); ¨ ligne 1}
public Object invoke(MethodInvocation mi) throws Throwable {
System.out.println("Avant addItem");
Object ret = proceed(mi); ¨ ligne 2
System.out.println("Après addItem");
return ret; ¨ ligne 3}
public Object construct(ConstructorInvocation ci)
throws Throwable {
return proceed(ci); ¨ ligne 4}}
```

Figure 1.18 Exemple d'un code de Wrapper

Les points de jonction pris en compte par un wrapper sont soit des exécutions de méthode, soit des exécutions de constructeur. Les méthodes *invoke* et *construct* permettent de définir pour ces deux types de point de jonction du code avant et après.

1.3.1.3.2.3 La Composition d'aspect et ordonnancement d'aspect:

Lorsque plusieurs aspects s'appliquent au même point de jonction, on doit définir l'ordre d'exécution des aspects. Cet ordre peut être spécifié dans le fichier descripteur de l'application avec la propriété *jac.comp.wrappingOrder*, qui est une liste ordonnée de classes wrapper. Chaque fois que deux classes wrapper spécifiées dans la liste s'appliquent au même point de jonction, leur ordre d'exécution est celui défini par la liste.

Par exemple, la définition suivante précise qu'AuthenticationWrapper doit être exécutée avant VerboseWrapper:

**jac.comp.wrappingOrder:\org.objectweb.jac.aspects.authentication.AuthenticationWrapper\org.objectweb.jac.wrappers.VerboseWrapper**

Il n'ya pas de règle automatique qui définit quelle classe wrapper doit être exécutée avant une autre; cette décision est à développeur de l'application.

1.3.1.3.2.4 Tissages:

JAC effectue le tissage lors de l'exécution. Les aspects peuvent être ajoutés ou retirés dynamiquement lors de l'exécution. JAC est un tisseur dynamique dont les aspects sont définis en 100% Java; Non seulement le langage de tissage de JAC peut s'exprimer dans deux formats différents (ACC, une syntaxe concise, ou XML) mais son comportement dynamique permet également de tisser ou d'annuler les greffes au cours de l'exécution d'une application. Il est donc particulièrement adapté aux environnements évolutifs dans lesquels nos applications doivent parfois changer de stratégie d'implémentation. La contre partie de cette souplesse se paie bien sûr en termes de performances, car JAC fait un usage intensif de l'API de Réflexion Java, ainsi que sur certaines limitations telles que l'impossibilité d'insérer de nouveaux attributs sur une classe.

Par contre, il est très important de souligner cette initiative, JAC fut le premier tisseur à offrir une bibliothèque d'Aspects techniques réutilisables: de la présentation à la distribution en passant par la gestion transactionnelle et la supervision, JAC nous offre en tout 19 Aspects de qualité professionnelle.

**Exemple illustratif:**

Nous présentons ci-dessous, un exemple d'un code advice issu de la documentation de JAC.

```
1 public class MyWrapper extends Wrapper {
2 // a wrapper must call the Wrapper(AspectComponent)
3 // constructor
4 public MyWrapper(AspectComponent ac) {
5 super(ac); }
6 // A wrapping method must always have this prototype
7 public Object verboseCall (Interaction interaction) {
8 Object ret = null;
9 System.out.println("<< calling "+interaction.method+
10 " with "+interaction.args+" >>");
11 ret = proceed(interaction);
12 System.out.println("<< "+interaction.method+
13 " returned "+ret+" >>");
14 return ret; } }
```

Figure 1.19- le code Advice ou wrapper dans JAC

Un décorateur ou wrapper est équivalent au code advice, représente la partie fonctionnelle d'un aspect. Le décorateur MyWrapper encapsule une fonctionnalité dédiée aux traces. Il est déployé dynamiquement au sein de l'application grâce au composant d'aspect ci-dessous.

```
20 public class Tracing_1_AC extends AspectComponent {
 21 Tracing_1_AC() {
 22 pointcut(".*",".*",".*",MyWrapper.class.getName(),
 23 "verboseCall",false,null);
 24 } }
```

Figure 1.20 Exemple de composant Aspect

Le composant d'aspect Tracing_1_AC (voir Figure 1.20) déploie l'aspect MyWrapper sur toutes les classes de l'application.

1.3.1.3.3 Evaluation:

JAC supporte le tissage dynamique [59] qui intervient durant l'exécution. Il permet donc une plus grande flexibilité (ajout, retrait changement d'aspects) qui a cependant un coût car le tissage dynamique intègre en plus une phase d'adaptation qui consiste à préparer l'application pour quelle puisse recevoir les nouveaux aspects.

La composition des aspects autour d'un point de jonction est configurable, ainsi que la définition des coupes qui est extérieure aux aspects.

JAC [65] ne supporte que deux points de jonctions : *exécution de méthode*, *exécution de constructeur*.

| Point de jonction sur trace (trace pointcut) | Type de tissage : dynamique/Statique | Composition d'aspect | Réutilisation aspect | Spécification du tissage (Application) | Indépendance coupe / aspect | Spécification Coupe | Type de Point de jonction | Type de Conseil | Gestion des Aspects au niveau d'un point de Jonction : Ordre d'exécution des aspects | Type d'advice par point de jonction | Nombre d'advice supporté par un point de jonction |
|---|---|---|---|---|---|---|---|---|---|---|---|
| NON | Dynamique lors de l'exécution. | (possible) Défini avec le système jac.comp.wrappingOrder. | OUI | Le mécanisme RTTI (Run-Time Type Information) | OUI | d'expressions régulières java | exécution de méthode, constructeur | around | Spécifié explicitement on définissant la propriété jac.comp.wrappingOrder | Homogène | illimité |

Tableau 1.3 Bilan sur JAC

### 1.3.2 Les Aspects dans les langages basés sur le composant:

Le composant dans ces approches ne correspond pas à un modèle de composant de l'architecture logicielle. Le composant est en fait un objet tel que les Java BEAN. Les représentants de cette approche sont Aspectual Component, JASco, CAESAR et Open Module.

#### 1.3.2.1 Aspectual Component:

##### 1.3.2.1.1 Introduction :

Aspectual Components [34] [35] a été inspiré partiellement par les idées de la programmation adaptative, basé sur la notion de **collaboration**[6]. Aspectual Components [75] définit une syntaxe étendue de Java pour décrire les connexions (les composants étant les classes Java). Afin d'offrir aux connexions, qui décrivent la sémantique des interactions, un niveau d'abstraction similaire à celui des objets dans un langage orienté objet, les auteurs d'Aspectual Components définissent pour les connexions une entité qui est l'équivalent de la notion de classe du paradigme objet. C'est cette entité qu'ils nomment composant.

Un composant définit un **ensemble de participants formels (participant graph** (**PG**)) à la communication, un ensemble de services requis et un ensemble de services fournis par la communication. Un *participant* énumère un ensemble d'opération qu'il s'attend à ce que d'autres aspects fournissent ou demandent, précédé par le mot-clé **expect**. Des opérations prévues sont employées/modifiées dans la définition spécifique d'Aspectual Component du participant.

L'instanciation du composant est réalisée par le biais d'une entité désignée par le mot-clef connector. C'est cette entité qui effectue la correspondance entre le nom réel des opérateurs et leur dénomination au sein du composant.

##### 1.3.2.1.2 Les Aspects dans Aspectual component:

Les aspects dans Aspectual component sont écrits dans leur propre graphe de classes abstraites se référant à des points de jonction qui constituent l'interface ***Expected*** d'Aspectual component. La détermination des points de jonctions se fait séparément de définition des aspects dans des connecteurs explicites.

---

[6] Une **Collaboration** est un ensemble de classe et un *protocole* déterminant comment les instances de ces classes interagissent. Le protocole définit le *rôle* de l'objet au sein de la collaboration.

1.3.2.1.2.1 Point de jonction et coupe:

L'implémentation d'Aspectual Component fournit quelque abstraction des coupes qui est lié à l'advice. Un connecteur est utilisé pour affiner ces coupes abstraites en coupe concrète.

Dans **Aspectual Components**, les aspects sont définis indépendamment comme un ensemble abstrait de points de jonction. Des **connecteurs** explicites sont alors utilisés pour lier ces points de jonction abstraits avec les points de jonction concrets dans l'application cible. De cette façon, le comportement d'aspect est gardé séparé des composants de bas, même à l'exécution. Un inconvénient possible de cette technique est que le module connecteur nécessite une recompilation après chaque modification.

1.3.2.1.2.2 Les déclarations des advices dans les interfaces d'Aspectual Component:

Dans le contexte [77] d'interface d'Aspectual component, le code Advice et leurs types peuvent être placés à des endroits différents. L'interface d'aspectual component est l'endroit approprié pour déclarer les types des advices. L'implémentation des advices est une partie de l'implémentation de composant aspectuel et non pas de la connexion. Les advices sont complètement séparés des coupes, ce qui facilite la réutilisabilité.

1.3.2.1.2.3 Composition et ordonnancement d'aspect

Les *aspectual components* proposent un modèle simple de composants et de connecteurs qui sont mis en œuvre sous forme de canevas à objets. Cette notion rend explicite, à la façon des modules, les interfaces fournies et requises par un composant [76] .

Ces composants peuvent être considérés comme des aspects et leur composition comme un tissage. Cette première proposition ouvre des perspectives intéressantes quant à la convergence de la programmation par aspects et par composants mais reste focalisée sur une vision structurelle plutôt que comportementale des programmes.

Les *aspectual components* peuvent être programmé en utilisant un langage de programmation Java, car cette approche ne présente pas de nouveaux constructeur de programmation, mais plutôt sont implémentés comme des composants ordinaires. Cependant cette approche n'offre pas un outil pour supporter le travail avec ce modèle.

1.3.2.1.2.4 Le tissage ou Aspect Weaving :

Cette approche, basée sur la programmation par Aspects, permet de rendre la **description des connexions** (les interactions) **indépendante d'un langage de programmation donné**. De plus, les auteurs privilégient l'instanciation d'un connecteur sur le binaire des composants (le byte-code). Ceci permet d'utiliser des connexions avec des composants dont le code source n'est pas disponible et d'**ajouter les connexions à l'exécution** (grâce au mécanisme de chargement dynamique de classes de Java). Les composants dans Aspectual component peuvent être considérés comme des aspects et leur composition comme un tissage.

1.3.2.1.3 Exemple illustratif:

La Figure 1.**21** présente la syntaxe d'Aspectual Component en montrant deux classes (indépendants), **Point** et **FileLogger**, qui serviront de composant de base dans l'exemple suivant:

```
        Point = Class {
    attributes = {
        x : Integer,
        y : Integer     },
methods = {
   setX = method (x : Integer)
      self.x = x
   end,
setY = method (y : Integer) . . .
set = method (x : Integer, y : Integer)
        self.x = x
        self.y = y
end
swap = method (). . .
tostring = method (). . .   }}
```

```
FileLogger = Class {
      attributes = {
         filename : String

 },
    methods = {
       log = method (msg : String)
       file append(filename, msg)
      end,
    }
}
```

Figure 1.21- Un composant de base dans Aspectuel Component

Un composant de base est complètement implémenté ou utilise les méthodes abstraites qui doivent être fournies par des sous-classes d'une classe abstraite de base. La Figure 1.**22** Illustre une collaboration qui implémente le modèle **Publisher/subscribe**r, définissant deux rôles : **Publisher** (Éditeur) et **Subscriber (**abonné**).**

```
PubSub = Collaboration {
   Publisher = Participant {
     expected = {
       changeOp = method () end
   },
    attributes = {     subscribers : List    },
    initialize = method ()
        self.subscribers = List:create()
    end,
methods = { attach = method (sub: Subscriber)      self.subscribers:append(sub)
  end,
detach = method (sub: Subscriber) . . .
 },
replacements = {  changeOp = method () expected () self.subscribers:foreach(     function
(i: Integer, s: Subscriber)
     s:subUpdate(%self)     end) end}},
Subscriber = Participant {      expected = { subUpdate = method (publ: Publisher) end     }
}}
```

Figure 1.22- Une Collaboration dans Aspectual component.

L'éditeur (**Publisher)** fournit la fonctionnalité de maintenir une liste d'abonnés (les abonnés, attache, détachent), tandis que l'abonné (**Subscriber)** déclare une méthode **subUpdate**, par lequel on peut annoncer un changement à l'abonné. Les méthodes **changeOp** et **subUpdate** sont des méthodes requises **(expected methods).** Une méthode **expected ou** requise est liée à une méthode concrète durant le déploiement, qui connecte les **participants** d'une collaboration aux classes d'un composant primitif. Le déploiement est défini par un module qui comporte un ensemble des tracés entre les participants et les classes de base, par lesquels toutes les méthodes **expected** sont liées. Un tel module de déploiement s'appelle un **Connecteur**.

La Figure 1.23 illustre comment la collaboration **PubSub** est appliquée aux classes Point et **FileLogger,** En haut niveau le rôle **Subscriber** est lié à la classe **FileLogger** et le rôle **Publisher** est lié à la classe **Point**. Chaque référence à une classe basse qui doit jouer un rôle donné est encore indiquée par la connexion des méthodes **expected**. Ainsi **FileLogger** dans son rôle **Subscriber** implémente le **subUpdate** en utilisant sa méthode de log() (de FileLogger).

La classe **Point** lie sa méthode expected **changeOp** d'une manière différente :

Le paterne "set." Se réfère à toutes les méthodes dont les noms commencent par **set** suivi d'exactement un caractère arbitraire.

Dans La classe **Point** setX et setY, signifie que ces deux méthodes jouent le rôle de la méthode **changeOp**, c.-à-d., les deux méthodes sont enveloppés par le

remplacement **changeOp** de Publisher. Si à la place {"set. *","swap} ont été employés comme paterne, set et swap serait traité comme **changeOp**.

```
LogSetOneCoordinate = Connector {
Collaboration = PubSub,
  Subscriber = {
 [FileLogger] = {
     subUpdate = method (publ : Point)
     self:log('"set.' on "..publ:tostring())
end } },
Publisher =
 { [Point] = {changeOp = "set." } } }
```

Figure 1.23- Un Connecteur dans Aspectual Component

1.3.2.1.4 Evaluation :

AspectualComponent prouve que la programmation Orientée aspect est plus facile si chaque aspect est programmé dans un modèle générique de données, *un graphe de participant*, qui est séparément connecté à un modèle concret de données ou d'autres. L'utilité de la séparation des composants et des connecteurs est bien connue dans l'architecture logicielle mais n'a pas été jusqu'ici appliquée à AOP.

| Point de jonction sur trace (trace pointcut) | Type de tissage : dynamique/ Statique | Composition d'aspect | Réutilisation aspect | Spécification du tissage (Application) | Indépendance coupe / aspect | Spécification Coupe | Type de Point de jonction | Type de Conseil | Gestion des Aspects au niveau d'un point de Jonction : Ordre d'exécution des aspects | Type d'advice par point de jonction | Nombre d'advice supporté par un point de jonction |
|---|---|---|---|---|---|---|---|---|---|---|---|
| NON | Dynamique (ajouter les connexions à l'exécution) | Par instanciation de connecteur | OUI | A travers des connecteurs | OUI | Expression régulière java | Points de jonction abstraits (selon le langage d'implémentation) | Non spécifié (selon le langage d'implémentation) | Spécifié implicitement selon l'ordre d'application des aspects, | Non spécifié (selon le langage d'implémentation) | illimité |

Tableau 1.4 Bilan sur Aspectual Component

#### 1.3.2.2 JASCO

##### 1.3.2.2.1 Introduction:

Le langage JAsCo[78] (Java Aspect Components) s'inspire principalement d'AspectJ et d'Aspectual Component. JAsco combine l'expressivité de langage de coupe d'AspectJ avec l'idée d'indépendance d'aspect d'Aspectual Component. JAsCo se base essentiellement sur les Java beans. Il introduit deux concepts : les aspects beans et les connecteurs. Un aspect bean[79] regroupe des coupes abstraites, des traitements d'aspects ainsi que des règles de tissage et décrit le où et le quand (les notions d'**advice** et de coupe en AOP). Un des grands apports de JAsCo réside dans son langage de coupe et ses connecteurs qui permettent de définir un contrôle plus fin sur l'ordre des aspects à exécuter

##### 1.3.2.2.2 Les Aspects dans JAsco

JAsCo [79] permet de définir des aspects sous forme **d'aspect bean.** Un aspect bean est un Java bean régulier qui peut déclarer une ou plusieurs classes intérieur appellée **hooks**. Un *hook* est une entité générique et réutilisable. Les hooks peuvent être considérés comme une combinaison d'un pointcut abstrait et d'un advice. Un *hook* peut contenir:

- Un constructeur qui spécifie d'une façon abstraite les conditions de déclenchement d'un hook (des coupes abstraites).
- Une méthode ***isApplicable*** Qui représente une condition supplémentaire de déclanchement d'un hook. ***(***équivalente a l'instruction 'if' de langage Aspectj)***.**
- Un ou plusieurs méthodes advices et des classes Java internes au hook.

```
1 aHook(m1(String s, int i),m2(..args),m3(PrintJob pj, ..args2) {
2 execution(m1) && !(cflow(m2) || withincode(m3));
3 }
```

Figure 1.24 – Exemple d'un constructeur de hook.

Le fragment de code **Figure 1.24** illustre un **Hook** qui déclare trois méthodes abstraites avec des paramètres déférents. Le constructeur [81] indique la condition de déclenchement employant les paramètres de la méthode abstraite. Il devra au moins indiquer le mot-clé **execution**. Ce mot-clé déclare que le hook sera déclenché a chaque liaison au paramètre de la méthode abstrait est exécuté. Il Permet ainsi de décrire l'interception d'un appel de méthode, du point de vue, respectivement, de l'appelant et de l'appelé.

1.3.2.2.2.1 Les coupes et les points de jonctions :

Le langage de coupe et les points de jonctions dans JAsco sont équivalents à ceux de langage AspectJ. Les expressions de point de coupe associées aux Advices soit à l'aide d'annotations java 1.5[7] ou de ***type call*** et ***execution.*** Les coupes sont abstraites et sont définies dans le constructeur du hook. Les références aux méthodes sont définies comme des paramètres dans le constructeur de hook (voir Figure 1. **34 au-dessus**).

Le principe des coupes sur les traces d'exécution n'est pas nouveau. Il porte le nom de *tracematches pointcut* dans la communauté aspect. Les coupes sur les traces d'exécution permettent de définir des coupes plus comportementales, c'est-à-dire capable de capturer un ensemble d'interaction entre composants. La première approche à l'avoir intégré est EAOP (*Event AOP)*, qui repose entièrement sur le principe d'événements. Ensuite JAsco a été étendu pour l'intégrer à *JAsco (stateful aspects)* [80].

1.3.2.2.2.2 Les types d'advices:

JAsco supporte cinq types d'advices [80]: **before**, **after, after throwing, after returning** et **around. Avec** le type **after throwing,** le compilateur insère un appel à l'advice juste après le déclenchement d'une exception, ce qui place l'advice avant la capture de l'exception ou son renvoie au niveau supérieur appelant. Dans le type **after returning,** le conseil est inséré après retour d'un appel de méthode.

1.3.2.2.2.3 Les connecteurs:

Les **beans aspect abstraits** sont déployés dans un contexte de composants concrets par usage de connecteur. Chaque connecteur permet d'une façon explicite d'instancier et d'initialiser un ou plusieurs hooks logiquement liés. Un connecteur permet également de gérer la multiplicité des aspects, définir l'ordre d'exécution des méthodes advices et de résoudre le problème des aspects conflictuels.

---

[7] Les Annotations permettent de marquer différents éléments du langage Java avec des attributs particuliers, dans le but d'automatiser certains traitements et même d'ajouter des traitements avant la compilation grâce au nouvel outil du JDK : Annotation Processing Tool (APT). Les nouvelles annotations Java permettent de simplifier et de structurer l'utilisation des Méta-données.

```
1 static connector PublishUpdates {
2 PublishManager.Publish publish =
3 new PublishManager.Publish(void ComponentX.update*(*));
4 publish.after();}
```

Figure 1.25- Le connecteur JAsco 'PublishUpdates' pour l'édition des Mises à jour

La Figure 1.25 illustre un exemple de connecteur *PublishUpdates* qui instancier le hook *Publish* (**Figure 1.27**), sur les méthodes de mise à jours du composant *ComponentX* ( ComponentX.update*(*))**.** Ce ci est réalisé par le passage de ces méthodes comme wildcards au constructeur de hook (ligne 2-3).

Dans le fragment de code **Figure 1.25**, il est spécifié que le code advice **after ()** de hook **Publish** doit être exécuté toutes les fois que le point de jonction de ce hook est atteint (ligne 4 Publish.after ()), suite a cette déclaration le hook *Publish* est appliqué a toutes les méthodes *update*(*)* de composant component X. Le résultat d'ajout de l'instruction Publish.after() au code de connecteur PublishUpDates est que tous les listeners doivent être notifiés après l'exécution de ces méthodes.

1.3.2.2.2.4 La composition d'aspect:

JAsco laisse la priorité des aspects varies selon les applications. Les connecteurs JAsco sont équipé d'un mécanisme de précédence et des stratégies de combinaison, se qui permet de gérer la multiplicité des aspects et de sélectionner et ordonner les méthodes dans les hooks instanciés.

La stratégie de précédence de JAsco signifie que lorsqu'aucune méthode d'exécution des advices n'est spécifiée dans un connecteur, les méthodes des advices sont déclenchées dans l'ordre dans lequel leurs hooks ont été instancié.

Cette stratégie fournis une solution pour un nombre limité de problèmes d'interaction, que certaines combinaisons aspect complexes nécessitent une manière plus expressive de déclarer comment leur comportement doivent coopérer. JAsco cependant aussi permet de définir une séquence d'exécution explicite en utilisant une stratégie de combinaison d'aspects en implémentant l'interface ICombinationStrategy.

```
1 interface ICombinationStrategy {
2
3 public HookList validateCombinations(HookList);
4}
```

Figure 1.26- L'interface pour stratégie de combinaison d'aspects

1.3.2.2.2.5 Le tissage ou Aspect Weaving :

JASCO offre un tissage dynamique à l'exécution (run-time weaving). Le tisseur *JAsco run-time* [78] optimise le temps d'exécution des advices en générant un fragment de code unique pour chaque point de jonction ce qui réduit le coût dont souffre toute approche dynamique. Avec le tissage dynamique, le lien entre l'application et les aspects est réalisé au moment de l'exécution. Grace à cette capacité de réaliser des tissages dynamiques, il devient aisé de faire évoluer (retrait, changement et ajout de conseil).

1.3.2.2.3 Exemple illustratif:

```
1 class PublishManager {
2
3 // Bookkeeping/notification code
4 void addListener(MethodListener ml) { ... }
5 void notifyListeners(String methodname, Object[]args){.. }
6
7 hook Publish {
8 Publish(topublish(..args)) {
9 execute(topublish);
10 }
11
12 after() {
13 notifyListeners(thisJoinPoint.getMethodName(), args);
14 } } }
```

Figure 1.27- Un exemple d'Aspect Bean éditeur 'PublishManager'

Dans le listing de la Figure 1.26, l'Aspect Bean **PublishManager** contient un certain nombre de méthodes standard pour contrôler et informer les listeners ou les objets écouteurs (lignes 3-5). Un listener est un objet qui possède au moins une méthode qui pourra être invoqué si l'événement attendu apparaît. Le hook **Publish** (lignes 7-14) qui est responsable d'invoquer la notification après l'exéution d'une méthode appropriée. Le hook à un ou plusieurs constructeurs qui indiquent dans une manière abstraite quand son comportement devrait être déclenché, et une ou plusieurs méthodes d'advices (before, around, after…) qui indiquent quels comportements doit être exécuté. L'advice after() (lignes 12-14) indique alors qu'après cet événement, les listeners devraient être informé du nom de méthode et des arguments.

1.3.2.2.4 Evaluation:

JAsCo fait partie des approches asymétriques traitent les aspects et les composants du système de façon différente. Il possède une très bonne adaptation au contexte et aux changements imprévus. Cependant, la plupart de ces changements nécessitent d'écrire et de compiler un nouveau connecteur, ce qui cause un encombrement important et un risque d'erreur. De plus, le caractère dynamique et flexible de JasCo entraîne une dégradation considérable de performances.

JAsCo permet de définir des aspects indépendamment de leurs futurs contextes de réutilisation. Les connecteurs et les stratégies de combinaison permettent eux de déployer dynamiquement et convenablement les aspects à l'exécution. De plus, JAsCo supporte le modèle de composant des *java beans* en permettant aux *connecteurs* d'instancier des *hooks* sur des évènements propre aux *beans*. Puisque les **aspects beans** sont décrits indépendamment d'un contexte spécifique, ils peuvent être réutilisés et appliqués sur une variété de composants.Malgré que JAsCo soit un modèle hybride entre objets et composants, ses capacités d'adaptation sont limitées à l'interception de méthodes. L'un des avantages de JasCo est qu'il offre des mécanismes permettant d'ordonnancer les comportements des aspects conflictuels ainsi que des stratégies de combinaison explicites et réutilisables, en vue de remédier aux problèmes d'interactions. Enfin il est à noter que JAsCo supporte le déclenchement d'aspects sur une séquence de points de jonction. JasCo.

| Point de jonction sur trace (Tracecuts) | Type de tissage : dynamique/ Statique | Composition d'aspect | Réutilisation aspect | Spécification du tissage (Application) | Indépendance coupe / aspect | Spécification Coupe | Type de Point de jonction | Type de Conseil | Gestion des Aspects au niveau d'un point de Jonction : Ordre d'exécution des aspects | Type d'advice par point de jonction | Nombre d'advice supporté par un point de jonction |
|---|---|---|---|---|---|---|---|---|---|---|---|
| **OUI** | **Dynamique A l'exécution** | **(possible) stratégie de précédence/ stratégie de combinaison d'aspects** | **OUI** | **A travers le JAsco run-time weaver** | **NON** | **(d'expressions régulières Java)** | **Execution call, des Annotations java 1.5.** | **Before, After , after throwing , after returning Around** | **Spécifié explicitement on spécifiant une stratégie de combinaison d'aspects** | **Hétérogène** | **illimité** |

Tableau 1.5 Bilan sur JAsCo

1.3.2.3 Caesar:

1.3.2.3.1 Introduction:

**Caesar** est une extension de Java et d'AspectJ modifiant en particulier le modèle AspectJ pour le rendre plus souple. Caesar permet de créer facilement une instance d'aspect pour chaque objet impliqué dans un rôle. Cette instance modifie le comportement de l'objet en conséquence.

Caesar [19] transpose essentiellement le modèle des coupes d'AspectJ aux EJB[8] incluant, par exemple, des coupes pour l'accès aux valeurs des champs et des coupes sur le flot de contrôle. Il produit un *bytecode* totalement compatible avec la machine virtuelle de Java. Le modèle sur lequel il s'appuie utilise des « Composants Aspectualisés ».

1.3.2.3.2 Les Aspects dans Caesar:

Caesar [81] propose *des interfaces de collaboration encapsulées par la notion d'aspect* : un *aspect* est un ensemble *d'interfaces de collaboration*. Chaque *interface de collaboration* possède des services requis, des redéfinitions d'implémentation et des services locaux. Les trois parties constituent tout l'aspect.

[8] Entreprise Java Bean

La partie abstraite d'aspect dans Ceasar est [83] défini dans son interface de collaboration d'aspect (Aspect Collaboration Interface ou ACI). Caesar utilise ces interfaces de collaboration pour lier aspects ET composants. Cette ACI établit un protocole de communication bidirectionnel entre l'implémentation de l'aspect et les connexions entre ses aspects.

1.3.2.3.2.1 Les coupe/ pointcut et les déclarations des advices:

Caesar définit [81] des méthodes advices et des coupes. Les coupes sont définies dans des descripteurs (wrappers), c'est-a-dire en dehors de la définition des composants, et ne nécessitent donc pas d'accès à l'implémentation des composants. La déclaration des points de jonction est explicite (voir Figure 1.42 *ligne 35 a ligne37).* dans cet exemple Ceasar défini différents coupe pour *un Point* p et une *ligne* l.

1.3.2.3.2.2 Le tissage d'aspect:

Caesar génère des classes de proxies lors du déploiement. On outre Caesar propose de composer un aspect (son implémentation et sa connexion) en une nouvelle unité appelée *weavelet* pour l'activation des aspects.

Un weavelet est une nouvelle classe dans lequel les implémentations respectives des méthodes requises et les méthodes fournies des modules implémentation et connexion sont composés. Un weavelet doit être déployé afin d'activer les coupes et les advices. Un déploiement weavelet est syntaxiquement désigné par le mot *deploy*. Le déploiement peut être statique (au moment du chargement) ou dynamique.

*Steamloom* [85] est une machine virtuelle java qui implémente le modèle CAESAR pour supporter le déploiement dynamique. Alors que CaesaeJ [85] [86] ne supporte que le déploiement statique.

1.3.2.3.2.3 La composition d'aspect:

Caesar fournit des mécanismes pour le problème d'ordonnancement des aspects, le premier permet l'ordonnancement arbitraire entre aspects à l'aide de ses interfaces de collaborations, le second fournit des analyses statiques d'interactions entre aspects et un jeu correspondant d'opérateurs de compositions pour la résolution de conflits identifié.

1.3.2.3.3 Exemple illustratif:

Caesar utilise des interfaces de collaboration pour lier aspects et composants. Les *aspects* définissent des extensions orthogonales (modélisées

par des *interfaces de collaboration*) au reste du système. Nous illustrons ces notions dans les exemples présentés dans les Figure 1.s 28,29 et 30 en dessous.

```
1 interface OserverProtocol {
2 interface Subject {
3 provided void addObserver(Observer o);
4 provided void removeObserver(Observer o);
5 provided void changed();
6 excepted String getState();
7 }
8 interface Observer {
9 excepted void notify(Subject s);
10 }}
```

Figure 1.28. Exemple d'une interface de collaboration avec Ceasar tiré de [81]

L'aspect ObserverProtocol est défini par deux *interfaces de collaboration* Subject et

Observer qui définissent les deux rôles du patron de conception observateur. Ces deux interfaces seront liées à une (des) classe(s) existante(s) au moment de la connexion de l'aspect. Cette connexion mettra en relation des méthodes de la classe avec les méthodes requises (dénotées par le mot clef **excepted)**. De plus, elle mettra au service de la classe les méthodes fournies (dénotées par le mot clef **provided**).

Les méthodes que les aspects fournissent aux interfaces de collaboration peuvent avoir plusieurs implémentations. Le choix d'une implémentation particulière se fait au moment du déploiement des aspects (c'est-à-dire pendant la dernière phase). Ci-dessous nous donnons un exemple d'une implémentation possible de l'aspect ObserverProtocol.

```
12 Class ObserverProtocolImpl implements ObserverProtocol {
13 Class Subject {
14 List observers = new LinkedList();
15 void addObserver(Observer o) { observers.add(o); }
16 void removeObserver(Observer o) { observers.remove(o); }
17 void changed() {
18 Iterator it = observers.iterator();
19 while ( it.hasNext() )
20 ((Observer)iter.next()).notify(this);}} }
```

Figure 1.29. Exemple de l'implémentation d'un l'aspect.

Pour réutiliser un aspect, il faut définir des connexions entre lui et les classes qui sont concernées. Ci-dessous, on trouve un exemple de connexion de l'aspect ObserverProtocol.

```
24 Class ColorObserver binds ObserverProtocol {
25 Class PointSubject binds Subject wraps Point {
26 String getState() {
27 return "Point colored" + wrappe.getColor();}}
29 Class LineSubject binds Subject wraps Line {
30 String getState() {
31 return "Line colored" + wrappe.getColor(); } }
32 Class ScreenObserver binds Observer wraps Screen {
33 void notify(Observer s) {
34 wrappe.display("Color changed" + s.getState()); } }
35 after(Point p) : (call(void p.setColor(Color)))
36 { PointSubject(p).changed(); }
37 after(Line l) : (call(void l.setColor(Color)))
38 { LineSubject(l).changed(); } }
```

Figure 1.30. Exemple de connexion de l'aspect

1.3.2.3.4 Evaluation:

Caesar utilise un modèle de programmation par aspects (comme AspectJ par exemple). Il possède deux propriétés : l'éclatement total des aspects et le déploiement statique ou dynamique des aspects.

*L'éclatement total* des aspects consiste à découpler à la fois la définition de l'aspect, son l'implémentation, ses connexions et son déploiement. Ce découplage présente l'avantage de pouvoir changer, pendant le déploiement, l'implémentation de l'aspect utilisée. Par contre, il diminue sérieusement la lisibilité de ce dernier.

L'implémentation de Caesar n'est pas encore complète, seul le système de type est réalisé. En ce qui concerne des éléments de généricité, il est possible de faire dépendre l'implémentation d'un aspect par des coupes abstraites et des valeurs de champs (AspectJ2EE) et d'utiliser la liaison tardive d'action d'un aspect en exploitant le polymorphisme de type entre interfaces de collaboration de Caesar. Une fois les aspects définis selon les modèles étendus se pose la question de leur tissage. Caesar génère des classes de proxies.

| | |
|---|---|
| Point de jonction sur trace | NON |
| Type de tissage : dynamique/ Statique | Dynamique A l'exécution Ou statique. Selon le modèle d'implémentation |
| Composition d'aspect | (possible) Programmable a l'aide de ses ACI/ analyses statiques |
| Réutilisation aspect | OUI |
| Spécification du tissage (Application) | génère des classes de proxies / (weavlet) |
| Indépendance coupe / aspect | OUI |
| Spécification Coupe | (d'expressions régulières Aspectj) |
| Type de Point de jonction | Explicit/ pas d'introduction |
| Type de Conseil | Selon le modèle d'implémentation (dans CaesarJ même type qu'aspectJ) |
| Gestion des Aspects au niveau d'un point de Jonction : Ordre d'exécution des aspects | Spécifié explicitement on définissant un ordonnancement arbitraire entre aspects à l'aide de interfaces de collaborations, |
| Type d'advice par point de jonction | Hétérogène |
| Nombre d'advice supporté par un point de jonction | illimité |

Tableau 1.6 Bilan sur Caesar

1.3.2.4 Open module :

1.3.2.4.1 Introduction:

Les systèmes de programmation orienté aspect fourni de puissants mécanismes de séparation des préoccupations, mais comprendre comment interagissent ces préoccupations peut être difficile. En particulier, plusieurs approche de programmation orientée aspect peuvent violer l'encapsulation, créant des dépendances entre les préoccupations qui rendent l'évolution des logiciels une opération délicate, difficile et sujette à l'erreur.

Les techniques de programmation modulaire sont fondées sur la distinction entre interface et implémentation, l'encapsulation de l'implémentation et la notion d'interface en tant que contrat du module. Les bonnes pratiques du génie logiciel accompagnent ces techniques. Le critère de *masquage de l'information* dans le processus de décomposition modulaire est l'exemple essentiel de ces bonnes pratiques, qui ne peuvent être complètement formalisées dans les langages de programmation mais influencent leur usage.

Malheureusement, les développeurs ne respectent pas toujours le critère de *masquage de l'information*, les limites de modules sont souvent atteintes pour des

raisons pratiques temporaires telles la dispersion et l'éparpillement de l'information dans le système ainsil'enchevêtrement avec le code de base.

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

L'utilisation de ce système de modules permet de préserver le contenu d'un module en déclarant explicitement les points de variations sur lesquels les aspects peuvent agir.

1.3.2.4.2 Les Aspects dans Open module

Aldrich propose les *Open Modules* [73]. L'idée des modules ouverts (*Open Modules*) est d'ouvrir un programme à l'approche par aspects tout en gardant la modularité en cachant les détails d'implantation d'un module. Un module est défini comme un ensemble d'entités qui partagent des points d'accès qui sont des points de jonction exportés par le module. L'utilisation de ce système de modules permet de préserver le contenu d'un module en déclarant explicitement les points de variations sur lesquels les aspects peuvent agir.

Ce système de modules rétablit l'encapsulation et intègre les coupes à l'interface du module visé. Ces coupes capturent les points de jonction internes au module et sont exposées dans l'interface. Un aspect extérieur a accès aux éléments de cette interface, appelle des méthodes et des coupes déclarées mais pas aux points de jonction non exposés. Un module est un regroupement de classes associé à une définition de points d'accès. Cette définition est l'interface du module. Un point de jonction devient accessible lorsqu'il est déclaré comme point d'accès d'un module.

L'interface d'un module peut donc déclarer [69] : des points de jonction, des fonctions, ou des coupes prédéfinies sur un ensemble d'éléments internes. Un module vérifie les propriétés définies ci dessous,

- **Règle 1** Des aspects extérieurs à un module peuvent interagir entre ce module et le monde extérieur incluant des appels extérieurs à des fonctions de l'interface d'un module.

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

1.3.2.4.2.1 point de jonction et coupe:

L'objectif des modules ouverts est de limiter l'accès aux points de jonction d'un système, qui sont atteint par les aspects rendant ces derniers envahissants, c'est-à-dire capable d'accéder à des attributs privés et de briser l'encapsulation des objets.

Open modules supporte qu'un seul type de points de jonctions de type ***call*** [69] et que les points de coupes statique**,** cependant l'implémentation de Open modules avec AspectJ permet de définir des points de coupes dynamique de type ***cflow.***

Open Modules [87] fait une distinction entre les appels internes et externes aux modules. Lorsque un aspect dépend d'un appel interne au module, on ajoute simplement un point de coupe a l'interface de module de tel façon le nouveau aspect reste conforme aux règles définies par le modèle Open modules.

Open modules [70] donne plus d'adaptation expressives possible par offrir une abélite d'exposer d'une façon explicite les coupes (pointcuts) dans l'interface de composant.

Aldrich dans [69] propose un nouveau langage pour implémenter l'approche *Open Modules* qui s'appelle *TinyAspect* qui assure la sûreté et préserve les principes l'encapsulation présentés dans l'approche Open modules (voir Figure 1.31).

Names n ::= x
Expressions e ::= n | fn x:τ => e | e1 e2 | ()
Declarations d ::= • | val x = e d | pointcut x = p d | around p(x:τ ) = e d
Pointcuts p ::= n | call(n)
General exp. E ::= e | d | p
Types τ ::= unit | τ1 → τ2
Decl. Types β ::= • | x:τ, β | x:π, β
Pcut. types π ::= pc(τ1 → τ2)
General types T ::= τ | β | π

Figure 1.31.Un extrait de syntaxe de langage *TinyAspect* [69].

1.3.2.4.2.2 Les déclarations des advices:

Open Modules permet de mettre des advices sur toutes les déclarations de fonctions dans l'interface d'un module se qui fournis plus d'extensibilité. La définition des méthodes Advices dans Open modules est similaire au langage AspectJ, afin de faciliter l'application de se modèle formel dans les autres langages de paradigme aspect.

Open modules est le premier système de module qui supporte de nombreuses utilisations des advices, tout en assurant à ce que la sécurité et d'autres propriétés sont maintenus. Les advices externes ne peuvent pas affecter les appels internes au sein du module, il est donc facile à raisonner sur le contrôle et les flux de données dans le module. Bien que les clients puissent mettre des advices sur les coupes exportés, ces advices peuvent être traité comme un rappel à une fonction cliente spécifié, et les techniques de raisonnement standard peuvent être utilisées.

1.3.2.4.2.3 Le tissage ou Aspect Weaving :

Open modules est modèle formel limité à un petit noyau de langage de programmation orienté aspect, cependant il offre un mécanisme non seulement pour la séparation des préoccupations dans un système mais qui renforce la modularité et l'encapsulation entre les différents modules de se système.

N. Ongkingco1 et al. [88] Propose une extension d'AspectJ pour intégré l'approche Open modules afin de palier aux problème d'accès aux points de jonction d'un système par les advices. Le modèle Open modules fait partie de la version 2,0 de compilateur *abc* (Aspect Bench Compiler) pour AspectJ. Le tissage dans se cas est statique a la compilation.

1.3.2.4.2.4 La composition d'aspect:

Un mécanisme similaire à AspectJ est le principe de précédence entre les aspects, pourrait être utilisé pour préciser si les aspects internes s'appliquent avant ou après les aspects externes. Le problème de la sémantique d'interaction reste difficile pour les aspects définis dans le même module de haut niveau. Beaucoup des outils, tels que le plugin AspectJ pour Eclipse, aide au problème d'interaction sémantique en montrant une vue de l'endroit où les aspects s'appliquent au code de base.

1.3.2.4.3 Exemple illustratif:

La Figure 1.32 montre une vue conceptuelle de open modules. Comme les systèmes ordinaires des modules, Open modules exporte une liste de données structurés et des fonctions telles que *moveBy* et *animer*. Cependant, open modules peut exporter des coupes indiquant les événements sémantiques internes. Par exemple, la coupe ***moves*** à la Figure 1.40 est déclenchée quand une forme (*Shape)* se déplace.

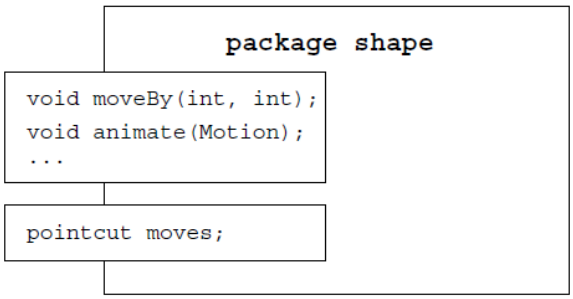



Figure 1.32. Une vue conceptuelle de Open modules.

```
structure shape = struct
val createShape = fn ...
val moveBy = fn ...
val animate = fn ...
...
pointcut moves = call(moveBy)
end :> sig
createShape : Description -> Shape
moveBy : (Shape,Location) -> unit
animate : (Shape,Path) -> unit
...
moves : pc((Shape,Location)->unit)
end
```

Figure 1.33. La spécification de module "Shape" avec *TinyAspect* [69]

1.3.2.4.4 Evaluation:

Open Modules augmente la lisibilité d'un système par rapport aux systèmes existants de programmation orientée aspect, parce que l'implémentation et le comportement d'un module peut être entièrement comprise dans l'isolement.

Le programmeur du module visé par l'aspect doit donc garantir la définition et la maintenance des coupes de l'interface, ce qui est en opposition avec le principe de transparence. Cependant, si un aspect est intéressé par plusieurs modules, chacun de ces modules devra par construction déclarer la coupe adéquate : on observe donc une forme (plus légère) de dispersion.

Comparé aux autres systèmes AOP, Open modules rend plus facile de prédire les résultats apporter de petits changements à l'implémentation de système. Parce que Open modules renforce l'encapsulation, les changements au code dans un module peuvent affecter directement le code et d'autres aspects dans ce même module. Tant que la sémantique de l'interface du module est préservée, le code externe et les aspects ne seront pas touchés.

| | |
|---|---|
| **Point de jonction sur trace (Tracecuts)** | **NON** |
| **Type de tissage : dynamique/ Statique** | **Selon le langage d'implémentation Statique à la compilation dans AspectJ** |
| **Composition d'aspect sur le même point de jonction** | **(Possible) stratégie de précédence** |
| **Réutilisation aspect** | **OUI** |
| **Spécification du tissage (Application)** | **Selon le langage d'implémentation: Machine virtuel java dans aspectJ** |
| **Indépendance coupe / aspect** | **NON** |
| **Spécification Coupe** | **Expression régulière dans *TinyAspect* / expression java ordinaire dans AspectJ** |
| **Type de Point de jonction** | ***call*** |
| **Type de Conseil** | **Before After Around** |
| **Gestion des Aspects au niveau d'un point de Jonction : Ordre d'exécution des aspects** | **Spécifié explicitement on définissant une stratégie de précédence entre les aspects** |
| **Type d'advice par point de jonction** | **Hétérogène** |
| **Nombre d'advice supporté par un point de jonction** | **illimité** |

Tableau 1.7 bilan sur Open Modules

### 1.3.3 Aspect et architecture logicielle :

En Architecture Logicielle, la spécification au niveau des interfaces de composants d'un besoin explicite en aspect technique (i.e. sécurité, persistance, traçage) rendaient ces composants réutilisables dans un contexte qui doit être pourvu explicitement de ces aspects techniques. Plus encore, si une architecture, qui est un composant composite, aurait été défini avec un aspect technique, cette architecture, ou composant composite, devrait être instancié avec ce même aspect technique, interdisant de ce fait tout changement. Cette situation est du principalement au concept d'encapsulation des composant à travers lequel la vue interne d'un composant composite fourni comme élément de bibliothèque, n'est pas du tout accessible. Cette situation réduit ainsi la réutilisation du composant. De tels composants ne peuvent pas être réutilisés indépendamment. Pour pallier à ce problème, une autre tendance en architecture logicielle est née : C'est l'Architecture Logicielle Orientée Aspect. Cette partie du chapitre étudie l'unification des aspects et des composants logiciels et fait le point sur l'intégration des aspects dans les approches d'architecture logicielle. Actuellement deux grandes tendances existent en Architecture Logicielle Orientée Aspect : Les approches dites symétriques et les approches asymétrique. Dans les approches symétriques un même modèle de composant est utilisé pour représenter les aspects techniques ou le métier. C'est le mécanisme d'assemblage des composants qui positionnera ces derniers composants comme étant des composants métier ou des composants aspect. Les approches symétriques sont représentées par Aspectual ACME et AO-ADL.

Les approches asymétriques sont plus nombreuses. Ces approches sont caractérisées par deux éléments fondamentaux : Premièrement, ils introduisent les aspects par adaptation d'une approche existante d'Architecture Logicielle. Ceci est aussi le cas dans certaines approches symétrique comme Aspectual ACME. Deuxièmes, l'adaptation se fait par l'ajout de nouveau elements orientés aspect pour l'approche adaptée. L'asymétrie se manifeste par le fait de distinguer explicitement les composants orientés aspects et les composants métier. Les composants et les aspects reposent sur des modèles différents et peuvent avoir de ce fait une structure différente. Les propositions les plus en vue dans ce contexte sont TranSAT, FAC, CAM/DAOP-ADL, AspectLEDA et AC2-ADL.

Les approches asymétriques:

1.3.3.1 TranSAT ( Transformation for Software Architecture Technologies):

1.3.3.1.1 Introduction :

TranSAT est une extension de l'approche d'Architecture Logicielle *SafArchie* **qui e**st un environnement pour la construction d'architectures logicielles validées. SafArchie dispose d'un modèle de composant léger et hiérarchique.

L'objectif de **SafArchie** est de définir un modèle abstrait de composant en vue de valider un assemblage de composants par rapport à des propriétés structurelles et comportementales.

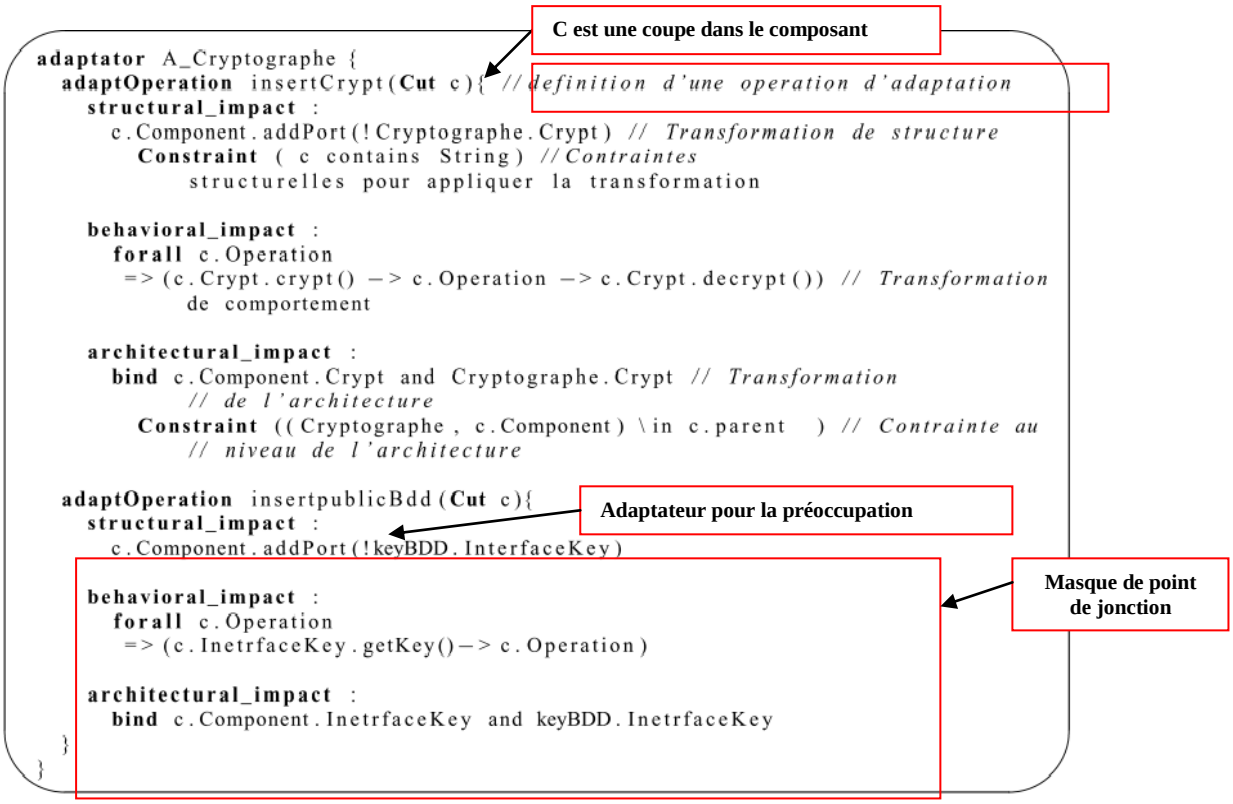
Le concept de préoccupation dans TranSat diffère du composant par une granularité plus importante. **Une préoccupation est toujours fournie avec son *adaptateur*** afin de faciliter son intégration. La Figure 1.34 présente la description d'une préoccupation de confidentialité à l'aide de deux composants : une base de données de clés publiques et un composant de cryptographie.

```
component Cryptographe {
  Port Crypt {
    provide String crypt(String pbkey, String message);
    provide String decrypt(String ptkey, String encrypt_message);
  }
}
component KeyBDD {
  Port InterfaceKey {
    provide String getkey(int uid);
  }
}
```

Figure 1.34. Préoccupation de confidentialité

Pour supporter l'orienté aspect TranSAT ajoute deux nouveaux concepts à SafArchie : **L'adaptateur et le Tisseur.**

L'*adaptateur* définit les règles d'intégration des composants techniques dans une architecture logicielle générique indépendamment du contexte d'intégration. L'adaptateur a une structure relativement proche du composant. Il est composé *d'opérations d'adaptation*.




```
adaptator A_Cryptographe {
  adaptOperation insertCrypt(Cut c){ //definition d'une operation d'adaptation
    structural_impact :
      c.Component.addPort(!Cryptographe.Crypt) // Transformation de structure
        Constraint ( c contains String) //Contraintes
            structurelles pour appliquer la transformation

    behavioral_impact :
      forall c.Operation
       => (c.Crypt.crypt() -> c.Operation -> c.Crypt.decrypt()) // Transformation
            de comportement

    architectural_impact :
      bind c.Component.Crypt and Cryptographe.Crypt // Transformation
            // de l'architecture
        Constraint ((Cryptographe , c.Component) \in c.parent  ) // Contrainte au
            // niveau de l'architecture

  adaptOperation insertpublicBdd(Cut c){
    structural_impact :
      c.Component.addPort(!keyBDD.InterfaceKey)

    behavioral_impact :
      forall c.Operation
       => (c.InetrfaceKey.getKey()-> c.Operation)

    architectural_impact :
      bind c.Component.InetrfaceKey and keyBDD.InetrfaceKey
  }
}
```

Figure 1.35. Un adaptateur pour la préoccupation de confidentialité

La Figure 1.35 décrit un exemple d'adaptateur pour l'intégration de la préoccupation de confidentialité. Il possède deux opérations d'adaptation. L'une décrit l'intégration du composant *Cryptographe*, l'autre modélise les modifications à apporter pour intégrer la base de données de clés publiques.

Le *tissage* assemble un adaptateur à une architecture. Il décrit l'interaction entre une architecture logicielle spécifique et une préoccupation technique. L'opération de tissage offre aussi l'avantage d'inverser les dépendances, en effet contrairement à une opération d'assemblage, le composant modifié ne doit pas nécessairement offrir d'interfaces compatibles avec les composants réalisant la préoccupation.

Avec TranSAT on tisse les composants avec des nouvelles préoccupations. Chaque préoccupation est fournie avec son adaptateur qui décrit les règles de transformation de l'architecture. Ces règles sont alors utilisables par un moteur de transformation qui pourra intégrer automatiquement la préoccupation.

Contrairement à l'adaptateur, le tisseur est dépendant du contexte. Il précise les éléments suivants :

- La préoccupation que l'on cherche à intégrer, celle-ci sera définie en fonction de l'adaptateur que nous incluons dans le tisseur.

- La définition des différentes coupes transversales. Une coupe est un ensemble de points de jonction. Le tisseur va permettre de lier chaque coupe avec une opération d'adaptation.
- Les opérations de tissage, c'est-à-dire le lien entre les coupes définies au sein du tisseur et les opérations d'adaptation de l'adaptateur.
- Le renseignement des différentes métadonnées définies au sein de l'adaptateur.

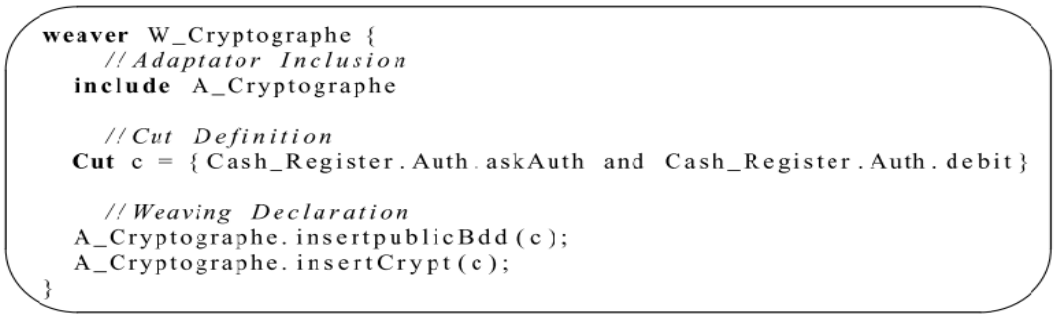



Figure 1.36. Un tisseur pour intégrer la préoccupation de confidentialité au sein d'une Station service

La Figure 1.36 présente l'exemple du tisseur pour l'intégration d'une préoccupation de confidentialité sur la station.

TranSAT propose de gérer les évolutions d'une architecture logicielle décrite sous forme d'une architecture à base de composants suivant le modèle proposée par SafArchie. Il contient [27] [18] [28]:

i. Un assemblage de composants, appelé *plan,* qui prend en charge les fonctionnalités nouvelles liées à la préoccupation à intégrer,

ii. Une description des éléments devant être présents dans l'architecture de base afin de pouvoir tisser le plan, appelé *masque de point de jonction* et

Un ensemble de *règles de transformation* qui définissent les modifications à apporter sur le plan initial pour l'intégration de la préoccupation.

| TranSAT | AOP |
|---|---|
| ▪ plan de base<br>▪ patron d'architecture<br>▪ nouveau plan d'architecture<br>▪ masque de coupe<br>▪ règles de transformations | ▪ programme de base<br>▪ préoccupation<br>▪ advice<br>▪ coupe paramétrable<br>▪ tissage |

Tableau 1.8 Comparaison entre TranSAT et l'AOP

1.3.3.1.2 Les Aspects dans TranSAT:

1.3.3.1.2.1 Le plan :

Le premier type d'information que l'on trouve au sein du patron d'architecture est le plan à intégrer. Ce plan [28] est un assemblage de composants construit à l'aide de *SafArchie*. Cet assemblage a la particularité de ne pas être nécessairement complet. Ainsi, certains ports peuvent être non connectés. En outre, deux composants liés du plan peuvent être intégrés dans des composites différents.

1.3.3.1.2.2 Masque de point de jonction :

Le masque de point de jonction décrit pour un patron d'architecture, les hypothèses faites sur le plan de base pour la spécification de la transformation. Il décrit un ensemble de contraintes sur le contexte architectural dans lequel la préoccupation peut venir s'intégrer. Il peut être abordé comme un système de typage des points de jonction. En effet, si par défaut, un patron peut venir modifier n'importe quel élément d'une description d'architecture logicielle, le masque de point de jonction permet de définir le type du lieu d'intégration sur le quelle patron va venir s'accrocher.

1.3.3.1.2.3 Règles de transformation :

Les règles de transformation spécifient les opérations à effectuer afin d'intégrer le plan au sein d'une architecture logicielle existante. Elles sont décrites uniquement à partir des éléments du masque de point de jonction et du plan afin de conserver la propriété d'indépendance du patron par rapport au contexte d'intégration.

Les règles de transformation spécifient comment tisser un nouveau plan et un plan de base en fonction d'un lieu d'intégration respectant les contraintes définies au niveau du masque de point de jonction. Il existe deux grandes familles:

- Les primitives de reconfiguration : Manipulent les éléments liés à la configuration de l'architecture
- Les primitives d'introduction: Manipulent les éléments liés au fonctionnement d'un composant
  - Modification structurelle d'un composant => modification du comportement

**Structurelle** : ajout d'une opération au sein d'un port

**Comportementale** : modification de la séquence d'exécution

- Introduction = modification structurelle + modification comportementale

1.3.3.1.2.4 Composition d'aspect:

Le principal point fort de TranSAT réside dans une bonne expressivité de l'approche pour la gestion de la composition de plans d'architecture. En effet, TranSAT fournit tout d'abord un mécanisme de point de coupe avec deux niveaux d'abstraction : une description abstraite de la coupe à l'aide du masque de point de jonction qui est spécialisée à l'aide d'une expression de coupe dépendante du contexte d'intégration au moment du tissage. Ce langage propose alors le moyen de tisser le même aspect sur plusieurs lieux d'intégration d'un plan de base à l'aide d'un langage expressif.

Contrairement [48] à de nombreuses approches, TranSAT épargne à l'architecte la spécification complètement manuelle de la liaison entre un plan de base et la fonctionnalité à intégrer et permet de spécifier des relation *n-aires* entre une préoccupation à intégrer et le plan de base. De plus, TransAT fournit un langage de transformation spécifique au domaine de l'architecture logicielle. Ce langage offre à l'architecte un haut pouvoir d'expression quand il souhaite spécifier l'intégration d'une préoccupation au sein d'un plan de base.

Ce langage permet en outre de conFigure 1.r finement la composition d'un plan en charge d'une fonctionnalité avec un plan de base. Cependant, TranSAT ne propose pas de mécanisme avancé pour gérer la composition de multiples patrons sur un même plan de base. L'ordre est pour le moment spécifié de manière absolue et il n'existe pas de mécanisme pour détecter l'absence de conflits entre patrons.

1.3.3.1.3 Evaluation :

Face à la complexité algorithmique concernant la recherche exhaustive de tous les lieux d'intégration compatibles avec un masque de point de jonction, TranSAT ne fournit pas de mécanismes efficaces permettant de trouver tous ces *lieux d'intégration* ni de langage de coupe permettant de sélectionner un sous-ensemble de lieux d'intégration compatibles. La définition du lieu d'intégration reste donc pour le moment manuelle dans TranSAT.

L'architecte associe l'ensemble des éléments du masque de point de jonction avec des éléments du plan de base qu'il souhaite modifier. Ce manque d'assistance à ce moment précis du processus de transformation n'est néanmoins pas dommageable. En effet, dans un processus de construction incrémentale, on peut aisément considérer que l'architecte, quand il choisit le patron, a déjà une certaine idée du ou des lieux d'intégration qu'il souhaite modifier.

TranSAT fournit alors un mécanisme d'analyse statique afin de vérifier quels lieux d'intégration choisis respectent l'ensemble des contraintes définies au niveau du masque de point de jonction.

TranSAT définit une coupe générique et ne précise pas clairement avec quel composant on doit tisser la préoccupation c'est à l'architecte ensuite de faire le lien et de préciser le composant qui répond aux contraintes définies dans l'adaptateur de préoccupation à tisser.

L'adaptateur dans TranSAT permet de capitaliser les modifications à apporter sur une architecture cible, c'est le rôle de l'architecte de conFigure 1.r cette intégration. TranSAT permet de modéliser la préoccupation et la façon de l'intégrer grâce aux composants et à l'adaptateur et l'architecte conFigure 1.son intégration au sein du tisseur.

La liaison entre le plan à intégrer et le plan de base est définie en sélectionnant des points de jonction sur le plan de base. De nombreux fragments de ce plan de base respectant le masque de points de jonction sont des points de jonction potentiels pour la préoccupation prise en charge par le patron d'architecture. La sélection de ces points de jonction est appelée *la coupe* par analogie avec le vocabulaire que l'on trouve dans le domaine de la programmation par aspects. Dans TranSAT, de par la nature complexe de ces points de jonction qui peuvent être constitués de nombreux éléments de l'architecture logicielle à transformer, il a été choisi d'appeler ces points de jonction des *lieux d'intégration* [48] . Ce nouveau nom vise à faire la différence entre un point de jonction au sens AspectJ qui représente un point dans le flot d'exécution d'un programme et un lieu d'intégration qui représente un fragment de modèle d'architecture logicielle à partir duquel TranSAT va venir tisser un nouveau plan.

Le résultat d'une transformation est une nouvelle architecture logicielle contenant l'ensemble des éléments du plan de base et du nouveau plan. Cette

architecture peut alors servir à nouveau de plan de base pour l'intégration d'une nouvelle architecture

Au niveau de la mise en oeuvre, TranSAT se fonde sur SafArchie qui permet de générer du code vers la plate-forme à composants Fractal ou le langage ArchJava. Pour cette transformation, SafArchie se place sur le modèle d'architecture résultant, au niveau de la mise en oeuvre, il n'y a pas de conservation de la structure de l'architecture en préoccupation.

| | |
|---|---|
| **Point de jonction sur trace (Tracecuts)** | **NON** |
| **Type de tissage : dynamique/ Statique** | **Tissage d'aspect revient à établir le plan de base et le plan décrivant la préoccupation à intégrer** |
| **Composition d'aspect** | **possible** |
| **Réutilisation aspect** | **OUI** |
| **Spécification du tissage (Application)** | **Sous forme d'un ensemble de règles de transformation** |
| **Indépendance coupe / aspect** | **OUI** |
| **Spécification Coupe** | **SafArchie ADL** |
| **Type de Point de jonction** | **Introduction et reconfiguration** |
| **Type de Conseil** | **beforeCall, afterCall, beforeExecute, afterExecute, beforeResponse, afterResponse** |
| **Gestion des Aspects au niveau d'un point de Jonction : Ordre d'exécution des aspects** | **Spécifié implicitement d'une façon absolue, pas de mécanisme de détection de conflits** |
| **Type d'advice par point de jonction** | **Hétérogène** |
| **Nombre d'advice supporté par un point de jonction** | **illimité** |

Tableau 1.9 bilan sur TranSAT

1.3.3.2 FAC:

1.3.3.2.1 Introduction :

***FAC*** pour Fractal Aspect Component, est construit comme une extension du modèle à composants Fractal[8] qui est un modèle de composant hiérarchique fortement typé dédié à la construction, au déploiement et à l'administration (observation, contrôle, reconfiguration dynamique) de systèmes logiciels complexes, tels les intergiciels ou les systèmes d'exploitation. Cette extension offre à Fractal un support pour la gestion des préoccupations transverses.

Le modèle de composants Fractal a pour base les notions de *composant*, *interface*, *liaisons* et contrats. Les originalités principales de ce modèle résident

dans une composition hiérarchique des composants qui autorise le partage et sur une prise en charge du contrôle à la fois par réflexivité et via des contrôleurs.

FAC (Fractal Aspect Components) [27] est un modèle de programmation qui étend le modèle Fractal et permet de combiner les styles de programmation à base de composants et d'aspects. Alors que jusqu'à présent la notion d'aspect a essentiellement été étudiée en relation avec les objets, la proposition FAC est originale au sens où elle envisage cette notion au niveau des composants.

FAC introduit trois artefacts pour développer des applications à base de composants et d'aspect: *composant d'aspect*, *domaine d'aspect* et *liaison d'aspect*. Ces trois artefacts n'introduisent pas de concepts supplémentaires dans le modèle Fractal.

- Le composant d'aspect est un composant primitif qui implémente le comportement d'un aspect (i.e. le code advice).
- Le domaine d'aspect est un composant composite qui regroupe un composant d'aspect et l'ensemble des composants qui sont impactés par cet aspect.
- La liaison d'aspect est une liaison entre un composant et le composant d'aspect : elle réifie le chemin de communication emprunté lorsqu'un aspect intercepte un point de jonction et y applique son comportement.

1.3.3.2.2 Les Aspects dans FAC:

Un aspect est défini dans FAC à l'aide d'un ensemble de composants. Ceux-ci peuvent être de deux sortes : les composants Advice et les composants Tissage. Les composants Advice définissent uniquement le métier de l'aspect. Ils sont génériques et donc réutilisables.

A l'opposé, les composants Tissage définissent les points de coupe où l'aspect doit intervenir. Ils sont spécifiques à une application particulière puisqu'ils servent à tisser l'aspect dans l'application en question.

1.3.3.2.2.1 Coupes et points de jonction :

Le modèle de points de jonction de FAC repose sur les opérations des interfaces clientes et serveurs d'un composant aspectisable. Ainsi les appels entrants ou sortants d'un composant peuvent être sous le contrôle d'un aspect. Un point de jonction est un élément du flot d'exécution d'un programme. Lorsque l'approche par aspects s'applique sur des objets, ces points de jonction sont

souvent des méthodes, attributs, exceptions, etc. Dans le cadre d'une approche à composants, les points de jonction considérés sont les appels entrants et sortants d'un composant, c'est-à-dire, les opérations des interfaces clientes et serveurs d'un composant aspectisable.

Généralement, les approches par aspects reposent sur deux mécanismes importants : l'introduction et l'*l'exécution*. FAC [58] reposant sur un modèle à composants, le mécanisme d'introduction est déjà supporté par le modèle lui-même. Il est par exemple possible d'ajouter un composant dans un composite. L'ajout d'interfaces à un composant pourrait suivre l'idée d'introduction de champs ou méthodes à une classe dans les approches par aspects traditionnelles. FAC se concentre uniquement à intercepter l'exécution d'une méthode, par exemple, et à en modifier ou en enrichir le comportement.

Les **coupes sur les traces d'exécution** permettent de capturer des comportements plus complexes dans un système. Elles définissent une **séquence de points de jonction passés**, c'est-à-dire sur un historique d'événements, d'enchaînement d'occurrences dans un Système.

Elles vont plus loin que les coupes strictement structurelles et caractérisant un seul événement. Grâce à ce genre de mécanisme, FAC [58] permet d'élaborer des liaisons d'aspect capturant des séquences d'événements, qui auraient requis le tissage de nombreux aspects pour arriver au même résultat. Les coupes sur les traces d'exécution permettent donc d'automatiser ce mécanisme. Elles augmentent le pouvoir d'expression des coupes FAC.

1.3.3.2.2.2 Le composant Advice :

Le composant Advice définit exclusivement et uniquement les traitements relatifs à un et un seul aspect. Ce composant Advice peut être défini à l'aide d'un composant primitif ou un composite. Ceci doit se faire d'une manière générique indépendamment de tout contexte.

Par conséquent, les composants Advice sont réutilisables puisqu'ils représentent exclusivement le métier de l'aspect sans prendre compte du domaine dans lequel il va être appliqué.

Un **composant d'aspect dans FAC** est un composant FRACTAL fournissant au moins une interface de conseil (équivalent à la notion d'advice dans L'AOP). Il définit le comportement d'un aspect. Une **interface de conseil** – ou interface de code advice – est une interface serveur définissant un code advice de

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

1.3.3.2.2.3 Le composant Tissage :

Il correspond à une liaison composite entre les composants fonctionnels et les composants Advice. Il gère les définitions de coupes et de règles de tissage.

En effet, un composant Tissage est responsable de l'invocation des opérations des composants Advice. Il a également la charge de gérer la multiplicité des aspects (cas de plusieurs aspects portant sur le même point de jonction).

Ce composant possède une interface fonctionnelle SEC (Server Execution Controller), complémentaire à l'interface CEC. Il possède également une interface cliente pour chaque interface serveur des composants Advice qu'il va utiliser. Le composant Tissage reçoit sur son interface SEC tous les messages donnant accès aux points du flot d'exécution du composant auquel il est lié. Il analyse et filtre ces points de jonction. Si une coupe a lieu, il utilise ses interfaces requises pour exécuter les opérations adéquates des composants Advices aux quels il est connecté.

1.3.3.2.2.4 Tissage d'aspect dans FAC:

Il existe deux façons de tisser un aspect avec FAC : établir les liaisons d'aspect manuellement, en appelant l'interface de tissage de chaque composant aspectisable, ou faire un tissage global, à l'aide d'une expression de coupe transformée en requête FPATH[9]. Dans ce dernier cas, l'opération s'effectue également depuis une interface de tissage. Cependant un paramètre supplémentaire doit être donné : le nom d'un composant dit *racine* qui est le point de départ de la requête FPATH. La Figure 1.37 présente les deux opérations liées à ce processus qui permet de tisser ou d'annuler le tissage d'un aspect sur un ensemble de composants.

[9] FPATH un langage de navigation dans des documents XML, et permet de naviguer dans une architecture de composants FRACTAL représentée sous forme d'un graphe orienté.

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

Figure 1.37 – Interface de tissage : tissage d'aspect avec FAC

Tisser un aspect revient alors à établir un ensemble de liaisons d'aspect. L'opération de tissage devient donc une opération de reconfiguration architecturale à part entière : parcours de l'architecture, création de liaisons d'aspect, création d'un composite pour le domaine d'aspect, et ajout des composants aspectisables dans ce composite.

Le tissage d'aspect dans FAC s'effectue sois avec FRACTAL-ADL, ou a travers Fractal EXPLORER. FRACTAL-ADL est un Langage de Description d'Architecture (ADL) pour le modèle à composants FRACTAL. Il s'agit d'un langage en XML, Grâce à ce langage, les types de composant, leur implantation, les composites et les liaisons peuvent être décrits, indépendamment de l'implantation des composants. FAC introduit donc deux modules dans FRACTAL-ADL pour la liaison aspect et le tissage d'aspect.

FRACTAL EXPLORER[10] permet également de définir une opération de tissage et a été étendu pour un tel support. En particulier le contrôle de tissage est visible dans la console.

1.3.3.2.2.5 Composition et ordonnancement d'aspect:

La question de la multiplcité se pose quand l'exécution d'un composant doit être altérée par plusieurs aspects. Ces derniers peuvent éventuellement être en conflit s'ils interviennent à un même point de jonction. Dans un tel cas, le développeur doit intervenir afin de préciser la sémantique de l'intégration. Le plus souvent, cela revient à définir l'ordre d'exécution des traitements des différents aspects. FAC adopte cette même démarche. Les composants de tissage ne sont plus liés directement à l'interface CEC du composant métier mais ils forment une chaîne à la manière du patron de conception "chaîne de responsabilités". Afin de constituer cette chaîne, tout composant Tissage possède une interface CEC *optionnelle*. A chaque fois que le composant Tissage reçoit un message sur son interface SEC (i.e. à chaque point de jonction), il déclenche les traitements de

[10] FRACTAL EXPLORER est une console graphique générique pour le management d'applications FRACTAL.

l'aspect auquel il appartient. Puis, il retransmet le message sur son interface CEC. Ainsi, le composant Tissage suivant dans la chaîne prend la main et fait ses traitements et ainsi de suite.

Lorsque plusieurs composants d'aspects sont tissés [30], soit directement, soit par une expression de coupe sur le même composant de base, leur ordre d'exécution est celui de leur tissage : c'est à dire les composants d'aspect tissés en premier sont exécutés en premier.

FAC ne permet pas de définir des ordres de précédence globaux ou locaux entre les aspects comme cela existe pour d'autres langages ou frameworks AOP (par exemple AspectJ JAC).

1.3.3.2.3 Exemple illustratif :

Dans cette section nous donnons un exemple d'utilisation de FAC pour la définition d'un aspect. L'aspect développé est celui de l'authentification. Afin de simplifier le discours nous nous limitons uniquement à vérifier que les requêtes reçues par l'agenda du directeur proviennent bien de l'organisateur de rendezvous de la secrétaire. Nous utilisons une approche d'authentification dite *upfront login authentication approach*. Elle consiste à demander, à toute entité essayant d'accéder au composant Agenda, de fournir un login et un mot de passe afin d'obtenir la permission d'utiliser l'interface Rdv.

Le développement de l'aspect authentification et son intégration sont réalisés suivant les étapes suivantes:

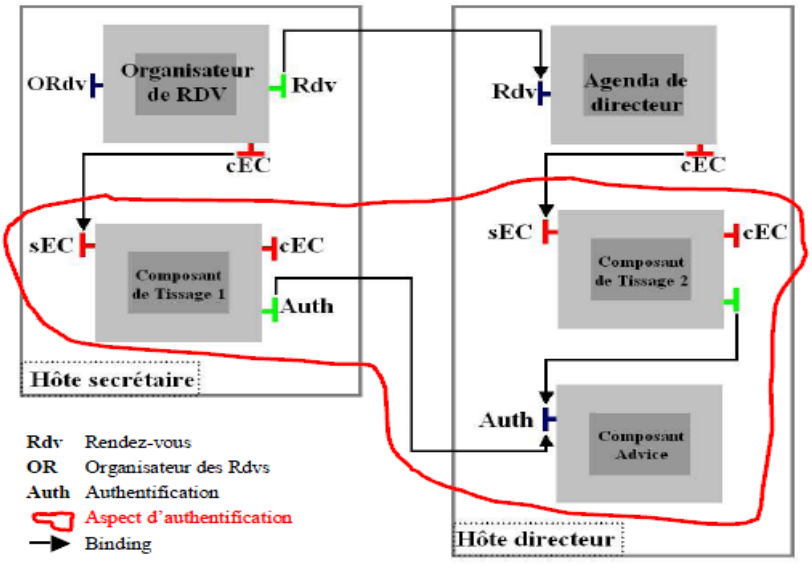



Figure 1.38. Exemple d'un aspect d'authentification avec FAC tiré de [8]

Étape 1 : Définition des composants Advice.

Dans cet exemple, notre aspect comporte un unique composant Advice. Ce dernier fournit une interface *Auth* définissant deux opérations : *login* et *isAuthenticated*. La première opération permet d'authentifier des entités en se basant sur leur login et leur mot de passe. La deuxième vérifie si une entité a été déjà authentifié ou pas.

Étape 2 : Définition des composants Tissage.

Dans notre exemple, il existe deux composants de Tissage. Le premier est lié au composant Organisateur de Rendezvous. Il capture tous les points de jonction correpondant à une émission de messages à partir de son interface client Rdv. Lors de l'émission du premier message par le composant Organisateur Rdv, il suspend ce dernier pour effectuer d'abord l'authentification (utilisation de l'opération *login* du composant Advice). Le second composant de tissage est lié au composant Agenda. Il capture tous les messages reçus par le composant Agenda. Avant d'exécuter un message, il vérifie
si l'émetteur a été déjà authentifié (utilisation de l'opération *isAuthenticated* du composant Advice).

Étape 3 : Tissage de l'aspect.

Le tissage est rendu actif en assemblant les composants de notre aspect avec ceux de l'application. Cet assemblage revient à lier l'interface CEC de chaque composant métier avec l'interface SEC d'un composant Tissage. Ainsi, le composant Agenda est connecté au composant Tissage 1.De manière analogue le composant Organisateur de Rdvs est connecté au composant Tissage 2. Une fois cet assemblage terminé, notre aspect est opérationel. L'application est ainsi enrichie pour assurer l'authentification.

1.3.3.2.4 Evaluation :

Avec FAC, Nicolas Pessemier et al, montrent qu'une programmation orientée aspect (AOP) sûre peut être supportée par la programmation orientée composant en proposant un mécanisme pour contrôler l'ouverture du composant avec le respect des techniques de l'AOP.

L'approche est capable d'ouvrir un composant pour l'AOP tout en gardant son contenu caché de l'extérieur. Ce compromis ouvre la voie à une intégration sûre pour AOP dans la programmation orientée composant.

Dans le modèle FAC un aspect est un composant d'aspect qui coexiste au milieu des composants métiers de l'application. Tisser un aspect à un ensemble de composants revient à définir une liaison que nous appelons liaison transverse. Toute définition de liaison transverse modifie la structure applicative pour partager le composant d'aspect avec les composants qu'il affecte dans un composite représentant la préoccupation transverse.

La définition d'un aspect est séparée en deux parties. L'une générique comprenant des composants Advice qui définit la partie métier de l'aspect. L'autre partie est spécifique à une application donnée comprenant des composants Tissage. Ces derniers définissent les points où l'aspect doit intervenir et déclenchent l'exécution des opérations adéquates au niveau des composants Advice. Dans ce contexte, le tissage revient à assembler les composants Tissage aux composants applicatifs.

Cette approche présente deux avantages pour faciliter la maintenance et l'évolutivité du système. D'une part, il permet une distinction visuelle entre les composants de base et les composants aspectuels. D'autre part, la composition entre les composants traditionnels et la composition entre les composants de base et les composants aspectuels sont définies dans deux sections différentes.

Cependant, FAC présente quelques inconvénients. En effet, comme le cas de DAOP-ADL, la distinction entre les composants aspectuels et les composants de base diminue la réutilisabilité des composants. De plus, bien que l'utilisation du langage XML facilite la génération de l'architecture et l'extensibilité de ce langage en mesure d'intégrer de nouvelles fonctionnalités non initialement prévu, elle présenté un inconvénient concernant la lisibilité de l'architecture logicielle.

| Point de jonction sur trace Tracecuts | Type de tissage : dynamique/ Statique | Composition d'aspect | Réutilisation aspect | Spécification du tissage (Application) | Indépendance coupe / aspect | Spécification Coupe | Type de Point de jonction | Type de Conseil | Gestion des Aspects au niveau d'un point de Jonction : Ordre d'exécution des aspects | Type d'advice par point de jonction | Nombre d'advice supporté par un point de jonction |
|---|---|---|---|---|---|---|---|---|---|---|---|

| | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| illimité | Hétérogène | fait implicitement selon leur ordre d'exécution | After Before Around | Introduction et exécution de méthode | (d'expressions régulières FRACTAL ADL) | OUI | Avec FRACTAL-ADL ou FRACTAL EXPLORER | OUI | (Non supporté) juste un mécanisme de résolution de conflit Chaine de responsabilités | Tissage d'aspect revient à établir un ensemble de liaisons d'aspect (non implémenté) | OUI |

Tableau 1.10 bilan sur FAC

1.3.3.3 CAM/DAOP-ADL :

1.3.3.3.1 Introduction:

CAM/DAOP est un modèle à composants et par aspects, qui combine les bénéfices des deux approches [58]. CAM (Component Aspect Model) est le modèle et DAOP (Dynamic Aspect Oriented Platform) en est la plate-forme d'exécution.

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

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

1.3.3.3.2 Les Aspects dans CAM/DAOP:

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

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

1.3.3.3.2.1 Coupes, points de jonctions et les déclarations des advices:

Les points de jonction supportés par CAM sont les interfaces publiques d'un composant. Le contenu d'un composant ne peut être affecté par des aspects pour préserver la propriété d'encapsulation des composants. Les points de jonction considérés sont donc les messages émis et reçus entre les composants ainsi que la création ou destruction d'un composant.

Ainsi, CAM définit un modèle de point de jonction où les aspects peuvent être appliqués avant et après (entrants et sortants) des messages et des événements, et aussi avant et après la création et la destruction des instances de composants. CAM supporte les points de jonction suivante:

- BEFORE_NEW: Il capture l'instant avant de créer une nouvelle instance d'un composant.
- AFTER_NEW: l'instant après de créer une nouvelle instance d'un composant.
- BEFORE_DESTROY: capture l'instant avant l'élimination d'une instance de composant de l'application.
- AFTER_DESTROY : capture l'instant après l'élimination d'une instance de composant de l'application.
- BEFORE_SEND: capture l'instant avant un composant source envoie un message.
- AFTER_SEND : l'instant après un composant source envoie un message.
- BEFORE_RECEIVE Il capture l'instant avant la réception d'un message par un composant cible.
- AFTER_RECEIVE Il capture l'instant après la réception d'un message par un composant cible.
- AFTER_THROW: Il capture l'instant après une exception est levée.
- SEND_EVENT Il capte l'instant où un événement est envoyé. Se point de jonction remplace le point de jonction "around".

Les coupes sont définies au niveau du langage d'architecture DAOP-ADL, donc extérieurement aux aspects respectant ainsi une symétrie de placement (symétrie de relation).

Les aspects ont accès à un contexte d'interception permettant de connaître le composant émetteur ou récepteur, ainsi que le message.

1.3.3.3.2.2 Le tissage d'aspect:

CAM / DAOP offre un mécanisme de tissage a l'exécution qui repose sur:

1. Le chargement de la spécification XML basé sur les coupes dans les structures internes de la plate-forme DAOP lorsque l'application est instanciée,
2. et consulter et utiliser cette information à l'exécution pour injecter les aspects en composants.

Le mécanisme de tissage dans l'implémentation Java de CAM/DAOP est basé sur l'utilisation du mécanisme de réflexion de Java. Les composants sont instanciés et communiquent entre eux à l'aide d'un ensemble de services offerts par la plate-forme DAOP.

Chaque fois qu'un de ces services sont utilisés et (i) une nouvelle instance d'un composant est créé, (ii) d'une instance existante d'un composant est éliminé du système, (iii) un message est envoyé et reçu, (iv) une exception est levée, ou (v) un événement est levée, La plateforme DAOP intercepte les points de jonction correspondant, et consulte les informations sur les aspects qui doivent être évaluées sur ces points de jonction et invoque la méthode advice appropriés sur l'instance d'aspect.

Un avantage important du CAM / DAOP est que le tissage des aspects dans les composants peuvent être adaptés de manière dynamique à l'exécution.

1.3.3.3.2.3 La composition d'aspect**:**

Aucun mécanisme ne permet une manipulation totale des aspects s'appliquant sur le même point de jonction.

1.3.3.3.3 Exemple illustratif:

```
1<component role="chat">
2 <providedInterface>ChatProv.xml</providedInterface>
3 <requiredInterface>
4 <fromTargetComponent role=chat/>
5 <requiresMessage>ChatReq.xml</requiresMessages>
6 </requiredInterface>
7 <requiredInterface>
8 <fromTargetComponent role=awarenessList/>
9 <requiresMessage>AwarReqlnt.xml</requiresMessages>
10 <requiredInterface>
11 <implementations>
12 <implementation>
```

```
13 <name>chat1</name>
14 <language>java</language>
15 <class>Chat.class</class>
16 </implementation>
17 </implementations>
18</component>
```

Figure 1.39– CAM/DAOP: exemple d'un composant en DAOP-ADL.

La Figure 1.39 illustre la définition d'un composant avec DAOP-ADL. Les interfaces fournies et requises sont décrites et font à chaque fois référence à des messages. Par exemple, l'interface fournie de la Ligne 2 fait référence à un autre fichier XML donné dans le Listing dans la Figure 1.40 Enfin les informations sur l'implantation du composant sont données aux Lignes 11–17.

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

Figure 1.40– CAM/DAOP : exemple de définition d'une interface en DAOP-ADL.

```
1<aspect role="persistence">
2 <evaluatedInterface>
3 <joinpoint>BEFORE_SEND</joinpoint>
4 <capturedMessages>PersistenceEval.xml</capturedMessages >
5 </evaluatedInterface >
6 <implementations>
7 <implementation>
8 <name>persistence1</name>
9 <language>java</language>
10 <class>LDAPPersistence.class</class>
11 </implementation>
12 <implementation><name>persistence2</name>
14 <language>java</language>
15 <class>OraclePersistence.class</class>
16 </implementation>
17 </implementations>
18</aspect>
```

Figure 1.41 – CAM/DAOP : exemple de définition d'un aspect en DAOP-ADL.

La Figure 1.41 montre comment un aspect est défini dans le langage d'architecture DAOPADL. Tout comme un composant, un rôle est attribué ainsi qu'une interface dite évaluée qui définit le type de point de jonction à la Ligne 3.

```
1<compositionRules>
2 <componentCompositionRules>
3 <compositionRuleFor role="chat">
4 <compositionRule>
5 <formatRole>awarenessList</formatRole>
6 <realRole>userList</realRole>
7 <compositionRule>
8 </compositionRuleFor>
```

```
9 <componentCompositionRules>
11 <aspectEvaluationRules>
12 <createComponent role="chat">
13 <BEFORE_NEW>
14 <concurrent>
15 <aspect role="authentification"/>
16 </concurrent>
17 </BEFORE_NEW>
18 </createComponent>
19 <aspectEvaluationRules>
20 <compositionRules>
```

Figure 1.42 – CAM/DAOP : exemple de définition d'une composition en DAOP-ADL**.**

Enfin la Figure 1.42 propose une déclaration de composition entre le composant et l'aspect. Il s'agit en réalité de la déclaration de coupe qui est séparée de la déclaration des codes advice.

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

1.3.3.3.4 Evaluation :

CAM/DAOP se distingue par son modèle général et indépendant de tout langage de programmation et de son langage d'architecture. CAM sépare avantageusement la définition des coupes des aspects, ce qui permet d'accroître la réutilisation de ces derniers. Cependant, l'approche propose une asymétrie d'élément et le modèle n'est pas hiérarchique ni facilement extensible.

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

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

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

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

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

**Symétrie de placement** Le modèle sépare clairement la définition d'un aspect de sa liaison aux composants, donc c'est une approche ***asymétrique***

| | |
|---|---|
| **Point de jonction sur trace (Tracecuts)** | **NON** |
| **Type de tissage : dynamique/ Statique** | **Dynamique à l'exécutions** |
| **Composition d'aspect** | **Pas possible** |
| **Réutilisation aspect** | **OUI** |
| **Spécification du tissage (Application)** | **Mécanisme de tissage Basé sur la plate forme DAOP** |
| **Indépendance coupe / aspect** | **OUI** |
| **Spécification Coupe** | **Langage d'architecture DAOP-ADL** |
| **Type de Point de jonction** | **createComponent, destroyComponen, sendMessage, ThrowEvent** |
| **Type de Conseil** | **BEFORE_NEW, AFTER_NEW, BEFORE_DESTROY, AFTER_DESTROY, EFORE_SEND, AFTER_SEND, BEFORE_RECEIVE, AFTER_RECEIVE , AFTER_THROW, SEND_EVENT** |
| **Gestion des Aspects au niveau d'un point de Jonction : Ordre d'exécution des aspects** | */* |
| **Type d'advice par point de jonction** | **Hétérogène** |
| **Nombre d'advice supporté par un point de jonction** | **illimité** |

Tableau 1.11 bilan sur CAM/DAOP

1.3.3.4 AspectLEDA

1.3.3.4.1 Introduction:

AspectLEDA est une approche qui étend l'ADL LEDA avec les concepts de la programmation orienté aspect. Il est basé sur le modèle d'architecture logicielle orienté aspect. Cette approche comporte deux étapes :

La définition d'un modèle initial de l'architecture (décrivant l'architecture du système de base) et l'ajout des aspects (spécifiant les aspects à appliquer au système). Le modèle initial l'architecture est définie en utilisant l'ADL LEDA.

Une fois l'architecture initiale est définie, les nouvelles exigences qui se dégagent lors des processus de raffinement et d'entretien sont intégrés dans l'architecture, sous forme d'aspects. Ainsi, l'ajout d'une nouvelle préoccupation revient à ajouter un nouvel aspect.

1.3.3.4.2 Les Aspects dans AspectLEDA:

En AspectLEDA, les aspects [89] sont décrits de la même manière que les composants. LEDA est un ADL sans connecteurs. Toutefois, les aspects et les composants de l'architecture initiale du logiciel sont définis à différents niveaux. Toutefois, il introduit la notion de coordonnateur pour définir le processus de tissage qui synchronise les aspects et les composants et coordonne les deux niveaux. Ce coordinateur préserve la réutilisabilité et l'encapsulation des aspects car les coordinations des aspects et des composants sont spécifiée à l'extérieur des aspects extérieurs.

La description architecturale en AspectLEDA[90] est ensuite traduite en LEDA pur par l'analyse du fichier AspectLEDA avec les analyseurs lexical et syntaxique correspondants. Après, ce code LEDA est traduit en Java, et donc un prototype simulant l'exécution du système étendu peut être obtenue.

Enfin, il est important de souligner que cette approche n'est encore qu'une proposition. Il ne fait pas disposer d'un outil pour supporter sa méthodologie, et il n'est pas capable de compiler ses aspects dans n'importe quelle plate-forme technologique.

1.3.3.4.2.1 Les coupe/ pointcut et les déclarations des advices:

La première étape pour décrire une architecture AspectLEDA génère une description intuitive représente les composants du système et les aspects qui doivent être ajoutée (comme composants).En outre, un nouveau type de connecteur étendant UML est défini, ce qui permet de représenter les interactions entre les composants et les aspects. Il va intercepter les interactions préalablement définis entre les composants du système. Ces objets représentent les connecteurs complexes, y compris le point, dans lequel l'aspect doit agir, les conditions dans lesquelles les aspects doivent être activés et le moment où elles doivent être faites.

Après la première approche intuitive du système le nouveau connecteur doit être décrit. Il va intercepter les méthodes ***call*** pour la définition des points de jonction.

1.3.3.4.2.2 Tissage d'aspect :

Les aspects dans AspectLEDA sont transformés en composantes qui restent séparés (non tissé) pendant tout le processus. Comme propriété supplémentaire, il est soutenu par un outil qui aide l'architecte lors de la conception lui fournissant la possibilité de générer un code exécutable Java pour obtenir la simulation de l'architecture

1.3.3.4.3 Exemple illustratif:

La première étape est la description du système de base. Le système de base ne comprend ni référence à des aspects particuliers, ni support d'aspects primitifs. Ainsi, le système de base est décrit dans LEDA. La Figure 1.1 montre l'architecture, qui est décrit comme un élément composé dont le client et le serveur deux composants génériques, qui sont définis par leurs interfaces. La description des rôles spécifie les protocoles d'interaction. La connexion entre les composants est spécifiée dans la section ***attachments***.

```
Component Basesystem {
interface none;
composition
    ccli: Client;
    cserv: Server;
attachments
ccli.requireM(serverop,param)<>cserv.provideM
(serverop,param);
}
component Client {
interface
   requireM:RequireM;
}
component Server {
interface
   provideM:ProvideM;
}
role ProvideM(serverop,param){
 spec is
   serverop?(answer).(value)answer!(value).
   ProvideM(serverop,param);
   }
role RequireM(serverop,param){
 spec is
   (answer)serverop!(answer).answer?(value).
   RequireM(serverop,param);
   }

instance basesys:Basesystem;
```

Figure 1.43. Architecture de base en LEDA

Ensuite, les aspects doivent être décrits. Ils sont exprimés en tant que composants LEDA (Figure 1.44).

```
component Aspect {
interface
   asprole:AspectRole;
}
role AspectRole(aspectop,param){
 spec is
   aspectop?(ans).(val)answer!(val).
   AspectRole(aspectop,param);
   }
```

Figure 1.44. Définition d'aspect avec LEDA

Maintenant, le système peut être étendu décrit dans AspectLEDA. Sa description est simple (Figure 1.45): Un système de AspectLEDA est défini comme un composant composé –section composition -, constituée par deux types d'éléments: système etaspects. Ils sont définis comme des éléments architecturaux. L'interaction entre leséléments du système et les aspects sont décrits dans la section ***attachment***. Enfin Les composants et leurs interfaces sont inclus dans la description AspectLEDA.

```
Component Extendedsystem {
composition
    basesystem: System;
    asp: Aspect;
attachements
  basesystem.cserv.serverop()<<RMA,
  condvalue,when_cond>>asp.aspectop();
}
component System {
interface
    none;
}
component Aspect {
interface
    asprole: AspectRole;
}
instance extsys: Extendedsystem;
```

Figure 1.45. L'Architecture *AspectLEDA* avec un seul Aspect.

1.3.3.4.4 Evaluation :

L'utilisation [93] de l'élément coordonnateur conserve la réutilisation et l'encapsulation des aspects vu que la coordination des aspects et des composants est spécifiée en dehors des aspects. Cependant, l'intégration d'une nouvelle abstraction architecturale augmente la complexité de la description de l'architecture logicielle car l'architecte se trouve obligé de se familiariser avec un nouvel élément autre que les abstractions de bases (composants et connecteurs). De plus, l'intégration d'une nouvelle abstraction architecturale nécessite la

définition d'une nouvelle plate-forme pour supporter les descriptions d'architectures utilisant AspectLEDA car les outils existants ne le permettent plus.

En effet, AspectLEDA oblige l'analyste à introduire de nouvelles exigences dans le modèle sous forme de nouveaux aspects, sans tenir compte qu'ils sont des aspects ou non. Ce qui constitue un inconvénient évident de cette approche. De plus, elle ne donne pas l'analyste l'occasion d'introduire des aspects au début du processus de développement logiciel. Ces aspects seront dégagés pendant l'étape de raffinement et de maintenance.

| Point de jonction sur trace (Tracecuts) | NON |
|---|---|
| Type de tissage : dynamique/ Statique | Pas de notion de tissage d'aspects |
| Composition d'aspect | Pas possible |
| Réutilisation aspect | NON |
| Spécification du tissage (Application) | / |
| Indépendance coupe / aspect | NON |
| Spécification Coupe | LEDA ADL |
| Type de Point de jonction | méthode call |
| Type de Conseil | / |
| Gestion des Aspects au niveau d'un point de Jonction : Ordre d'exécution des aspects | / |
| Type d'advice par point de jonction | / |
| Nombre d'advice supporté par un point de jonction | / |

Tableau 1.12 Bilan sur AspectLEDA

1.3.3.5 AC2-ADL

1.3.3.5.1 Introduction:

AC2-ADL est un nouvel ADL orientée aspect qui fournit un support formel pour décrire les architectures logicielles d'une application ou d'un système orienté aspect.

1.3.3.5.2 Les Aspects Dans Casear:

Dans AC2-ADL [91], les concepteurs peuvent modéliser des aspects architecturaux à l'aide de composants aspectuels. AC2-ADL fait recours aussi à un type particulier de connecteur appelé Aspectual Connector pour capturer l'interaction transversale de certains éléments architecturaux.

La composition entre les composants de base et la composition entre les composants de base et les composants aspectuels sont définies dans une même section, appelée section de configuration.

1.3.3.5.2.1 Les composants Aspects:

Les composants aspectuels (Aspectual Component) caractérisés par une interface de type spécial appelée cosscutting interface décrivant les services transversaux fournis par les composants aspectuels.

En AC2-ADL, les composants Aspectuels ont un nouveau type d'interfaces. Contrairement aux interfaces classiques (tels que les interfaces fournies et requises), ces interfaces sont appelées interfaces transversales décrivant les services fournis par les composantes transversales aspectuelle. Dans cette section, nous fournissons une syntaxe partielle du prototype de la composante aspectuelle et des interfaces. Un composant aspectuel est modélisée dans AC2-ADL comme suit dans Figure 1.46:

```
aspectComponent ::=
 aspectComponent component_name is
{
 parameters
    {parameters name is simpleType;}
 methods {methods}
 crosscuttingInterface
     {crosscuttingInterfaces}
 requiredInterface {requiredInterfaces}
 [subArchtecture subArchtecture_name
              is{ architeture|null;}]
 [mappingDeclaration
      {mapping_expression}| null;]
 [internalProcess is
   {{process_expression;}| null;}]
 [constrains is
     {{constrains_expression;}| null;}]
 }
```

Figure 1.46. La syntaxe d'un composant aspectuel dans AC2-ADL

```
crosscuttingInterfaces ::=
 crosscuttingInterface_name is {
 [ add_Operation
     method | {method;}| null ]|
 [ replace_Operation
      { method | {method;} | null ] |
 [ introduce_Operation
     { [parameters_name | methods|
       providedInterfaces |
       requiredInterfaces] } | null]}
requiredinterfaces::=
   requiredInterface_name is {
    [Operation method | {method;}| null ] }
providedInterfaces::=
   providedInterface_name is {
    [Operation method | {method;} | null ] }
method::= method_name is
  [direction (parameters_name)|
  { parameters_name;})| null]
direction ::= in | out | in|out
```

Figure 1.47. Une interface aspectuelle dans AC2-ADL

1.3.3.5.2.2 Les connecteurs Aspects:

Les connecteurs [93] aspectuels comportent deux types de rôles : le rôle de base (BaseRole) connecté à un port d'un composant de base et le rôle transversal (Crosscutting Role) connecté à un port d'un composant aspectuel.

Les connecteurs aspectuels représentent les participants qui peuvent se joindre dans l'interaction décrite par ce connecteur. Le rôle de base indique un ensemble d'objets appliqués par les composants aspectuels, comme des instances de composants, les instances de connecteur ou les interfaces, et le rôle transversal indique un ensemble d'objets transversaux, tels que les interfaces transversales des instances de composants aspectuels. Chaque rôle peut contenir un ou plusieurs comportements, qui spécifient une liste d'activités de ce rôle.

En comparaison avec les connecteurs aspectuels, les connecteurs traditionnels dans AC2-ADL possèdent seulement le rôle de base. La conception du connecteur aspectuel est principalement concentrée sur la spécification des protocoles d'interactions qui représentent les relations transversales entre les rôles transversaux et ceux de base.

```
aspectConnector::=
  aspectConnector connector_name is {
baseRole baseRoles
crosscuttingRole crosscuttingRoles
crosscuttingProtocols
        crosscuttingProtocol_expressions |
{crosscuttingProtocol_expressions;}| null;
[constrains is {constrains_expressions;}| null;] }
baseRoles::={Role_name;}
|{Role_name behaviors behavior_names;}
crosscuttingRole_name::= {Role_name;}|
{Role_name behaviors behavior_names;}
crosscuttingProtocol_expressions::=
crosscuttingRole_name. behavior_name
crosscuttingProtocol_types
baseRole_name. behavior_name
crosscuttingProtocol_types::= before | after |around| introduce
```

Figure 1.47. La syntaxe d'un connecteur aspectuel dans AC2-ADL

1.3.3.5.2.3 Les coupe/ pointcut et les déclarations des advices:

En plus des déclarations de connecteurs aspectuels et les définitions des rôles et des comportements, le connecteur aspectuel est principalement concentré sur les spécifications d'interaction qui représentent les relations transversales entre les rôles transversaux et celles de base. AC2-ADL propose quatre types de protocoles d'interactions transversales énumérées par les mots-clés: **before, after, around, introduce**. La sémantique des types d'interaction transversale

protocole est similaire à celle de la composition des conseils advices dans AspectJ.

Dans la configuration architecturale réelle, un rôle de base d'un connecteur aspectuel peut être joué par plusieurs points de jonctions architecturales qui peuvent être affectés par des composants d'aspect. AC2 définit se qu'on appel PCD (équivalent a la notion de coupe dans AOP) Ainsi, un pointcut designator (PCD) devrait être défini comme une formule qui spécifie certains points de jonctions ou l'ensemble des points de jonction à laquelle l'interface transversale d'un composant aspect est applicable.

Outre la composition typique des composants dans des configurations architecturales, AC2-ADL définit explicitement les points de jonction architecturale: Ils ya certains endroits où l'effet des composants aspectuelle peut se produire. Ces lieux incluent: ① la composante / instances de composants aspectuelle, ② les différentes interfaces de ces cas, ③ les opérations à partir des interfaces, ④ et même les connecteurs 'instances'. Ainsi, quatre types différents de point de jonction architecturaux sont proposés pour désigner la sémantique des points de jonctions qui sont respectivement définis comme suit: **component_JP, interfaces_JP, operation_JP et connector_JP.**

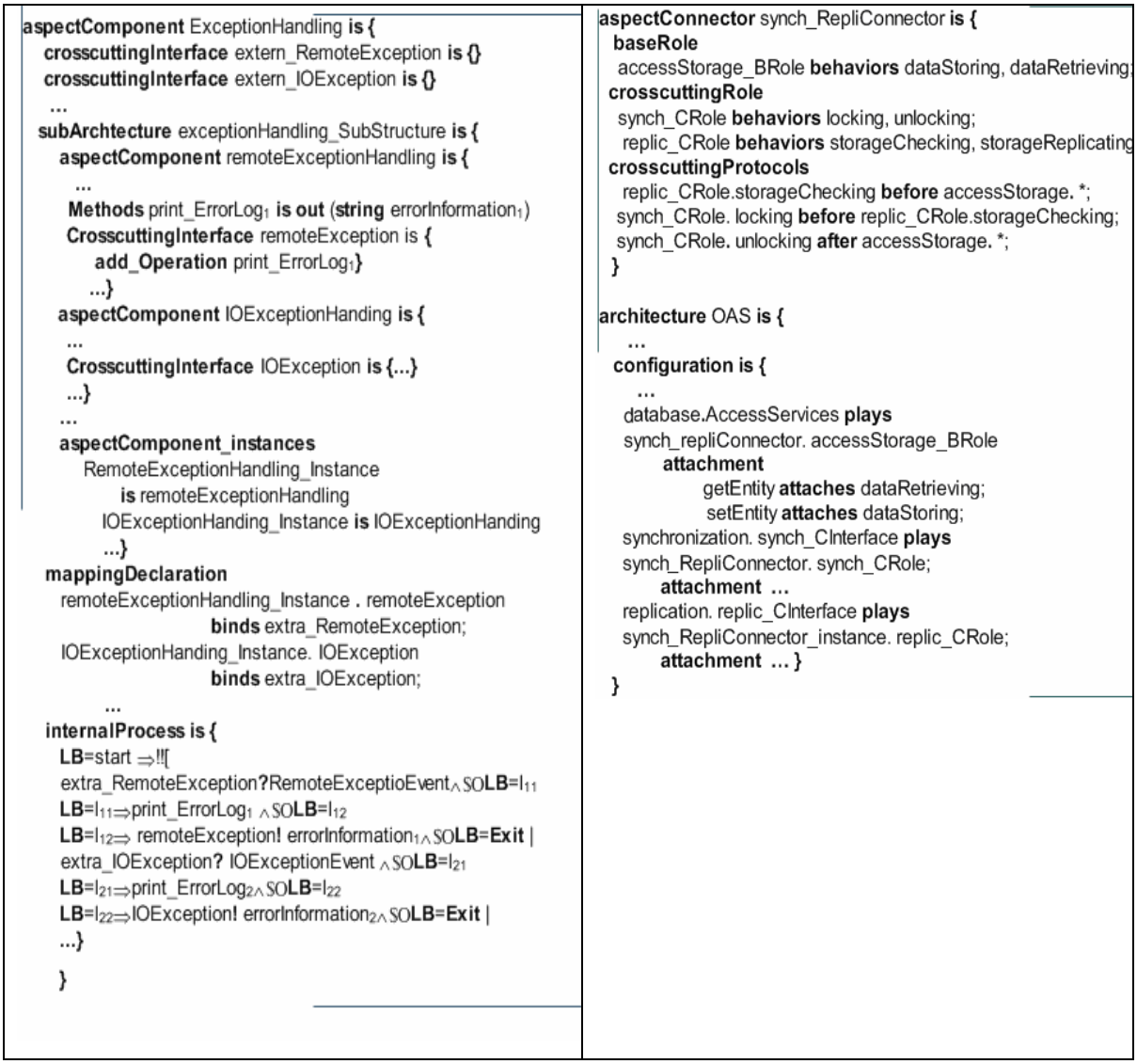
1.3.3.5.2.4 Le tissage d'aspect:

Dans AC2-ADL on parle plus de tissage mais de configuration architecturale. La configuration architecturale, est définie par l'inscription d'un ensemble de connexions entre les PCD et les rôles de base d'un connecteur, et un ensemble de connexions entre les interfaces transversales et des rôles transversaux. En outre, pour les expressions plus explicites et précises des attachements entre les différentes opérations et les comportements correspondants. Les compositions entre les composants et les composants aspect peuvent être plus clairement décrites dans la configuration architecturale.

1.3.3.5.3 Exemple illustratif :

Un exemple de composants aspectuels hétérogènes est le composant aspectuel *ExceptionHandling* .Il ya différentes situations exceptionnelles dans OAS (Online Auction System) qui exigent des solutions différentes, comme des problèmes de communication, le stockage des données et des problèmes de récupération, invalides problèmes de données et ainsi de suite. Toutes ces

catégories de problèmes peuvent être représentés comme des sous composants aspectuels constituant le composite ExceptionHandling comme indiqué dans Figure 1.48. Chaque sous-composante aspectuelle fournit une interface transversale correspondant à l'encapsulation des différentes fonctionnalités de gestion des exceptions**.**



```
aspectComponent ExceptionHandling is {
  crosscuttingInterface extern_RemoteException is {}
  crosscuttingInterface extern_IOException is {}
  ...
 subArchtecture exceptionHandling_SubStructure is {
   aspectComponent remoteExceptionHandling is {
     ...
    Methods print_ErrorLog1 is out (string errorInformation1)
    CrosscuttingInterface remoteException is {
        add_Operation print_ErrorLog1}
       ...}
   aspectComponent IOExceptionHanding is {
    ...
    CrosscuttingInterface IOException is {...}
    ...}
   ...
   aspectComponent_instances
      RemoteExceptionHandling_Instance
           is remoteExceptionHandling
        IOExceptionHanding_Instance is IOExceptionHanding
        ...}
  mappingDeclaration
   remoteExceptionHandling_Instance . remoteException
                    binds extra_RemoteException;
   IOExceptionHanding_Instance. IOException
                    binds extra_IOException;
        ...
  internalProcess is {
   LB=start ⇒!![
   extra_RemoteException?RemoteExceptioEvent∧SOLB=l11
   LB=l11⇒print_ErrorLog1 ∧SOLB=l12
   LB=l12⇒ remoteException! errorInformation1∧SOLB=Exit |
   extra_IOException? IOExceptionEvent ∧SOLB=l21
   LB=l21⇒print_ErrorLog2∧SOLB=l22
   LB=l22⇒IOException! errorInformation2∧SOLB=Exit |
   ...}
  }
```

```
aspectConnector synch_RepliConnector is {
 baseRole
  accessStorage_BRole behaviors dataStoring, dataRetrieving;
 crosscuttingRole
  synch_CRole behaviors locking, unlocking;
  replic_CRole behaviors storageChecking, storageReplicating
 crosscuttingProtocols
  replic_CRole.storageChecking before accessStorage. *;
 synch_CRole. locking before replic_CRole.storageChecking;
 synch_CRole. unlocking after accessStorage. *;
}

architecture OAS is {
  ...
 configuration is {
    ...
  database.AccessServices plays
  synch_repliConnector. accessStorage_BRole
       attachment
            getEntity attaches dataRetrieving;
            setEntity attaches dataStoring;
  synchronization. synch_CInterface plays
  synch_RepliConnector. synch_CRole;
       attachment ...
  replication. replic_CInterface plays
  synch_RepliConnector_instance. replic_CRole;
       attachment ... }
}
```

Figure 1.48. Modélisation des composants aspectuels hétérogènes

1.3.3.5.4 Evaluation:

AC2-ADL permet de fournir une base formelle pour la représentation de préoccupations transversales et d'établir l'architecture du logiciel avec plus de fiabilité. De plus, il faut mentionner qu'AC2-ADL permet d'exprimer les mécanismes de quantification pour décrire des ensembles de joinpoints jouant le même rôle.

Comme pour le cas de tous les langages développés de zéro, AC2-ADL nécessite plus d'effort pour sa mise en oeuvre. De plus, la distinction visuelle entre les abstractions aspectuelles et les abstractions de bases facilite l'entretien et l'évolution du système. Cependant, la distinction entre les composants aspectuels et les composants de base diminue la réutilisabilité des composants. De même, l'utilisation d'un connecteur aspectuel avec une nouvelle interface diminue la réutilisabilité des connecteurs.

De plus, la définition des différents types de composition dans une même section présente un inconvénient pour assurer la tâche de maintenance. Concernant la génération de code, AC2-ADL ne présente aucun type de générateur de code.

| Point de jonction sur trace (Tracecuts) | Type de tissage : dynamique/ Statique | Composition d'aspect | Réutilisation aspect | Spécification du tissage (Application) | Indépendance coupe / aspect | Spécification Coupe | Type de Point de jonction | Type de Conseil | Gestion des Aspects au niveau d'un point de Jonction : Ordre d'exécution des aspects | Type d'advice par point de jonction | Nombre d'advice supporté par un point de jonction |
|---|---|---|---|---|---|---|---|---|---|---|---|

| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| illimité | Hétérogène | / | Before after around introduce | component_JP,interfaces_JP, operation_JP et connector_JP | AC2- ADL | OUI | Dans la partie configuration architecturale | OUI | Pas possible | Configuration architecturale ensemble de connexions entre les PCD (coupes) et les rôles de base d'un connecteur | NON |

Tableau 1.13 Bilan sur AC2-ADL

Les approches symétriques :

Les approches symétriques utilisent le même type de module pour représenter toutes les préoccupations d'un système (chaque composant du modèle architectural est considéré comme une préoccupation).

1.3.3.6 Aspectual ACME :

1.3.3.6.1 Introduction :

AspectualACME est une extension de l'ADL ACME pour assurer la représentation modulaire des préoccupations transversales.

1.3.3.6.2 Les Aspects dans Aspectual ACME :

Dans AspectualACME, les aspects sont définis comme des composants ACME. Les Aspects sont modélisés au moyen de connecteurs, les rôles transversaux, des rôles de base, et des composants. La seule extension nécessaire pour intégrer les aspects dans ce langage est l'introduction de la notion de connecteur aspectuel (Aspectual Connector).

1.3.3.6.2.1 Les connecteurs aspectuels:

Un connecteur aspectuel est un connecteur ACME avec une nouvelle interface qui est définie d'une part, pour distinguer entre les composants de base et les aspects et, d'autre part, pour capturer comment sont interconnecter les différentes catégories de composants. En conséquence, un connecteur aspectuel relie un composant aspectuel avec un composant de base.

```
Connector aConnector = {
Role aRole1;
Role aRole2;
}
```

```
Connector aConnector = {
Base Role aBaseRole;
Crosscutting Role aCrosscuttingRole;
Glue glueType;}
```

(a) regular connector in ACME (b) aspectual connector in AspectualACME

Figure 1.49. Un connecteur Régulier et un connecteur aspectuel

Ce connecteur aspectuel contient au moins un rôle de base lié à un port d'un composant de base, un rôle transversal lié à un port d'un composant aspectuel ainsi qu'une clause glue (hérité d'ACME) qui spécifie les détails de composition entre les composants de base et les aspects. Il existe trois types de **glue**s aspectuelle: after, before, et around. La sémantique est similaire à celle de la composition des advices d'AspectJ.

```
Connector aConnector = {
Base Role aBaseRole1, aBaseRole2;
Crosscutting Role aCrosscuttingRole1,
aCrosscuttingRole2;
Glue { aCrosscuttingRole1 before aBaseRole1;
aCrosscuttingRole2 after aBaseRole2;}
```

Figure 1.50. la clause glue

Pour les connecteurs aspectuels binaires (un seul rôle transversal et un rôle de base), la clause glue est tout simplement une déclaration du type glue (Figure 1.49b).

1.3.3.6.3 Exemple illustratif:

Dans cette section, nous présentons la modélisation de la préoccupation "persistance" en utilisant AspectualACME (La Figure 1.51).

Le composant ***persistence*** affecte le composant ***GUI*** et le composant ***Business***. La composition de composant ***persistence*** avec le composant ***GUI*** est modélisée par le connecteur aspectuel ***Persist***. Dans la section attachements, le connecteur ***Persist*** se connecte ***UpdateStateControl*** avec ***registerUser*** et avec ***registerComplaint*** (voir Figure 1.51).

La clause glue de ***Persist*** spécifie que l'élément lié au rôle transversal (source) agit après l'exécution de l'élément lié au rôle de base (cible), cela signifie que chaque fois qu'un utilisateur ou une plainte est déposée, la fonction ***persistence*** est activée par le composant ***persistence***.

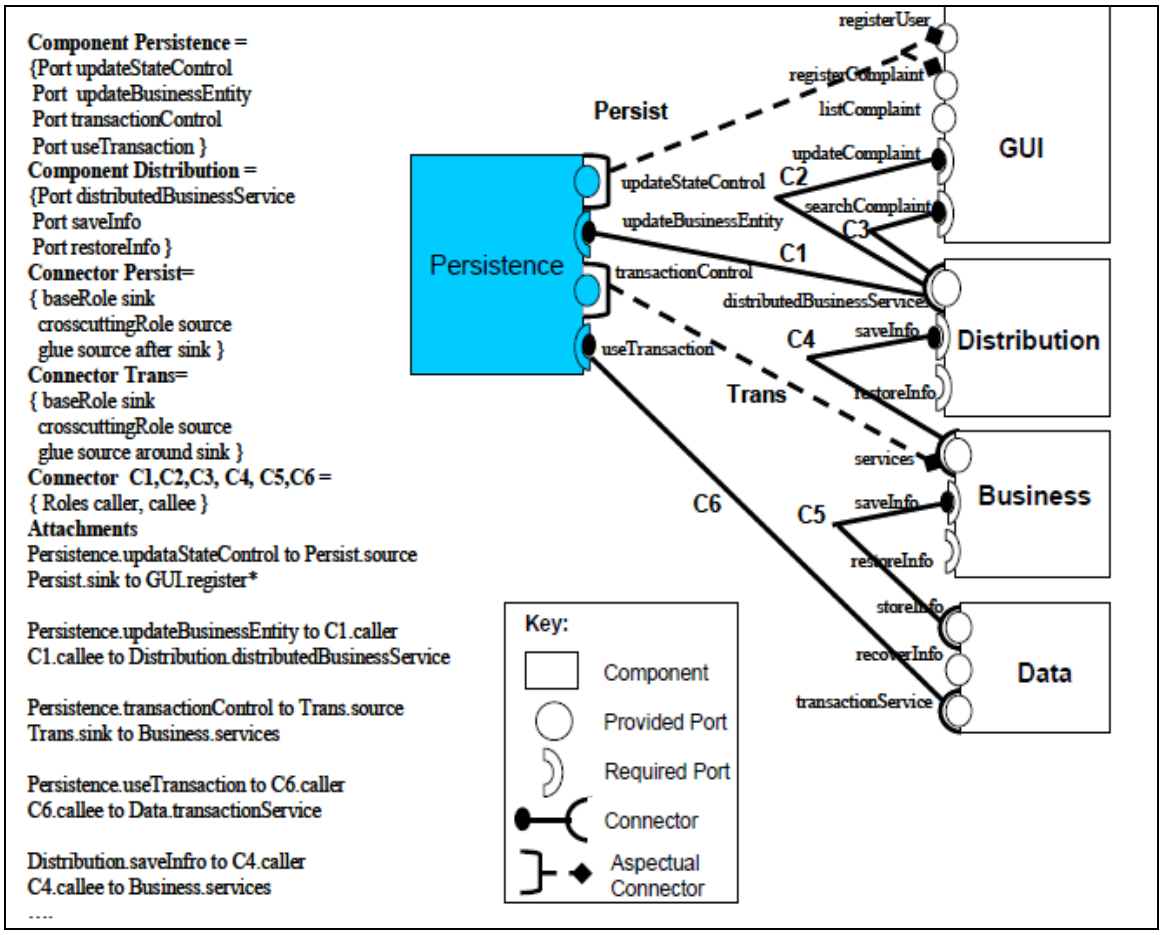



Figure 1.51.Description de la "Persistance" avec AspectualACME tiré de [88]

1.3.3.6.4 Evaluation:

AspectualACME [93] garde la simplicité et l'expressivité d'ACME. Il est facile à étendre et ne comporte aucune nouvelle abstraction pour représenter les systèmes d'architecture orientée aspect, elle adopte donc un simple enrichissement de la sémantique de composition soutenue par les connecteurs de l'architecture. AspectualACME n'introduit pas les concepts de paradigme aspect (tels que, les aspects, points de jonction , et advices).

Most AO ADLs are different from AspectualACME because they introduce a lot of concepts to model AO abstractions (such as, aspects, joinpoints, and advices) in the ADL. DAOP-ADL

L'absence de distinction explicite entre les aspects et les composants augmente la réutilisabilité des composants. En effet, un composant peut jouer un rôle d'aspect pour une application et ce même composant peut être réutilisé comme un composant de base dans une autre application.

Cependant, l'utilisation d'un connecteur aspectuel avec une nouvelle interface diminue la réutilisabilité des connecteurs. De plus, la définition de la

composition entre les différents types de connecteurs et les composants dans une seule et même section présente un inconvénient de cette approche qui affecte la maintenance et l'évolution du système.

| Point de jonction sur trace (Tracecuts) | Type de tissage : dynamique/ Statique | Composition d'aspect | Réutilisation aspect | Spécification du tissage (Application) | Indépendance coupe / aspect | Spécification Coupe | Type de Point de jonction | Type de Conseil | Gestion des Aspects au niveau d'un point de Jonction : Ordre d'exécution des aspects | Type d'advice par point de jonction | Nombre d'advice supporté par un point de jonction |
|---|---|---|---|---|---|---|---|---|---|---|---|
| NON | Revient à établir une connexion entre un composant aspectuel avec un composant de base. | Pas possible | OUI | / | OUI | ACME ADL | Des interfaces d'un composant | After, before, et a round | / | / | / |

Tableau 1.14 Bilan sur AspectualACME

1.3.3.7 AO-ADL

1.3.3.7.1 Introduction :

AO-ADL conserve les principaux blocs architecturaux des ADLs traditionnels qui sont considérés comme suffisants pour spécifier des architectures orientées aspect. AO-ADL est un ADL entièrement nouveau. AO-ADL considère que les composants peuvent modéliser les préoccupations transversales (composants aspectuels) ainsi que les préoccupations non transversales (composants de base) présentant ainsi un modèle de décomposition symétrique.

AO-ADL[92] est un nouveau langage de description d'architecture basé sur XML spécifiquement bien adapté pour décrire les architectures orientées aspects. Comme tous les ADLs traditionnelle les principaux éléments architecturaux d'AO-ADL sont les composants et les connecteurs.

Au lieu d'étendre les ADLs avec les concepts introduits par les langages de programmation orientée aspect (AOP), nous intégrons tous ces concepts (par exemple les points de jonction, pointcuts, etc) dans la définition de composants et de connecteurs. Comme nous l'avons déjà dit plus haut, la principale différence

entre les préoccupations transversale et non transversales est simplement dans le rôle qu'ils jouent dans une composition de liaison particulière et non pas dans le comportement interne lui-même. Par conséquent, au lieu d'inventer un nouvel élément structurel, nous redéfinissons le connecteur et d'étendre sa sémantique avec les connecteurs aspectuelle. Notez qu'un architecte logiciel a aussi la possibilité de spécifier un connecteur classique sans y compris la spécification des rôles aspectuelle ni des connections aspectuelle. Cela signifie que quand il n'est pas un comportement transversal à préciser, l'architecte logiciel peut utiliser des connecteurs comme dans les ADLs traditionnels.

1.3.3.7.2 Les Aspects dans AO-ADL:

Au lieu de définir une nouvelle entité pour modéliser les aspects ou étendre le composant avec un nouveau type d'interface, un composant est considéré comme un aspect quand il participe à une interaction aspectuelle. AO-ADL étend la sémantique des connecteurs traditionnels pour représenter l'effet transversal des composants aspectuels.

1.3.3.7.2.1 Les connecteurs dans AO-ADL:

Dans les ADLs[92] traditionnels les connecteurs sont les bloques utilisé pour les modélisé les interactions entre les composantes et les règles qui régissent ces interactions De même, le connecteur est l'élément architectural pour la composition de AO-ADL, mais étendu avec des fonctionnalités supplémentaires pour soutenir les interactions avec les composants «aspectuelle».

Il ya deux principales différences entre la représentation des connecteurs dans les ADLs traditionnelle, et la représentation des connecteurs en AO-ADL La première différence et la plus pertinente est la distinction entre le binding composant (interactions entre des composants de base) et binding aspectuelle (interactions entre «aspectuelle» et «composants de base»).

Cela signifie que les composants référencés dans la section binding aspectuelle du connecteur sont à jouer le rôle d'un composant aspectuel dans l'interaction spécifié. Ainsi, la deuxième différence est l'utilisation de quantifications pour préciser les rôles du connecteur.

1.3.3.7.2.2 Le tissage d'aspect:

Comme pour le cas de DAOP-ADL, la composition entre les composants de bases et les aspects est définie par un ensemble de règles de composition. Ces règles sont spécifiées dans une section aspect bindings distincte de la section

component bindings qui définie la composition entre les composants de base. Ces règles sont décrites en utilisant le langage XML.

1.3.3.7.3 Exemple illustratif:

La Figure 1.1 montre le connecteur **BalancesManagment** pour le système ATM. Le composant **Bank** est responsable de l'établissement de l'équilibre de composant **Account**. La composition de liaison entre ces éléments est spécifiée dans la clause Binding de composant dans les lignes 17 à 22.

```
1 <connector name="BalanceManagement">
2   <provided-role name="BankMgm">
3     <role-specification>
4       /component/required-interface[@role="AccountMgm"]
5     </role-specification>
6   </provided-role>
7   <required-role name="AccountMgm">
8     <role-specification>
9       /component/provided-interface[@role="BalanceMgm"]
10    </role-specification>
11  </required-role>
12  <aspectual-role name="BalanceReplication">
13    <role-specification>
14      /component/provided-interface[@role="BalanceReplMgm"]
15    </role-specification>
16  </aspectual-role>
17  <componentBindings>
18    <binding name="BMBinding">
19      /connector[@name="BalanceManagement"]/provided-role[@name="BankMgm"] and
20      /connector[@name="BalanceManagement"]/required-role[@name="AccountMgm"]
21    </binding>
22  </componentBindings>
23  <aspectBindings>
24    <aspectual-binding name="BM-ReplicationBinding">
25      <pointcut-specification>
26        /connector[@name="BalanceManagement"]/componentBindings/
27                    binding[@name="BMBinding"] and
28        /operation[@name="setBalance"]
29      </pointcut-specification>
30      <binding operator="after">
31        <aspectual-component
32          aspectual-role-name="/connector[@name="BalanceManagement"]/
33                    aspectual-role[@name="BalanceReplication"]
34          advice-label="/operation[@name="update-balance"]" />
35      </binding>
36    </aspectual-binding>
37  </aspectBindings>
38 </connector>
```

```
39 <component name="Bank">
40   <required-interface role="AccountMgm"
41     uri="//interface[@name="BalanceManagement"]" />
42 </component>
43 <component name="Replication">
44   <provided-interface role="BalanceReplMgm"
45     uri="//interface[@name="BalanceReplication"]" />
46 </component>
```

```
47 <interface name="BalanceManagement">
48     <operation name="setBalance" />
49     ...
50 </interface>
51 <interface name="BalanceReplication">
52     <operation name="update-balance" />
53     ...
54 </interface>
```

Figure 1.52. Exemple de AO ADL connecteur et composant

1.3.3.7.4 Evaluation :

L'adoption [93] d'une approche qui opte pour la représentation des aspects sous forme de composants traditionnels, permet d'augmenter le potentiel de réutilisation des composants. En effet, ces derniers peuvent jouer un rôle

aspectuel ou non en fonction des interactions particulière aux quelles ils participent. Cependant, étant donné qu'AO-ADL est un nouveau langage développé de zéro, ceci constitue un inconvénient de cette approche car ce travail nécessite plus d'effort que d'étendre un langage déjà existant. De plus, il est nécessaire de définir toute une plate-forme pour supporter ce nouveau langage.

Comme dans le cas de DAOP-ADL, l'utilisation du langage XML assure une extensibilité facile en mesure d'intégrer de nouvelles fonctionnalités non initialement prévu et l'architecture peut être générée facilement. Cependant, elle présente un inconvénient concernant la lisibilité de l'architecture logicielle.

Notez qu'AO-ADL permet d'intégrer des langages dédiés pour spécifier la sémantique des composants. Cette fonctionnalité est utilisée par d'autres ADLs (par exemple ACME). Son principal avantage est la possibilité de réutiliser la notation standard existante.

Enfin il faut mentionner qu'aucun générateur de code n'a été mis en place jusqu'à présent.

| | |
|---|---|
| **Point de jonction sur trace (Tracecuts)** | **NON** |
| **Type de tissage : dynamique/ Statique** | **Tissage d'aspect revient à établir un binding entre le composant aspectuel et le composant de base** |
| **Composition d'aspect** | **/** |
| **Réutilisation aspect** | **NON** |
| **Spécification du tissage (Application)** | **Sous forme d'un ensemble de règles de composition** |
| **Indépendance coupe / aspect** | **NON** |
| **Spécification Coupe** | **langage XML** |
| **Type de Point de jonction** | **/** |
| **Type de Conseil** | **After Before** |
| **Gestion des Aspects au niveau d'un point de Jonction : Ordre d'exécution des aspects** | **/** |
| **Type d'advice par point de jonction** | **Hétérogène** |
| **Nombre d'advice supporté par un point de jonction** | **Limité** |

Tableau 1.15 Bilan sur AO ADL

#### 1.3.3.8 DAOP-ADL

##### 1.3.3.8.1 Introduction:

DAOP-ADL a été développée comme complément au modèle CAM (Component Aspect Model) et la plate-forme DAOP (Plate-forme dynamique Orientée Aspect). DAOP-ADL est un ADL orienté aspect qui n'étend aucun autre ADL. Il permet de décrire des architectures logicielles basées sur des composants et des aspects en suivant le modèle CAM. Par la suite, DAOP-ADL devrait être interprété par la plate-forme DAOP.

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

En utilisant ce langage, le modèle de l'application CAM peut être transformé en un ensemble de documents XML qui peut être facilement interprété par un environnement d'exécution - c'est à dire par la plateforme DAOP. La description d'architecture utilisant DAOP-ADL comprend deux parties: on définit les composantes autonomes et les aspects, et la seconde partie est la composition.

Dans ce langage, les composants et les aspects sont des éléments de premier ordre, les connecteurs ne sont pas considérés comme des éléments de l'architecture.

##### 1.3.3.8.2 Les Aspects dans DAOP-ADL:

Les aspects sont représentés par des composants avec un type spécial d'interface appelée interface évaluée (evaluated interface). Cette interface permet de spécifier comment les aspects affectent les interfaces des composants. Les contraintes de composition sont formées par des règles de composition qui déterminent comment connecter les composants entre eux, et des règles d'évaluation des aspects qui déterminent comment et quand appliquer les aspects aux composants pour étendre le comportement du système avec des propriétés aspectuelles (tissage entre les composants et les aspects). Ces règles sont décrites dans une section spécifique, en dehors des composants, en utilisant le langage XML.

1.3.3.8.2.1 Les coupe/ pointcut et les déclarations des advices:

Les points de jonctions supportés par DAOP sont les interfaces publiques d'un composant. Les points de jonction considérés sont les messages émis et reçus entre les composants ainsi la création ou destruction d'un composant. Les coupes sont définis au niveau du langage d'architecture DAOP-ADL, donc extérieurement aux aspects.

1.3.3.8.2.2 Le tissage d'aspect:

DAOP-ADL est le langage d'architecture qui a été conçu pour être interprété par la plate-forme DAOP. DAOP fournit un mécanisme de composition qui connecte les aspects et les composants dynamiquement à l'exécution. L'architecte logiciel va utiliser DAOP-ADL au moment de la conception pour décrire l'architecture de l'application. Plus tard cette information sur l'architecture est chargée dans la plate-forme DAOP, utile pour établir les connexions entre les composants dynamiques autonomes et les aspects.

Les contraintes de composition sont formées par des règles de composition qui déterminent comment connecter les composants entre eux (tissage entre les composants et les aspects). Ces règles sont décrites dans une section spécifique, en dehors des composants, en utilisant le langage XML.

1.3.3.8.3 Exemple illustratif:

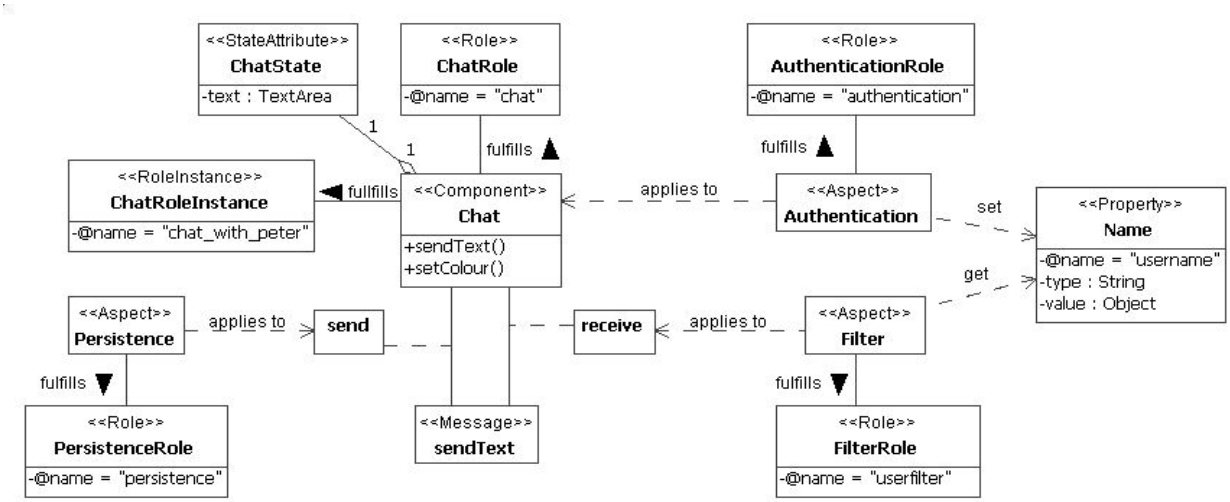



Figure 1.53. Un exemple de spécification d'architecture de l'application Chat avec CAM[11]

- **Spécification des composants autonome et les aspects:**

La Figure 1.54 montre la spécification DAOP-ADL des aspects autonomes et composants comprenant le système de la Figure 1.53. Les propriétés sont également spécifiées ici (chat, l'authentification, la persistance, filtre).

[11] CAM (Component-Aspect Model) est la notation visuelle de DAOP-ADL

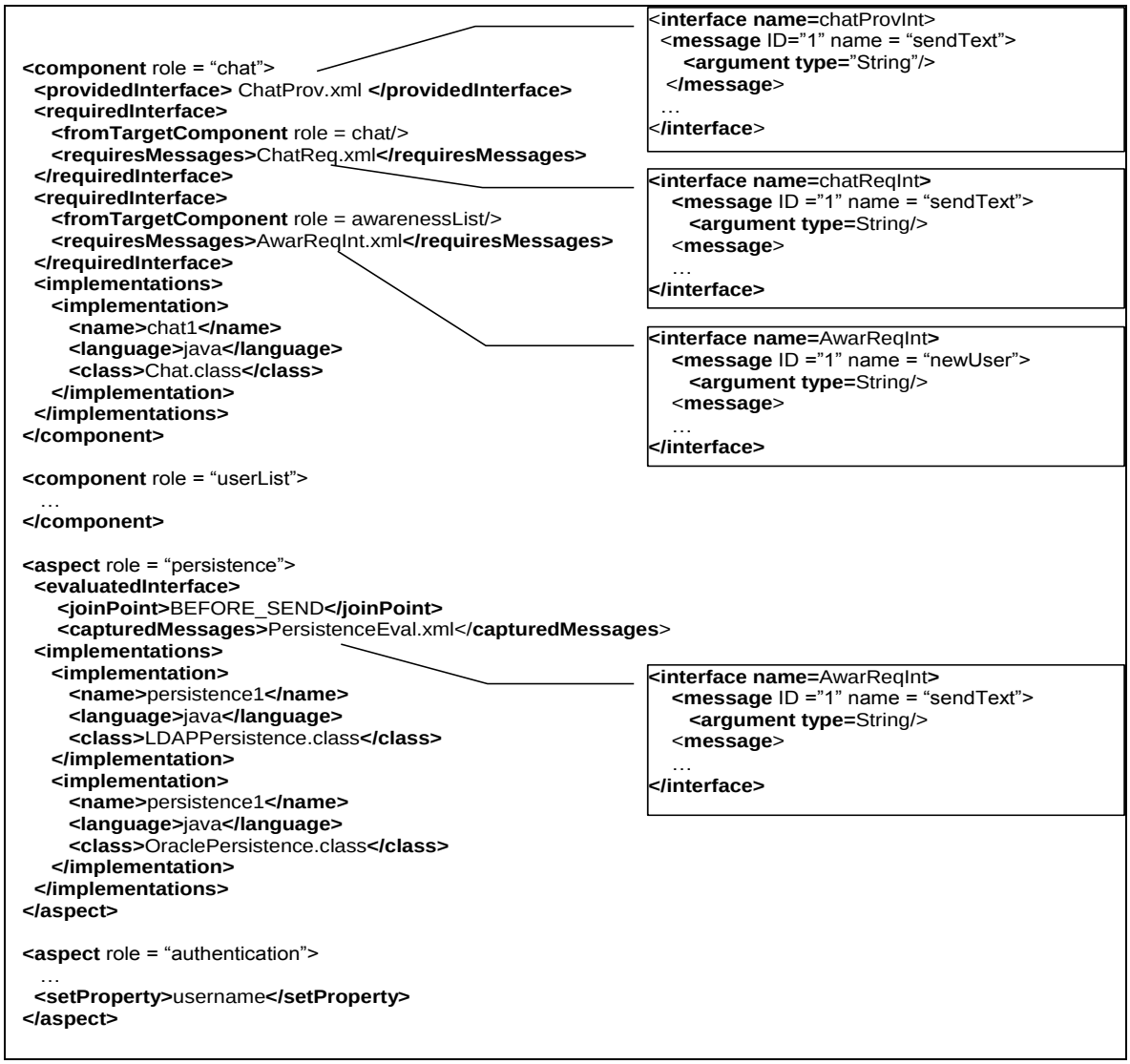



Figure 1.54. Spécification d'architecture (composants et aspects) de l'application Chat avec DAOP-ADL

- **Spécification des règles de composition:**

Cette spécification inclut la définition de coupes (pointcuts) , qui dans CAM / DAOP sont définies en utilisant la langage DAOP-ADL et jamais comme une partie de la définition ou l'implémentation des composants et des aspects.

```
<compositionRules>
   <componentCompositionRules>
     <compositionsRuleFor role="chat">
       <componentRule>
         <formalRole>awarenessList</formalRole>
         <realRole>userList</realRole>
       </componentRule>
     </compositionsRuleFor>
   </componentCompositionRules>
   <aspectEvaluationRules>
     <createComponent role=chat>
         <BEFORE_NEW>                          //rule to apply the authentication aspect
          <concurrent>
             <aspect  role ="authentication/>
           </concurrent>
         </BEFORE_NEW>
     </createComponent>
     <sendMessage>
        <source-comp>
           <roles>chat</roles>
        </source-comp>
        <target-comp>
           <roles>chat</roles>
        </target-comp>
        <targetMessages>
           <message name="sendText"/>
        </targetMessages>
        <BEFORE-SEND>                         //rule to apply the persistence aspect
          <concurrent>
             <aspect  role ="persistence"/>
           </concurrent>
        </BEFORE_SEND>
         <BEFORE_RECEIVE>                     //rule to apply the userfilter aspect
          <concurrent>
             <aspect  role ="userfilter"/>
           </concurrent>
         </BEFORE_RECEIVE>
      </sendMessage>
  </aspectEvaluationRules>
</compositionRules>
```

Figure 1.55. Spécification des règles de composition avec DAOP-ADL de l'application CHAT.

1.3.3.8.4 Evaluation :

Etant donné que DAOP-ADL est un nouveau langage développé de zéro, il nécessite donc plus d'effort que d'étendre un langage déjà existant. De plus, il nécessite toute une plate-forme pour le supporter (plate-forme DAOP).

La distinction visuelle entre le composant traditionnel et le composant aspectuel facilite la tâche de maintenance. Cependant, cette distinction diminue la chance de réutilisabilité des composants.

De plus, la composition entre les composants de base d'une part, et la composition entre les composants de base et les aspects d'autre part, sont définies dans une même section. Ceci présente un inconvénient de cette approche. En effet, la combinaison des compositions traditionnelles et celles aspectuelles, dans une seule et même section ajoute de la complexité à la phase de maintenance et de raffinement du système. En outre, l'utilisation du langage XML permet au langage DAOP-ADL d'être facilement extensible et l'architecture peut être facilement générée. Cependant, l'utilisation du langage XML constitue un

inconvénient en ce qui concerne la lisibilité de l'architecture logicielle (augmentation tries rapide de la taille des fichiers XML).

Enfin, il faut dire que ce langage ne supporte pas les mécanismes de quantification architecturale et aucun générateur de code n'a été mis en place jusqu'à présent.

| | |
|---|---|
| **Point de jonction sur trace (Tracecuts)** | **NON** |
| **Type de tissage : dynamique/ Statique** | **Dynamique a l'exécution** |
| **Composition d'aspect** | **/** |
| **Réutilisation aspect** | **NON** |
| **Spécification du tissage (Application)** | **Sous forme d'un fichier XML** |
| **Indépendance coupe / aspect** | **NON** |
| **Spécification Coupe** | **DAOP ADL** |
| **Type de Point de jonction** | **createComponent, destroyComponen, sendMessage, receiveMessage,ThrowEvent** |
| **Type de Conseil** | **BEFORE_SEND , AFTER_SEND BEFORE_RECEIVE AFTER_ RECEIVE** |
| **Gestion des Aspects au niveau d'un point de Jonction : Ordre d'exécution des aspects** | **/** |
| **Type d'advice par point de jonction** | **Hétérogène** |
| **Nombre d'advice supporté par un point de jonction** | **Illimités** |

Tableau 1.16 Bilan sur DAOP-ADL

## 1.4 Conclusion

Dans ce chapitre, nous avons présenté l'architecture logicielle et les principaux concepts des langages de description d'architecture. Nous avons présenté un aperçu de différents travaux sur l'emploi des aspects pour la simplification du développement des applications pour les plates-formes à base de composants, Les architectures logicielles et quelques modèles de composants ainsi que nous avons réalisé une étude comparative sur les différents ***travaux sur l'intégration des aspects et des composants et on a terminé chaque étude par une évaluation de cette dernière .***

Ensuite, nous avons montré l'importance de l'intégration des concepts de l'orienté aspect au niveau architectural en présentant la discipline de développement logiciel orienté aspect, AOSD. Nous nous sommes intéressés à l'intégration de l'AOSD et de l'ADL pour aboutir à des langages de description d'architecture orientés aspect. Enfin, nous avons présenté une étude approfondie

des différents ADLs orientés aspect existants qui suit soit une approche symétrique ou une approche asymétrique.

Sur la base des connaissances établies suite à notre étude bibliographique, nous avons eu l'idée de définir un langage de description d'architecture orienté aspect. Nous proposons dans le chapitre suivant notre approche en détails.

| | Spring AOP | JBoss-AOP | JAC |
|---|---|---|---|
| Point de jonction sur trace (Tracecuts) | NON | NON | NON |
| Type de tissage : dynamique/ Statique | Dynamique | dual dynamique/ statique | Dynamique lors de l'exécution. |
| Composition d'aspect | (possible) L'implémentation de l'interface | Non supporté (Juste des mécanismes de Détection de conflits) | (possible) Défini avec le système jac.comp.wrappingOrder. |
| Réutilisation aspect | OUI | génération de proxies | OUI |
| Spécification du tissage (Application) | A travers des proxys (JDK dynamic proxy) | OUI | Le mécanisme RTTI (Run-Time Type Information) |
| Indépendance coupe / aspect | OUI | OUI | OUI |
| Spécification Coupe | (d'expressions régulières Java) Par annotations /fichiers de | (d'expressions régulières Java) Par annotations /fichiers de configuration | d'expressions régulières java |
| Type de Point de jonction | Exécution | Exécution méthode, constructeur, attribut, all, et méthode call | exécution de méthode, constructeur |
| Type de Conseil | Before, After returning, After throwing, After (finally), Around | Around Avant ou après un point de jonction | around |
| Gestion des Aspects au niveau d'un point de Jonction : Ordre d'exécution des aspects | Spécifié explicitement en implémentant l'interface org.springframework.core.Or | (il n'y a pas de stratégie) Juste des mécanismes de Détection de conflits | Spécifié explicitement on définissant la propriété jac.comp.wrappingOrder |
| Type d'advice par point de jonction | Hétérogène | Non supporté | Homogène |
| Nombre d'advice supporté par un point de jonction | illimité | Non supporté | illimité |

| | | | |
|---|---|---|---|
| OUI | NON | NON | NON |
| Dynamique<br>A l'exécution | Dynamique<br>(ajouter les connexions à l'exécution) | Dynamique A l'exécution<br>Ou statique.<br>Selon le modèle d'implémentation | Selon le langage d'implémentation<br>Statique à la compilation dans AspectJ |
| (possible)<br>stratégie de précédence/<br>stratégie de combinaison | Par instanciation de connecteur | (possible)<br>Programmable a l'aide de ses ACI/<br>analyses statiques | (Possible)<br>stratégie de précédence |
| OUI | OUI | OUI | OUI |
| A travers le JAsco run-time weaver | A travers des connecteurs | génère des classes de proxies /<br>(weavlet) | Selon le langage d'implémentation:<br>Machine virtuel java dans |
| NON | OUI | OUI | NON |
| (d'expressions régulières Java) | Expression régulière java | (d'expressions régulières Aspectj) | Expression régulière dans *TinyAspect* /<br>expression java ordinaire |
| Execution call, des Annotations java 1.5. | Points de jonction abstraits (selon le langage d'implémentation) | Explicit/<br>pas d'introduction | *call* |
| Before, After , after throwing , after returning Around | Non spécifié<br>(selon le langage d'implémentation) | Selon le modèle d'implémentation (dans CaesarJ même type qu'aspectJ) | Before After Around |
| Spécifié explicitement on spécifiant une stratégie de combinaison d'aspects | Spécifié implicitement selon l'ordre d'application des aspects, | Spécifié explicitement on définissant un ordonnancement arbitraire entre aspects à l'aide d'interfaces de collaborations, | Spécifié explicitement on définissant une stratégie de précédence entre les |
| Hétérogène | Non spécifié<br>(selon le langage d'implémentation) | Hétérogène | Hétérogène |
| illimité | illimité | illimité | illimité |
| JAsCo | Aspectual Component | Caesar | Open module |

| | | | | | |
|---|---|---|---|---|---|
| NON | OUI | NON | NON | NON | NON |
| Tissage d'aspect revient à établir le plan de base et le plan décrivant la préoccupation à intégrer | Tissage d'aspect revient à établir un ensemble de liaisons d'aspect (non implémenté) | Dynamique à l'exécutions | Dynamique a l'exécution | Pas de notion de tissage d'aspects | Configuration architecturale ensemble de connexions entre les PCD (coupes) et |
| possible | (Non supporté) juste un mécanisme de résolution de conflit | Pas possible | / | Pas possible | Pas possible |
| OUI | OUI | OUI | NON | NON | OUI |
| Sous forme d'un ensemble de règles de transformation | Avec FRACTAL-ADL ou FRACTAL EXPLORER | Mécanisme de tissage Basé sur la plate forme DAOP | Sous forme d'un ficher XML | / | Dans la partie configuration architecturale |
| OUI | OUI | OUI | NON | NON | OUI |
| SafArchie ADL | (d'expressions régulières FRACTAL ADL) | Langage d'architecture DAOP-ADL | DAOP ADL | LEDA ADL | AC2- ADL |
| Introduction et reconfiguration | Introduction et exécution de méthode | createComponent, destroyComponen, sendMessage, ThrowEvent | createComponent, destroyComponen, sendMessage, | méthode call | component_JP, interfaces_JP, operation_JP et |
| beforeCall, afterCall, beforeExecute, afterExecute, beforeResponse, afterResponse | After Before Around | BEFORE_NEW, AFTER_NEW, EFORE_SEND, BEFORE_DESTROY, AFTER_DESTROY, AFTER_SEND, BEFORE_RECEIVE, AFTER_RECEIVE , | BEFORE_SEND , AFTER_SEND BEFORE_RECEIVE AFTER_ RECEIVE | / | Before after around introduce |
| Spécifié implicitement d'une façon absolue, pas de mécanisme de détection de conflits | fait implicitement selon leur ordre d'exécution | / | / | / | / |
| Hétérogène | Hétérogène | Hétérogène | Hétérogène | / | Hétérogène |
| illimité | illimité | illimité | Illimités | / | illimité |
| TranSAT | FAC | CAM/DAOP-ADL | DAOP-ADL | AspectLEDA | AC2-ADL |

| | | |
|---|---|---|
| **Aspectual ACME** | **NON** | **NON** |
| **/** | **Revient à établir une connexion entre un composant aspectuel avec un composant de base.** | **Tissage d'aspect revient à établir un binding entre le composant de base et le composant aspectuel** |
| **/** | **Pas possible** | **/** |
| **/** | **OUI** | **NON** |
| **/** | **/** | **Sous forme d'un ensemble de règles de composition** |
| **/** | **OUI** | **NON** |
| **/** | **ACME ADL** | **langage XML** |
| **/** | **Des interfaces d'un composant** | **/** |
| **/** | **After, before, et a round** | **After Before** |
| **/** | **/** | **/** |
| **/** | **/** | **Hétérogène** |
| **/** | **/** | **Limité** |
| **AO-ADL** | | |

Tableau 1.17 Bilan sur l'orienté aspect

# CHAPITRE 2 : IASA

## 2.1 Introduction:

Les approches à base de composants apparaissent de plus en plus incontournables pour le développement de systèmes et d'applications répartis. Il s'agit de faire face à la complexité sans cesse croissante de ces logiciels et de répondre aux grands défis de l'ingénierie des systèmes : passage à grande échelle, administration, autonomie.

Après les objets dans la première moitié des années 1990, les composants se sont imposés comme le paradigme clé de l'ingénierie des intergiciels et de leurs applications dans la seconde moitié des années 1990. L'intérêt de la communauté industrielle et académique s'est d'abord porté sur les modèles de composants pour les applications comme EJB, CCM ou .NET. À partir du début des années 2000, le champ d'application des composants s'est étendu aux couches inférieures : systèmes et intergiciels. Il s'agit toujours, comme pour les applications, d'obtenir des entités logicielles composables aux interfaces spécifiées contractuellement déployables et configurables ; mais il s'agit également d'avoir des plates formes à composants suffisamment performantes et légères pour ne pas pénaliser les performances du système. Le modèle de composants IASA remplit ces conditions.

L'approche IASA consiste à considérer simultanément dans une spécification, les aspects abstraits et concrets, les aspects structurels et comportementaux et l'aspect spécification de propriétés non fonctionnelles. Cette approche repose sur des modèles de composants et de connecteurs orientés vers le support naturel de ces divers aspects. Le but du modèle à composants IASA est d'offrir un cadre architectural global pour le développement d'applications à base de composants. Contrairement à d'autres modèles tels que les EJB et CCM qui offrent un modèle concret du composant, IASA regroupe une hiérarchie de modèles.

Dans la suite, on présente en détail les éléments fondamentaux du modèle de composant. On commencera par les éléments modélisant la vue externe, à savoir le concept de point d'accès et le concept de ports. On présentera ensuite le concept d'enveloppe et nous étudierons le modèle de la vue interne. Les autres aspects liés au modèle de composant, notamment les états et la validation de composant seront présentés en fin de cette section.

## 2.2 Le modèle de composants IASA :

Le composant est l'un des éléments fondamentaux de définition d'une architecture logicielle. Le modèle de composant distingue entre deux grandes catégories de composants **(Figure 2.1)**: Les composants *primitifs* et les composants *composites*.

L'objectif de l'élaboration d'une application consiste à réaliser un composant composite. Ainsi, à l'exception des connecteurs, tout est composant.

Le modèle de composant définit deux vues dans un composant [43], une ***vue externe*** à laquelle doit adhérer n'importe quel composant et une ***vue interne***, applicable exclusivement aux composites.

La vue externe est représentée par le concept d'enveloppe ou sont localisés les ports modélisant le composant. La vue interne est organisée en deux grandes parties : la partie ***opérative*** et la partie ***contrôle***.

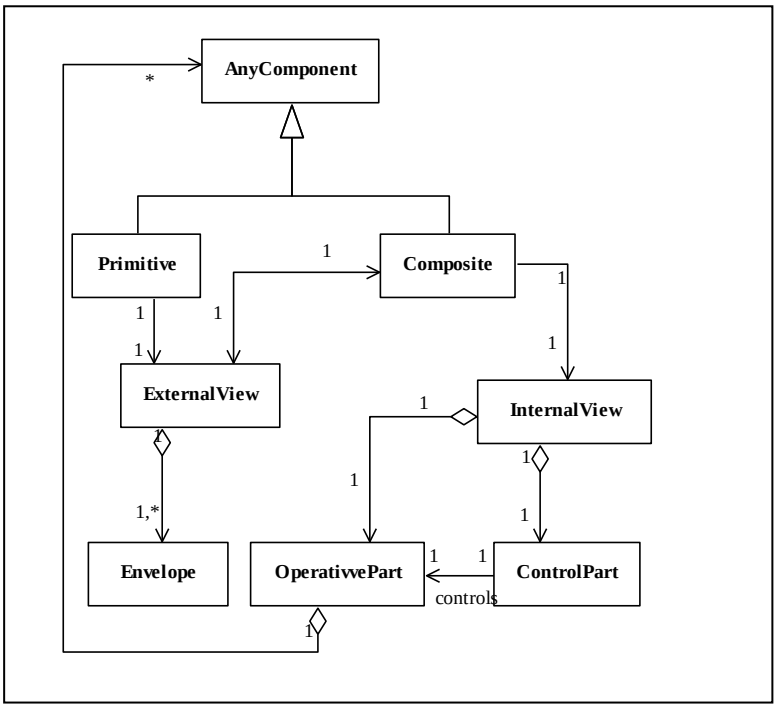



Figure 2.1: Diagramme de classe du modèle de composant

Un composant est un type. Tous les composants sont des sous types du type *AnyComponent*. Ce dernier renseigne sur les éléments fondamentaux d'un composant à savoir le nom d'instance, une liste de ports et une liste d'enveloppes applicables.

Deux sous type ont été défini pour distinguer clairement entre composants primitifs et composants composites. Ce sont les types *PrimitiveComponent* et *CompositeComponent.* Le type *CompositeComponent* indique qu'il y'a une organisation interne du composant en terme de partie opérative et partie contrôle. Il maintient aussi, plusieurs listes représentant son état structurel (liste de composants, liste de connecteurs).

### 2.2.1 La vue externe et le concept d'enveloppe :

L'instanciation d'un composant est réalisée dans le contexte du concept d'enveloppe. Une enveloppe permet d'isoler l'instance pure d'un composant de son environnement d'exploitation en fournissant à ce dernier les éléments nécessaires à l'exploitation de l'instance. L'enveloppe est l'endroit ou seront

solutionnés les divers problèmes liés au déploiement de l'instance du composant et à la spécification de topologies très variées, notamment celle mettant en œuvre directement les points d'accès de port.

La vue externe de tout composant est formée d'une enveloppe munie d'un certain nombre de ports comme montre la **Figure 2.2**. Le concept d'enveloppe a été introduit pour permettre d'atteindre les objectifs suivants :

- ✓ L'isolation totale la vue interne d'un composant du monde externe.

- ✓ Offrir le support nécessaire à la réalisation de topologies qu'il n'est pas possible de réaliser dans le contexte de ports typés par les interfaces, comme ceci est le cas dans les divers ADL et UML.

- ✓ Offrir le support nécessaire au déploiement des instances d'un composant. Selon son environnement d'existence, une instance sera enveloppée par l'enveloppe adéquate (Une instance s'habille adéquatement pour une situation particulière).

- ✓ Offrir le support nécessaire à la réalisation des opérations de validation.

- ✓ La transformation de n'importe quel composant provenant de n'importe quelle source, en un composant prêt à être exploité dans les opérations d'assemblage selon le modèle de composant.

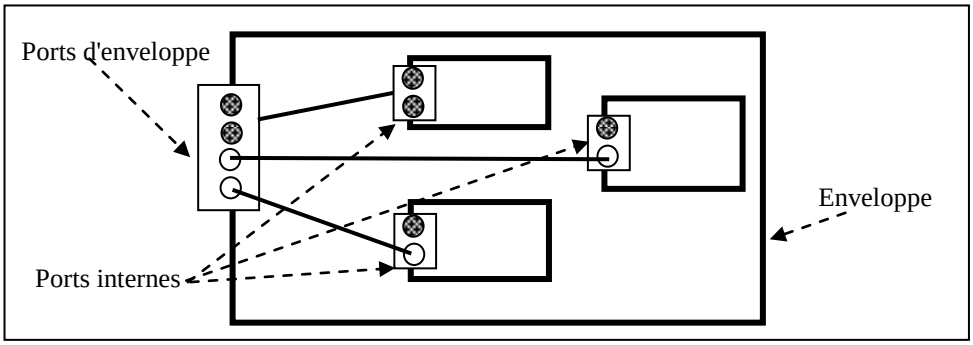



Figure 2.2- Enveloppe, port d'enveloppes et connexion non supportés par les ADL actuels

Pour atteindre ces divers objectifs, un composant est en fait associé à plusieurs enveloppes. Selon l'objectif visé, une enveloppe spécifique est alors appliquée à une instance de composant. Ainsi pour la validation il faut appliquer une enveloppe dotée de support pour la validation. Pour la génération de l'application, il faut utiliser une enveloppe dotée de supports adéquats, notamment les adaptateurs de ports.

Nous avons pour l'instant identifié deux sortes d'enveloppes [42] : Les enveloppes marquées et les enveloppes de déploiement **(Figure 2.3)**.

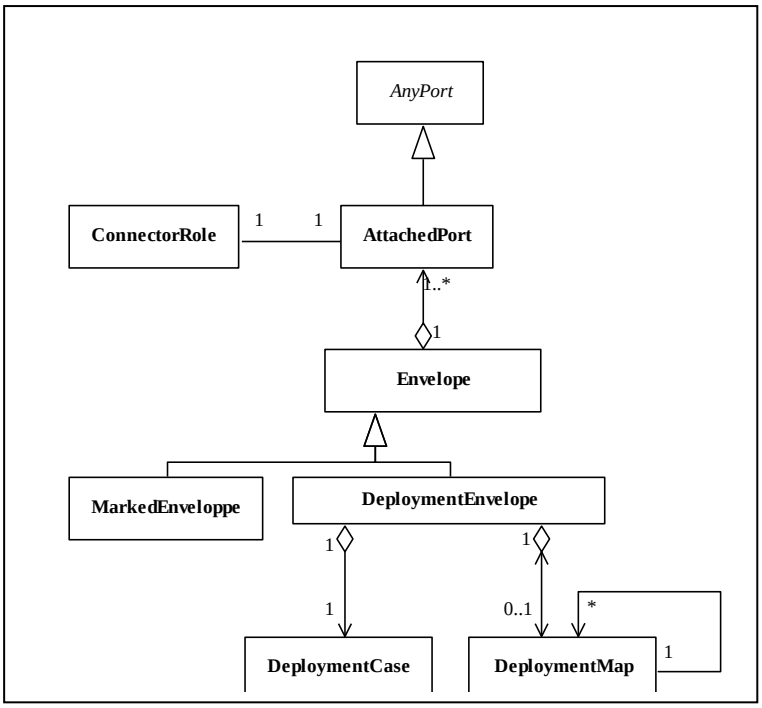



Figure 2.3- Diagramme de classes de l'enveloppe

2.2.1.1 Les Enveloppes Marquées :

Les enveloppes marquées sont utilisées dans le processus de vérification de l'état global (stabilité) d'un composant. Dans de telles enveloppes, il est possible de marquer totalement ou partiellement les points d'accès. Le marquage d'un point d'accès signifie que ce dernier est correctement connecté. Une enveloppe totalement marquée est une enveloppe dont tous les points d'accès sont marqués. Dans une enveloppe marquée partiellement vers l'extérieur seules les services requis (*ClientDataPoint*) et les *DataPoint* dans le sens est *in* ou *inout* sont marqués. Dans une enveloppe marquée partiellement vers l'intérieur, seules

les services fournis (*ServerActionPoint*) et les *DataPoint* dans le sens est *out* ou *inout* sont marqués.

#### 2.4.1.2 Les Enveloppes de Deploiement :

Une enveloppe de déploiement est dotée d'un cas de déploiement et d'un plan de déploiement opérationnel. Ce dernier détermine les cas de déploiement effectif de toutes les instances dans les divers niveaux de la hiérarchie de composition. Grâce à ce concept, il est très possible d'associer à des instances d'un même type de composant, des cas de déploiement totalement différents. L'application d'une enveloppe de déploiement se fait soit à l'instanciation d'un composant dans le processus de construction d'un composite soit au type de composant. Dans ce dernier cas, l'application de l'enveloppe a pour but de générer l'application.

Les enveloppes de déploiement prennent en charge la résolution des divers aspects liés au cas de déploiement correspondant. Ils offrent d'une part, le support nécessaire pour la faisabilité des diverses connexions et d'autre part, ils abritent, au niveau implémentation, les rôles des connecteurs avec les adaptateurs nécessaires et attachent ces rôles aux ports concrets. Les ports de l'enveloppe reprennent aux ports internes les ports concrets, qu'ils réorganisent si nécessaire au moment de l'instanciation selon le cas de déploiement spécifié et selon les connecteurs utilisés.

Une enveloppe de déploiement, peut aussi être appliquée à un ensemble d'instances d'un composite. Ceci se produit lorsque nous désirons affecter à plusieurs instances un même cas de déploiement. Cependant ces instances enveloppées ne forment pas un type pouvant être par la suite réutilisé.

### 2.2.2 La vue Interne du composant IASA:

La vue interne est organisée en deux parties (Figures 2.1 et 2.4). Une ***partie opérative*** et une ***partie contrôle***.

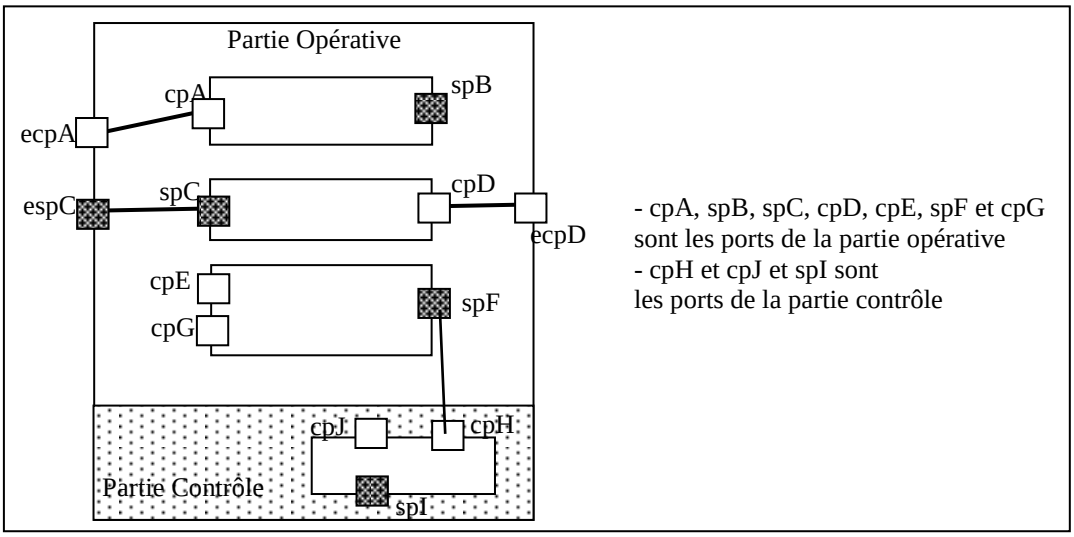



Figure 2.4-Vue interne d'un composant composite

#### 2.2.2.1 La partie opérative :

Elle contient les composants représentants par leurs interconnexions la logique générale du composite à un moment bien précis. Elle est représentée par le type *OperativePart***.** Parmi les éléments importants de la partie opérative, nous retrouvons la liste de ports qui seront liés aux ports de l'enveloppe par des connecteurs de délégation.

Les instances de composants et connecteurs sont soit statiques soit dynamiques. Une instance statique est définie en tant que telle à la spécification. Elle est toujours présente dans la partie opérative et ne peut en être supprimée.

Une instance dynamique peut disparaître et apparaître durant tout le cycle de vie de l'instance de son composite. Elle peut ainsi être créée, s'auto détruire ou être supprimée.

En plus des instances de types publiques de composants, la partie opérative peut comporter l'instanciation de type de composants internes. L'instanciation de ces types internes n'est possible qu'au niveau de la partie opérative du composant dans lequel les types internes ont été définis.

Les types internes ont été introduits pour représenter des aspects très spécifiques à un composant. Les interfaces homme machine sont un exemple d'applications où il y a une mise en œuvre intense du concept de types internes de composant.

2.2.2.2 La partie contrôle :

La partie contrôle, représentée par le type *ControlPart* (Figure 2.5), réalise les diverses opérations de contrôle sur les composants de la partie opérative, tels que la gestion du flux de contrôle des divers services (arrêt, lancement en séquence ou en parallèle), le contrôle de l'évolution structurelle, la gestion des exceptions, l'exportation des états du composant et la génération de log.

Les opérations de la partie contrôle sont assurés par quatre composants spécifiques (Figure 2.5) :L'*OpPartController*, l'*OpPartStateCmp***,** **l'***OpPartExceptionCmp* et l'*OpPartLogCmp*. Malgré que la partie contrôle est conçue pour contenir n'importe quel type de composant**,** et de ce fait permettrait de supporter des comportements très complexes**,** nous nous sommes imposé les limites suivantes : la partie contrôle ne peut contenir qu'une seule instance des types de composants qui lui sont spécifiques. L'instance de l'*OpPartController* est obligatoire alors que les instances des trois autres types sont optionnelles.

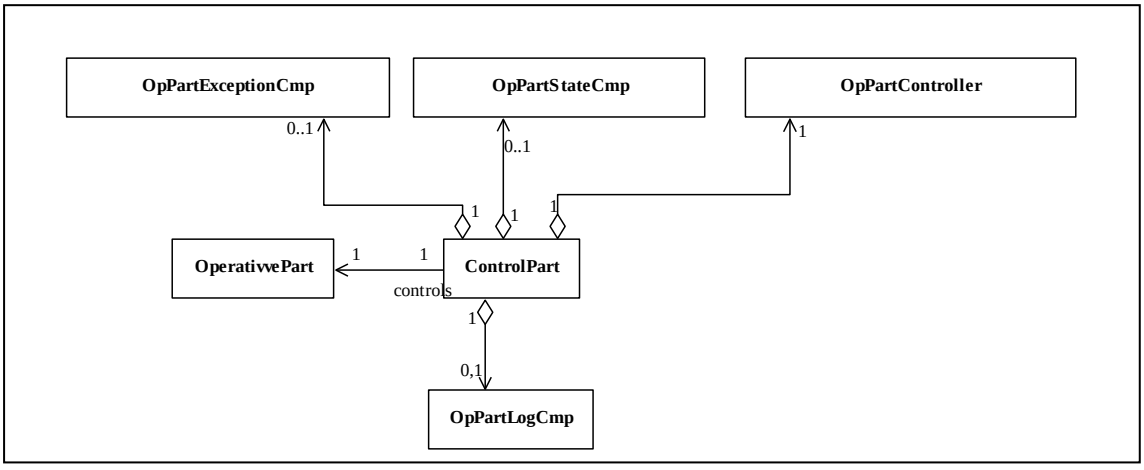



Figure 2.5- Diagramme de classe de la partie contrôle

Un *OpPartController* est un composant dit comportemental dont la réalisation est faite complètement en langage d'action SEAL. Un composant comportemental est un type primitif dans le sens ou il ne dispose pas de structure interne accessible.

Les ports d'un composant comportemental permettent de sélectionner un comportement interne ou charger un comportement externe.(Figure 2.6)

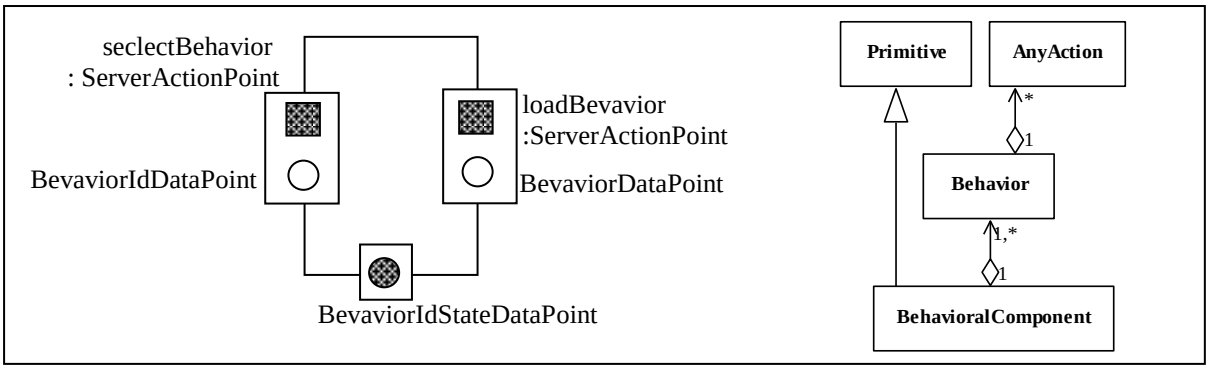



Figure 2.6: Modèle, ports et points d'accès d'un composant comportemental

Grâce aux concepts "partie contrôle" et composant de contrôle de la partie opérative, il devient possible de spécifier pour un ensemble de composants, diverses architectures. Cette tendance pourrait ouvrir la voie vers des architectures basées sur une catégorie bien précise de composants, et un nombre limité d'instances, dû par exemple à un nombre limité de licences.

Le fonctionnement de l'architecture sera déterminé à partir du plan d'interconnexion qui sera établi dynamiquement par le composant de contrôle (*OpPartController*)

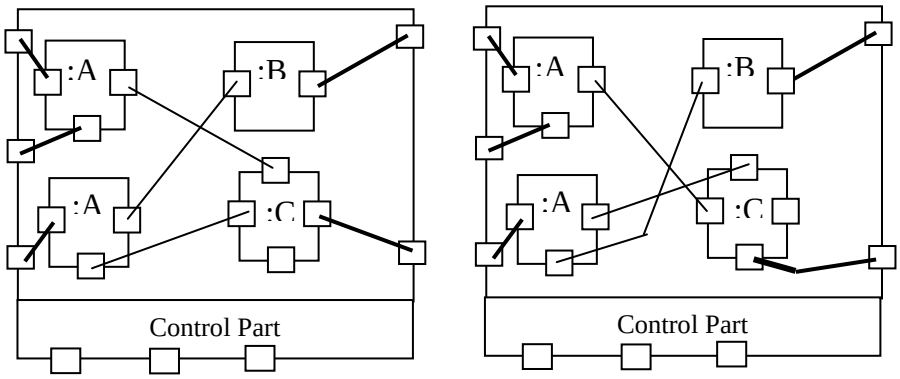



Figure 2.7: Topologies différentes utilisant les mêmes types de composants

Les actions qu'applique le contrôleur (*OpPartController)* à la partie opérative font partie d'un contexte d'action bien précis (*ControlActionContext*) du langage SEAL. Ce contexte adresse entre autres les aspects suivants :

- La sélection/chargement d'un comportement
- Le contrôle global, partiel ou individuel des composants de la partie opérative (activation/désactivation)
- Gestion du transfert du flux de contrôle entre les composants de la partie opérative (lancement séquentiel ou parallèle de service)
- L'évolution dynamique de la partie opérative (création/ destruction de composants et de connecteurs)**.**

A Chaque instanciation ou suppression de composant dans une partie opérative, l'instance créée ou supprimée reporte cette information (nom d'instance, nom de type) à l'*OpPartController* du composite dans lequel elle a été instanciée ou supprimée. Le composant *OpPartController* reporte au composant d'états, l'état structurel initial et tout changement structurel durant l'exécution.

2.2.2.2.1 Les Composants de la partie controle :

En plus du composant contrôleur que nous avons présentés précédemment, la partie contrôle contient trois composants spécifiques : Le composant *d'état,* le composant de *gestion des exceptions* et le composant de *logs*.

1. Le Composant D'etat **(**OpPartStateCm*p*) :

Le composant d'états est conçu pour être utilisé uniquement dans la partie contrôle. Il est utilisé pour collecter les divers états de la partie opérative. Il gère plusieurs listes et attributs décrivant la partie opérative à un moment donné. Les diverses listes sont exportées vers le monde extérieur à travers le port d'états (*StatePort*) dont est doté chaque composant composite.

Un composant d'états est instancié soit en mode actif soit en mode passif. Un composant d'états, instancié en mode actif, se connecte automatiquement au contrôleur et à tous les ports d'états des instances de composants de la partie opérative, qu'ils soient crées statiquement ou dynamiquement. Il se déconnecte d'un composant lorsque ce dernier est en phase de suppression. En mode passif,

les connexions du composant d'états aux autres composants du composite sont établies de manière explicite.

La connexion entre le contrôleur et le composant d'états est utilisée par le contrôleur pour envoyer au composant d'états l'état structurel de la partie opérative. Ces opérations de transferts ont lieu à chaque opération impliquant un changement de la structure (ajout / suppression / activation / désactivation de composant, de ports et de connecteurs).

2. LE Composant de gestion des exception (*OpPartExceptionCmp*) :

Ce composant joue le rôle principal de mise en zone de garde des divers composants, sources d'exception. Si tous les composants sont mis en zone gardée, alors le composite ne sera pas une source d'erreurs pour ses utilisateurs. Le composant de gestion des exceptions peut être instancié soit en mode actif ou passif. En mode actif il met automatiquement tous les composants sources d'erreurs en zone gardée. En mode passif, c'est l'architecte qui doit explicitement tracer les chemins de la gestion des exceptions. L'instanciation en mode actif n'exclu pas le fait que l'architecte peur intervenir sur la topologie de gestion des exceptions

3. Le composant LOG (OpPartLogCmp) **:**

Les objectifs suivants ont été à l'origine de la nécessité de doter la partie contrôle de ce composant spécifique.

- ✓ Faire ressortir de manière explicite l'aspect **log** dans les applications.
- ✓ Définir un lieu commun ou seront spécifiés les divers formats de log.
- ✓ Permettre de transmettre les logs vers diverses destinations telles qu'un fichier, une base de données ou une quelconque application intéressée par l'analyse des logs.

Comme les autres composants, le composant de logs peut être instancié en actif ou en passif. L'instanciation en mode actif permet à ce composant de se connecter automatiquement à tout port de log de n'importe quel composant de la partie opérative.

2.3 Le point d'accès :

Le point d'accès est le plus petit élément manipulable dans une architecture. Il représente l'élément de base pour la définition d'un port. Toute information, quelque soit sa nature arrive ou part d'un composant à travers un point d'accès. Il permet de localiser les diverses ressources fournies ou requises à travers un port et de renseigner sur les éléments intervenant dans la réalisation d'un comportement observable sur un port.

Contrairement aux autres modèles de ports, où le concept de point d'accès correspond à toute une méthode, les points d'accès dans notre modèle représentent les concepts de base échangés entre deux ports de composants. Pour l'instant, les concepts supportés par les points d'accès se réduisent à l'échange de données et au transfert du flux de contrôle. Ainsi, un paramètre d'une méthode peut être associé à un point d'accès vu qu'il véhicule une information.

Un point d'accès peut être manipulé indépendamment des autres points d'accès. Il est ainsi possible de tirer une connexion d'un point d'accès d'un port de composant vers un autre point d'accès d'un autre port, sans que les autres points d'accès des deux ports n'interviennent dans cette connexion (Figure 2.8). De cette manière il deviendrait possible d'utiliser un paramètre indépendamment de la méthode dans laquelle il est défini.

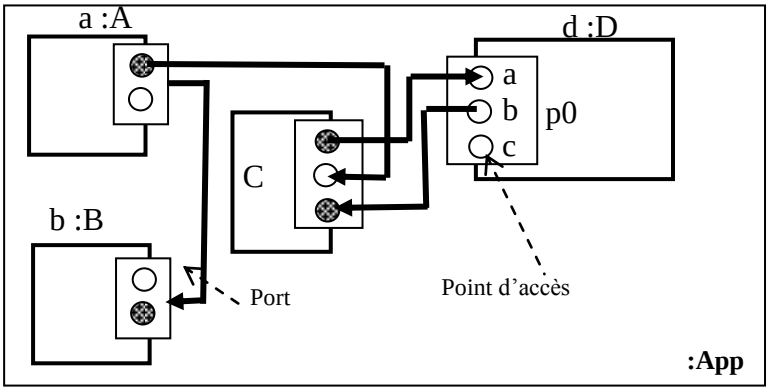



Figure 2.8- Connexions utilisant les points d'accès

2.3.1 Les types fondamentaux relatifs aux points d'accès :

Les points d'accès sont tous représentés par le type de base *AnyPoint (*Figure 2.9**).** Le type *AnyPoint* possède un nom d'instance et deux attributs décrivant le mode d'interaction du point d'accès (*synchrone, asynchrone*) et le temps de validité de l'information véhiculée par le point d'accès (*TimeValidity)*.

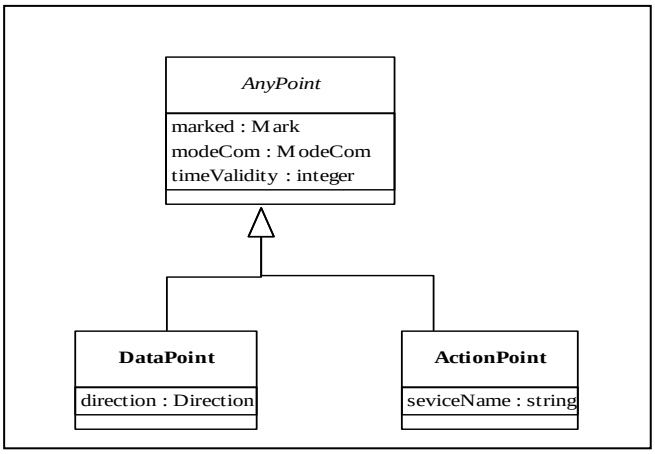



Figure 2.9: Diagramme de classes du point d'accès

Un point d'accès peut être marqué ou non marqué (*marked, unmarked*). Un point d'accès marqué est un point d'accès correctement connecté. Cette caractéristique est très utile dans le processus de validation d'une architecture et est exploitée dans un processus de conception du général vers le particulier avec validation progressive de l'architecture durant les différentes phases de conception, sans nécessité que les composants utilisés soient effectivement réalisés.

Le point d'accès est destiné à supporter deux concepts : le transfert de donnée et le transfert de flux. Le transfert de flux correspond souvent à l'invocation synchrone ou asynchrone d'un service. Chacun de ces deux concepts est supporté par un point d'accès spécifique (Figure 2.9) :

- ✓ Le point d'accès dédié aux transferts de données est représenté par le type *DataPoint*
- ✓ Le point d'accès dédié au transfert de flux de contrôle (invocation d'une action, reprise du flux de contrôle) est représenté par le type *ActionPoint.* Un point d'accès de type *ActionPoint* indique la présence d'un service qui peut être initié à partir de ce point

### 2.3.2 Les points d'Accès aux données : LE TYPE *DataPoint*

Un *DataPoint* est utilisé pour spécifier un transfert explicite de données. L'architecte peut utiliser un ensemble de *DataPoint* prédéfini (Figure 2.10) ou définir un *DataPoint* qui lui est spécifique. Les *DataPoint* prédéfinis englobent les types de données primitifs que nous retrouvons dans les divers langages de

programmation (entier, réel, caractère, booléen) ainsi que des types spécifiques à notre approche, tels que les types de données renseignant sur l'état structurel d'un composant composite.

A titre d'exemple, *IntDataPoint*, *ByteDataPoint*, *CharDataPoint* et *BooleanDataPoint* sont successivement associés aux types primitifs *int, byte, char* et *boolean*. Les points d'accès *SubCmpSetDataPoint, ConSetDataPoint, DConSetDataPoint, OpPartPortSetDataPoint,* et *CtrlPartPortSetDataPoint* sont associés à des types de données renseignant sur l'état structurel d'un composite (i.e. listes de composants, de connecteurs, de ports internes etc..).

La définition de nouveau *DataPoint* spécifique doit suivre un style de nommage et une méthodologie de définition bien précise. Le style de nommage apparaît clairement dans les exemples précédents. Le nom du nouveau type commence par le nom du type associé au point d'accès et se termine par le nom *DataPoint*.

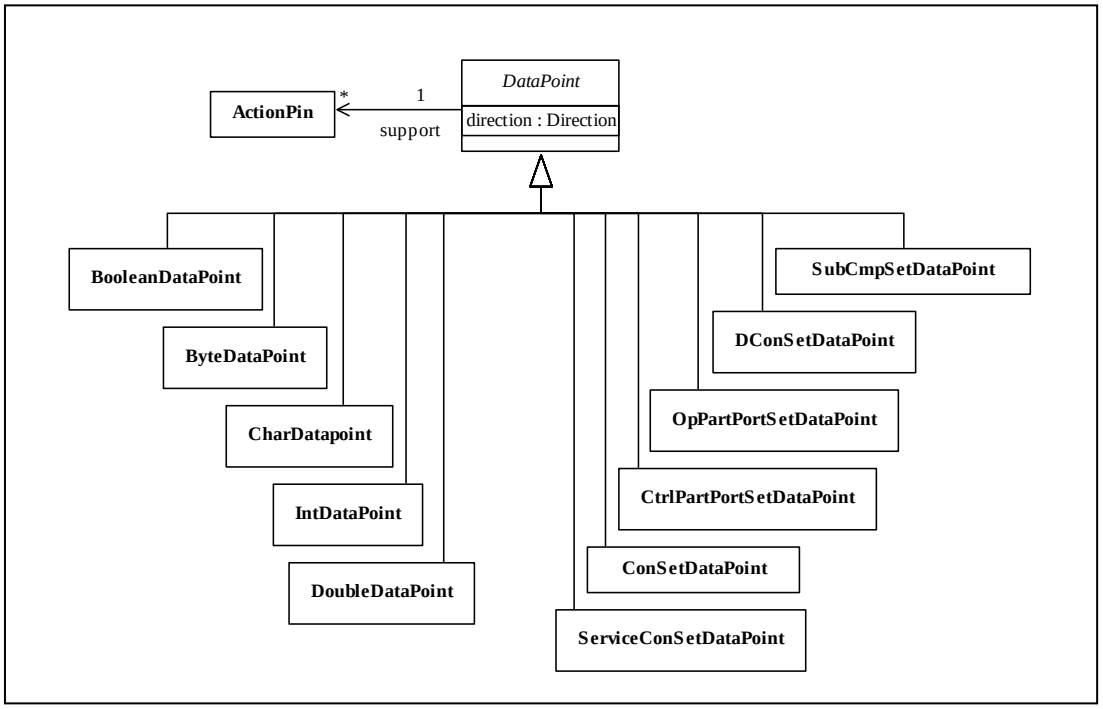



Figure 2.10 : Point de données prédéfinis

La méthodologie de construction d'un nouveau point d'accès de données consiste principalement à définir dans le port la zone qui contiendra la donnée et pourvoir le point d'accès de fonctionnalités usuelles tels que les accesseurs (méthodes *get* et *set*), la copie de données de point d'accès vers un point d'accès (méthode *copy*) et l'instanciation d'un point d'accès à partir d'un autre point d'accès ou du type de donnée auquel il correspond.

La transmission effective d'une donnée se fait dans le contexte du déclenchement d'une action. Celle ci pourrait être une action pure de transfert de données (envoyer, recevoir) ou à une action engendrant la réalisation d'un service particulier. Un *DataPoint* peut être associé à au une ou plusieurs entrées ou sorties d'une action.

Un point d'accès de type *DataPoint* est doté d'attributs indiquant le sens des opérations de transfert de données (Figure 2.10). Trois valeurs sont possibles pour spécifier le sens des données : *in, out* et *inout*. L'attribut *in* indique la nécessité de pourvoir le point d'accès d'une donnée. L'attribut *out* indique que le point d'accès est une source de données. L'attribut *inout* spécifie que la donnée peut être transférée dans les deux directions. L'accès à un point d'accès *inout* est par défaut synchronisé. Comme ceci est le cas dans plusieurs approches, les variables globales (mémoire partagée) sont considérées comme des implémentations possibles du concept de *DataPoint* possédant l'attribut *inout*. Dans notre cas, un point d'accès *inout* spécifie que le concept associé possède une existence réelle et n'est pas réduit à une référence. Ainsi si deux point d'accès *inout* de deux composant sont connectés, il y'aura alors deux copies de la donnée associée aux point d'accès, chacune localisée dans son composant. Si deux points *inout* sont connectés, les deux données associées aux points d'accès auront toujours les mêmes valeurs.

L'attribut de sens permet d'indiquer avec précision le type d'*ActionPin* auquel le point d'accès est associé. Ainsi un *DataPoint* ayant le sens *in* et un *DataPoint* ayant le sens *out* correspondent successivement à des *InputActionPin* et à des *OutputActionPin*. Un *DataPoint inout* correspond à un *InoutActionPin*.

Les types de données, associés aux points d'accès, peuvent être très complexes. Dans nos objectifs, lors des opérations de gestion dynamique d'une architecture, un composant entier pourrait être transmis et reçus à travers un point d'accès donné supportant de manière explicite le type de composant**.**

La connexion de point d'accès suit les règles suivantes : Un point *in* (respectivement *out*) ne peut se connecter qu'à un point *out* (respectivement *in*) ou *inout*. Un point *inout* est connectable à un point *in*, *out* et *inout*. Lorsqu'un point d'accès est correctement connecté, il est alors marqué (positionnement de l'attribut de marquage du point d'accès à la valeur *marked*).

Les points d'accès peuvent être explicitement spécifiés avec des valeurs d'initialisation (Figure 2.11). Un point d'accès initialisé, dont le sens est *in*, est un point marqué qui peut être non connecté. Les points d'accès *in* avec initialiseurs sont utilisés pour spécifier des valeurs par défaut pour des points d'accès qui, éventuellement, ne serait pas connectés. Si le point d'accès *in* avec initialiseur est connecté la valeur de l'initialiseur n'aura aucun sens. Un point d'accès avec initialiseur, et dont le sens est *out* est un point représentant une valeur constante qui ne peut pas être altérée durant l'exécution.

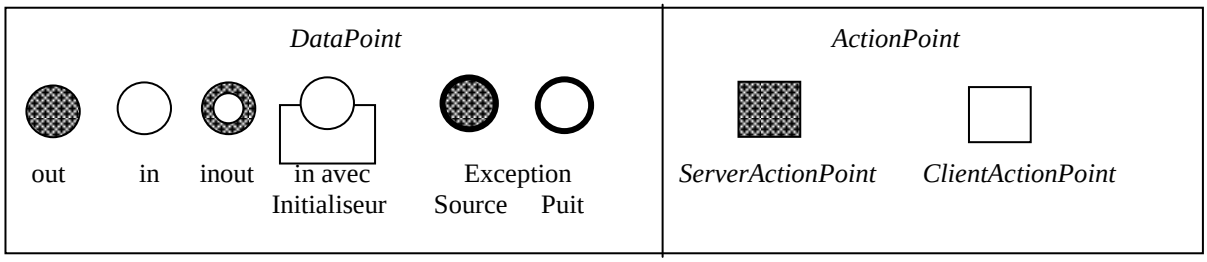



Figure 2.11: Représentation graphique des points d'accès

2.3.3 Les points d'accès de services : LE TYPE *ActionPoint :*

Un point d'accès de type *ActionPoint* indique qu'un service peut être initié à partir de ce point. Dans l'état actuel de la définition du modèle, nous considérons, comme ceci est le cas dans l'ADL *RAPIDE*, qu'un *ActionPoint* correspond uniquement à un seul service et un service permet de réaliser plusieurs actions.

Généralement le nombre d'actions que peut prendre en charge un service, est assez restreint et concerne un aspect bien précis d'un domaine d'application. C'est cette idée qui est à la base de la définition de la notion de *contexte d'action* au niveau du langage *SEAL*. C'est ainsi qu'un *ActionPoint* est associé à un ensemble d'actions SEAL (*ActionSet*) (Figure 2.12).

L’ensemble d’actions d’un *ActionSet*, associé à un *ActionPoint*, contient les identificateurs d’actions appartenant à l’espace de nom que défini le contexte d’actions associé. Chaque action d’un *ActionSet* est associée à un attribut spécifiant si l’action est complètement définie ou non (marquée ou non marquée).

Un point d’accès *ActionPoint* est dit marqué si toutes les actions qui lui sont associées sont marquées (définies). Le marquage de point d’accès joue un rôle fondamental dans le processus de validation d’une architecture en statique ou en exécution (dynamique)

Vis-à-vis d’un service, un point d’accès ne peut être que fournisseur ou client. Un point d’accès à travers lequel un service est fourni est représenté par le type spécifique *ServerActionPoint* (Figure 2.13). Lorsqu'un point d’accès spécifie un besoin de service, il est alors représenté par le type spécifique *ClientActionPoint* (Figure 2.13).

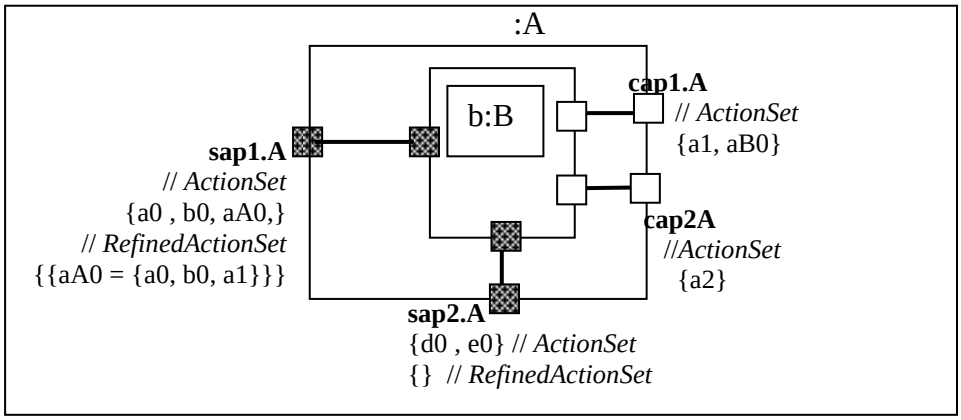



Figure 2.12: Les ensembles d'actions au niveau des points d'accès client et de services

Un *ServerActionPoint* dispose d’un deuxième ensemble d’action (*RefinedActionSet*), pouvant être vide. Cet ensemble spécifie pour chaque action abstraite contenue dans son *ActionSet* les sous actions nécessaires à sa réalisation (Figure 2.12). Lorsqu'un *RefinedActionSet* est défini, et que les actions abstraites citées dans l'*ActionSet* du point d'accès sont aussi citées dans le *RefinedActionSet*, alors le raffinement des actions abstraites au niveau du point d'accès n’est plus possible.

2.3.4 Les points d'acces specifiques :

Ce sont des points d'accès orientés vers le support de la spécification explicite des divers aspects usuels, non métier, que nous trouvons dans tout logiciel. Dans notre cas nous nous concentrons sur les aspects contrôle, états, exceptions et log.

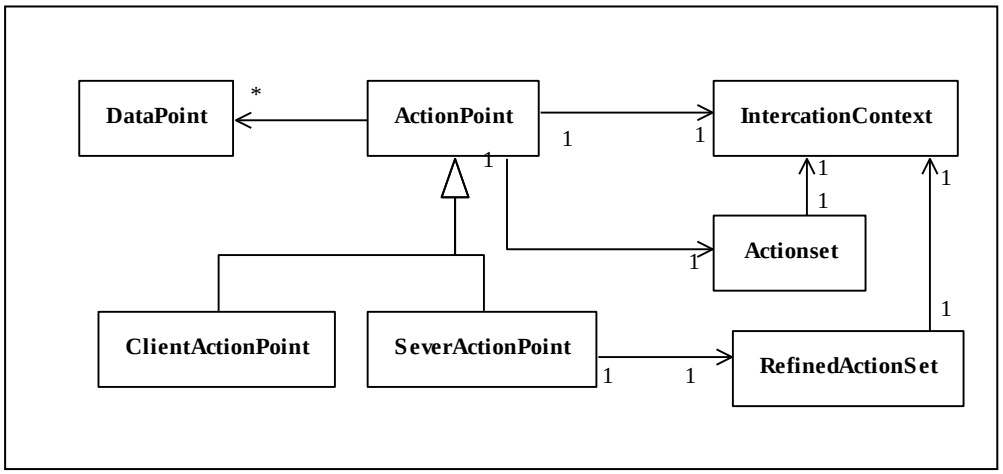



Figure 2.13: Diagramme de classe du point d'accès *ActionPoint*

**2.3.4.1** Les Points D'acces dedie au contrôle**:**

L'objectif du contrôle est de permettre la spécification d'opérations permettant :

- ✓ La spécification de la disponibilité ou non d'une ressource.
- ✓ La spécification du démarrage, arrêt, pause, reprise et redémarrage de service.

Le champ d'action de ces opérations peut concerner tout un composant, un port ou un point d'accès particulier. Pour la prise en charge des ces opération nous avons défini deux point d'accès particuliers : *ControlledServerActionPoint* et *EnableDataPoint*.

Le *ControlledServerActionPoint* est un *ServerActionPoint* spécifique (Figure 2.14). La spécificité est déterminée par le fait que ce type de point d'accès est doté, en plus des actions ordinaires supportées par le service fourni, d'un troisième ensemble d'actions (*ControlActionSet)*. L'objectif de ce dernier est de spécifier les actions qui peuvent agir sur l'état du service. Les actions de contrôle de ce troisième ensemble d'actions sont déterminées à partir du contexte d'actions

de contrôle du langage SEAL. Ce contexte définit les actions primitives suivantes : *Enable*, *Disable*, *Start*, *Stop, Pause, Resume et Restart*. Les opérations de contrôle concernent le service dans sa globalité et ne permettent pas d'aller plus en détail pour le contrôle individuel d'actions d'un service.

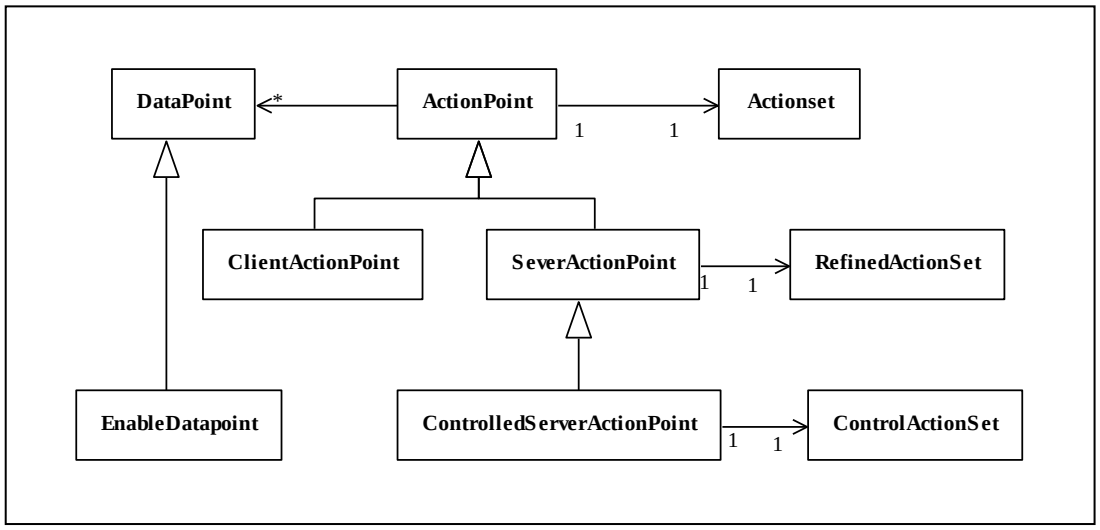



Figure 2.14: Diagramme de classes des points d'accès dédié au contrôle

Le point d'accès *EnablelDataPoint* est un *DataPoint* doté de la direction *in* pour le transfert des données. Il agit sur un port tout entier en permettant de contrôler la mise à disponibilité ou non de toutes les ressources du port. La figure 2.15 montre la représentation graphique d'un point de service contrôlé (figure 2.15a) et du point de contrôle *EnableDataPoint* (figure 2.15b).

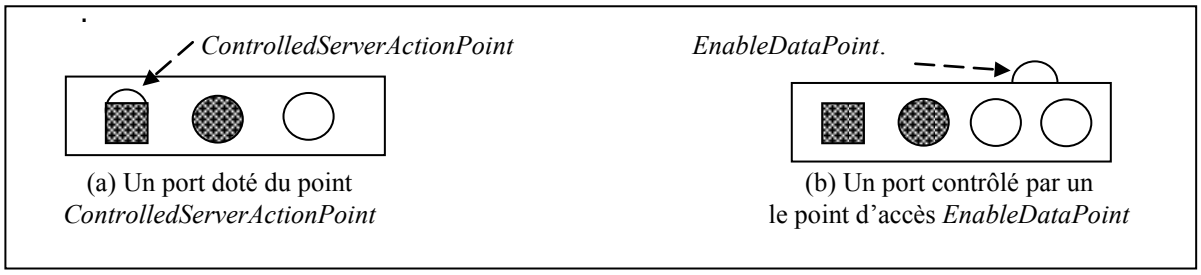



Figure 2.15: Représentations graphique des points d'accès de contrôle

Le positionnement du point *EnableDataPoint*(figure 2.15b) à l'extérieur du port indique que le point de contrôle représente une sorte de porte d'accès aux ressources du port. Les valeurs acceptées sur le point de contrôle sont les deux valeurs énumérées *ENABLE* et *DISABLE.*

2.3.4.2 Les points d'accès d'états

Les points d'accès d'états permettent d'exporter vers le monde externe les divers états d'un composant, de ses ports et des services sur les ports. Ils permettent aussi de définir une topologie bien précise exploitant ces états.

Pour le report des états des services présents sur les ports, nous avons identifié deux points d'états qui doivent toujours être associés à un point de service : Le *StateDataPoint* et le *StateClientActionPoint.*

Le *StateDataPoint* possède une direction *out* du flux de donnée et reporte les états du service auquel il est associé. Les états définis sont représentés par les valeurs énumérées *STARTED, STOPPED, PAUSED, RESUMED.*

Le point d'accès *StateClientActionPoint* est un *ClientActionPoint* pourvu d'une table de correspondances (*StateActionMap*) dans laquelle nous retrouvons pour chaque état une action de l'ensemble des actions attachées aux points d'accès.

Pour les composants deux états sont reportés : l'état global et l'état structurel du composant. L'état global est reporté à travers le point d'accès du type *GlobalStateAccessPoint*.

Actuellement, les états globaux reportés concernent le concept de stabilité d'un composant. Un composant est dit stable si l'application d'une enveloppe marquée à ce composant, provoque le marquage de tous les points d'accès de ce composant à tous les niveaux de sa hiérarchie de composition.

L'état structurel est reporté à travers des points d'accès prédéfinis, chacun décrivant une vue précise de cette structure (Figure 2.10). A titre d'exemple, les instances de composant (*SubCmpSetDataPoint)* et les connecteurs entre ces composants (*ConnectorSetDataPoint)* sont des états structurels.

2.3.4.3 Les points d'accès exception

La gestion des erreurs est un des aspects que nous isolons des fonctions métiers d'un composant. Nous utilisons la technique des exceptions pour la signalisation et la prise en charge du traitement des erreurs. Une exception est un événement qui apparaît durant l'exécution d'une architecture et qui déroute le flux normal d'exécution.

Les points d'accès d'exception sont représentés par le type *ExceptionDataPoint.* Les points d'accès d'exception sont les points à travers

lesquels l'architecte spécifie soit qu'un composant est une source potentielle d'erreur, soit qu'un composant est l'endroit ou doit être gérée l'erreur. Lorsqu'un composant est doté d'un point d'accès exception dont la direction est *out*, il est considéré comme une source potentielle d'erreur. Il doit impérativement être mis dans une zone gardée. La mise dans une zone gardée se traduit par la nécessité de connecter le point d'accès exception. Si par contre le point d'exception est doté d'une direction *in,* le composant auquel il est associé est un composant dédié à la gestion d'exception.

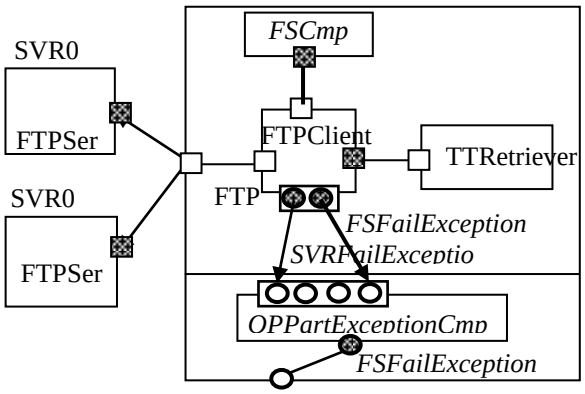



Figure 2.16: Exemple de mise en œuvre des points d'accès d'exception

Les points d'accès d'exceptions représentent les places où l'architecte doit expliciter les réponses à l'apparition d'exceptions. C'est la topologie de connexion de ces points d'accès qui indiquera la méthode préférée pour le traitement de l'erreur. Dans le contexte de l'approche intégrée, tous les points d'exception doivent être reliés à un composant spécifique de la vue interne d'un composite (*OpPartExceptionCmp*), chargé du traitement des exceptions

La logique de prise en charge des exceptions est supportée par ce composant spécifique. Elle correspond, en général, à deux grandes stratégies:

- ✓ Relayer l'information décrivant une exception vers le niveau supérieur de la hiérarchie de composition. Dans ce cas, le point d'accès exception réapparaitra comme source d'exception au niveau du composant *OpPartExceptionCmp* et sera alors lié par un connecteur de délégation à un point d'accès exception du composant englobant (Figure 2.16).

- ✓ Traiter l'exception. La logique de traitement de l'exception est définie dans le composant *OpPartExceptionCmp*.

Comme dans le cas de certains langages de programmation, nous avons prédéfini un ensemble de type d'exception. Les exceptions spécifiques doivent être définies comme sous type du type *ExceptionDataPoint.* A titre d'exemple, l'exception prédéfinie *UndefinedServiceException* est provoquée dans le cas où un client tente d'accéder à un service qui n'est pas en état de disponibilité, suite à sa suppression. L'exception *UnavailableServiceException* apparaît lorsque le service est dans l'état d'indisponibilité. Ces deux exceptions sont transmises à travers les points d'accès *UndefinedServiceExceptionDataPoint* et *UnavailableServiceExceptionDataPoint*.

2.4 Le port :

Un port est un regroupement de points d'accès, étroitement associés dans le contexte de la réalisation d'un objectif commun. Tous les ports sont représentés par le type *AnyPort* (Figure 2.17). Un port possède un attribut renseignant sur le nom de l'instance. Le port représente un espace de nom pour les points d'accès. Chaque point d'accès est identifié de manière unique dans le contexte d'un port. Le composant représente un espace de nom pour un port. Ce dernier est identifié de manière unique dans un composant. Les ports modélisent la vue externe d'un composant. C'est seulement à travers les ports qu'un composant est manipulable. Les ports divulguent les ressources (services et données) requises et fournies d'un composant ainsi que les aspects comportementaux du composant, observables sur ces ports. Les comportements sont décrits dans le langage d'action SEAL qui est une des composantes essentielles de notre approche. Un port est une entité à part, pouvant être ajouté ou supprimé à un composant. Il peut en outre être modifié par l'ajout ou la suppression de point d'accès, notamment les points d'accès de données.

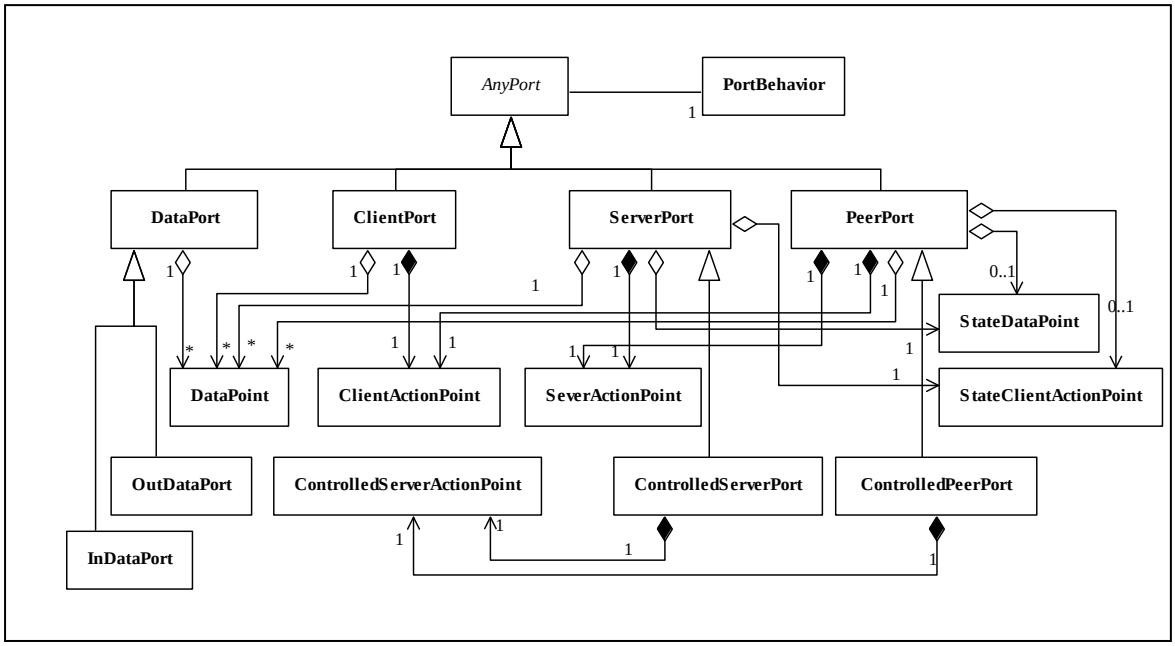



Figure 2.17: Diagramme de classe des ports

2.4.1 Les catégories des ports :

Le port fait une abstraction totale des mécanismes logiciels qui sont utilisés lors de l'établissement d'une connexion. Cependant il est souvent d'usage que tôt dans le cycle de conception, l'architecte choisit de manière explicite, une technologie d'interconnexion (i.e. bus CORBA, protocole FTP, Web services etc..), ou de manière implicite, lors de la mise en œuvre d'un style architectural bien connu (i.e. architecture à plusieurs niveaux centrée sur les EJB). Pour supporter ce type de spécification, un certain nombre de ports ont été prédéfinis.

La structure externe d'un composant fait ressortir des aspects qu'il est nécessaire d'isoler de l'aspect métier des composants. Ces aspects concernent la gestion des erreurs par exception, l'état des composants, les *logs* et les contrôles globaux sur les composants. Pour une prise en charge explicite de ces aspects, des ports dédiés à de tels aspects ont été prédéfinis.

Les ports que nous utilisons pour atteindre les objectifs que nous venons de citer sont organisés en quatre catégories :

- Les ports ordinaires.
- Les ports contrôlés.
- Les ports orientés aspects non métiers ou spécifiques.
- Les ports standards.

2.4.1.1 Les ports ordinaires :

Nous avons défini quatres types de ports ordinaires, qui répondent à une grande variété d'applications : *ClientPort, ServerPort, PeerPort* et le *DataPort* (Figure 2.17).

Un port de données (*DataPort*) peut contenir plusieurs points d'accès de données. Un port *InDataPort* (respectivement *OutDataPort*) ne contient que des points d'accès dont le sens est *in* (respectivement *out*).

Un port client (*ClientPort)* contient un et un seul point d'accès *ClientActionPoint* et optionnellement un ou plusieurs point d'accès de données (*DataPoint*). Cette structure implique qu'un port est associé avec un seul service. En pratique, cette technique parait naturelle, surtout dans le cas de services Internet, ou chaque service est associé à un numéro de port bien précis.

Un port de service (*ServerPort)*, comme conséquence du choix précédent, ne peut contenir qu'un point d'accès de service (*ServerActionPoint)*. Les autres points d'accès optionnels sont des points d'accès de données et éventuellement des points d'accès d'états (*StateDataPoint*, *StateClientActionPoint*).

Un port de type *PeerPort* doit contenir un point d'accès *ServerActionPoint*, un point d'accès *ClientActionPoint* et éventuellement des points de données et des points d'états. Ce type de port a été introduit pour permettre le support de communication d'égal à égal.

2.4.1.2 Les ports contrôlés:

Un port contrôlé supporte la spécification des diverses opérations de contrôle définies dans le contexte des actions de contrôle du langage SEAL (*Enable*, *Disable*, *Start, Stop, Pause, Resume, et Restart*). L'accès global au port (*ENABLE/DISABLE*) est assuré par la présence du point *EnableDataPoint* (Figure 3-18). Les autres contrôles sont rendus possibles par la présence dans le port d'un point de service contrôlé (*ControlledServerctionPoint*).

Trois type des ports contrôlés sont pour l'instant identifiés (Figure 3.18): les ports de données contrôlés (*ControlledDataPort*), les ports de service contrôlés (*ControlledServerPort*) et les ports d'égal à égal contrôlés (*ControlledPeerPort*).

Un *controlledServerPort* et *un ControlledPeerPort* doivent obligatoirement contenir un *ControlledServerActionPoint*.

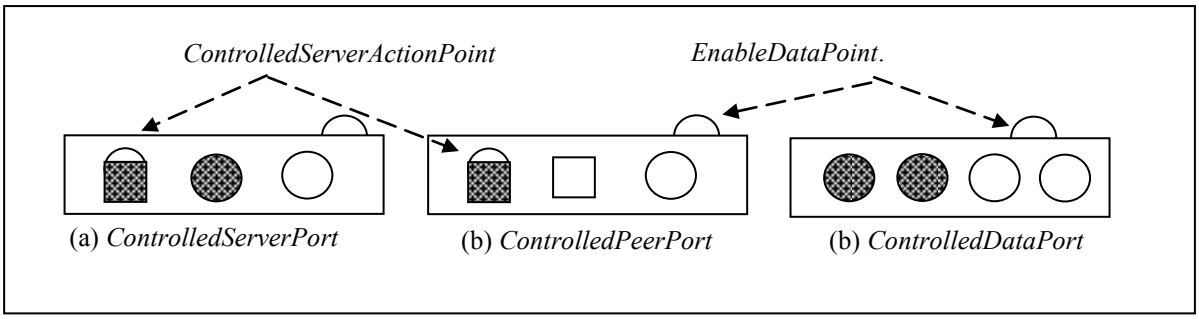



Figure 2.18 Représentation graphique des ports contrôlés

2.4.1.3 Les ports spécifiques, orientes aspects non métier:

Ces ports sont destinés à supporter les aspects tels que les exceptions, les états, les logs et le contrôle global sur le composant. Dans l'état actuel des travaux nous avons identifiés les 4 ports suivants:

- ✓ Le port principal du composant.
- ✓ Le port d'état du composant.
- ✓ Le port de gestion des états exceptionnels.
- ✓ Le port destiné aux logs.

Le port d'états exceptionnels, représenté par le type *ExceptionDataPort* (Figure 3-19) regroupe uniquement les point d'accès exceptions. Ces derniers correspondent soit à des exceptions relayées au composites par ses composantes internes, soit à des exceptions qui apparaissent au niveau de la vue interne d'un composite.

Les exceptions qui apparaissent au niveau de la vue interne d'un composite concernent les opérations d'évolution dynamique du composite et le déroulement des interactions entre les composants du composite. Elles sont provoquées soit par les connecteurs dans le cas de la violation de la spécification d'une interaction soit par le composant *OpPartController* qui est un composant spécifique du modèle de la vue interne des composants composites.

Le port d'état reporte l'état global d'un composant et ses états structurels. Un port d'état, représenté par le type *StatePort*, ne peut être composé que des points d'accès d'états.

Les points d'accès rentrant dans la constitution du *StatePort* (Figure 2.19) sont:

- ✓ Le *GlobalStateAccessPoint* qui est le point d'accès qui renseigne sur l'état global du composant.
- ✓ Les points d'accès supportant les types de données renseignant sur les états structurels du composite (i.e. *SubCmpSetDataPoint* représente les composants formant le composite, *ConSetDataPoint* pour les connecteurs, *DConSetDataPoint* pour les connecteurs de délégation *etc.*).

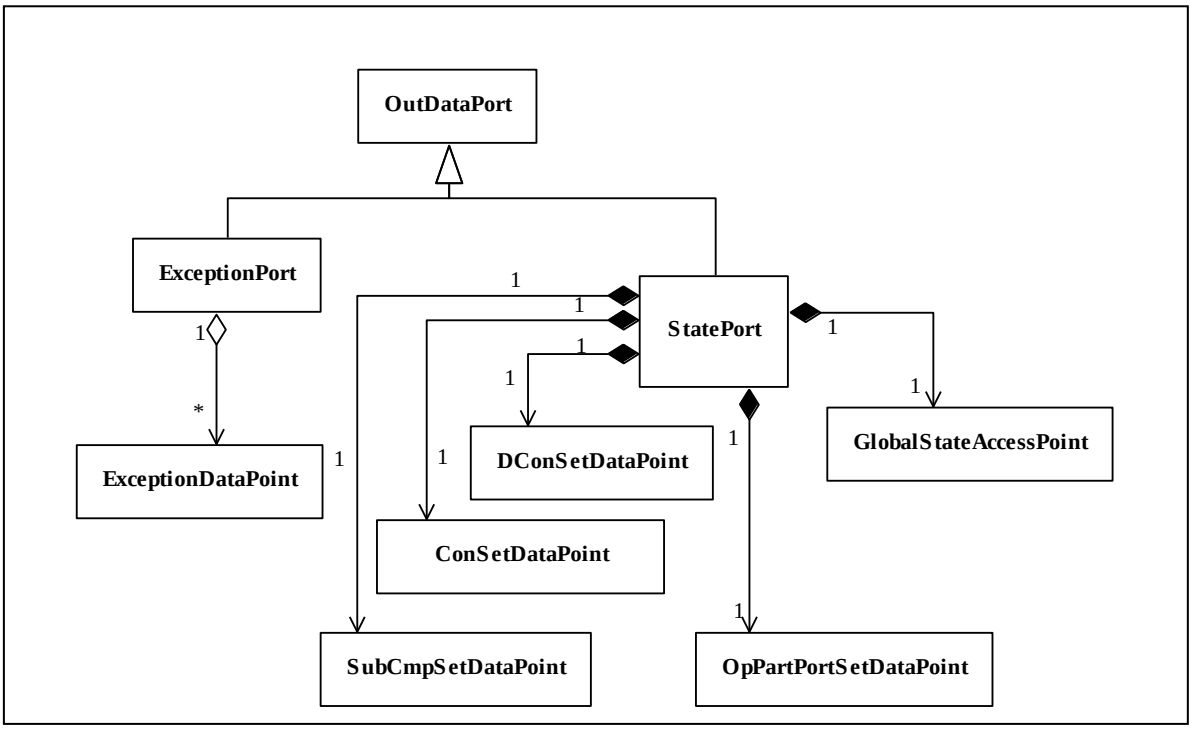



Figure 1.19: Diagramme de classes des ports d'exception et d'états

Chaque composant composite est obligatoirement doté d'un port principal (*MainCmpPort).* Ce port est un *ControlledServerPort* qui permet la spécification de contrôles affectant le composant dans sa globalité. Ceci est rendu possible par le fait que ce port est directement lié au composant *OpPartControllerCmp* qui est un composant essentiel de la vue interne des composants composite. L'impact des opérations de contrôle sur les instances de composants formant le composite dépend de la topologie de contrôle global définie dans le composite à un moment donné. Ces opérations ne concernent que les composants du composite

appartenant à cette topologie de contrôle global (une sorte de réseau de composants contrôlé globalement avec leur composite).

Pour une clarté de la spécification d'une architecture, un port spécifique, le *LogDataPort* a été défini. Le *LogDataPort* doit être lié à un composant spécifique de la partie contrôle chargé de mettre en forme les logs. Le composant spécifique pour les logs représente l'endroit unique où sont homogénéisés les logs (format de sortie des logs) et à partir duquel seront transmis les logs vers leurs destinations finales (i.e. fichiers, applications).

2.4.1.4 Les ports predefinis:

Ce sont des ports orientés vers le support de connecteurs très répandus tels qu'une interaction dans le contexte d'un protocole standard de communication ou une interaction avec un environnement d'exploitation (système d'exploitation, serveur d'application). La grande différence entre ces ports et les autres ports résident dans les faits suivants :

- La vue concrète du port s'attache directement à une extrémité du connecteur standard correspondant et ne nécessite aucun service supplémentaire d'adaptation.
- Les ports prédéfinis peuvent contraindre le déploiement d'un composant à un nombre restreint de cas de déploiement. Ainsi si le composant dispose d'un port *UnixEnvPort*, le composant ne peut pas être déployé comme composant *EJB*.

Certains ports standards relient l'application en cours de conception à des applications autonomes. Ces applications sont complètement définies, et sont souvent standardisées. Ainsi une connexion FTP met en œuvre un serveur FTP. Le port coté serveur est complètement défini dans le serveur et ne fait pas partie de l'application. De même pour un connecteur vers l'environnement UNIX ou vers un serveur HTTP. Les ports standards qui s'attachent aux applications externes, possèdent des ports concrets vides (pas de vue concrète). Les vues concrètes de ces ports sont définies dans leurs applications. Ces ports sont cités dans une architecture uniquement pour la rendre claire et explicite.

Actuellement nous avons défini un nombre restreint de ports standards supportés par les types de port suivant : *FTPServerPort, FTPClientPort, HTTServerPort, HTTPclientPort, HTTPServerPort, CORBAClientPort,*

*CORBAServerPort*, *EJBClientPort, EJBServerPort, UnixServerPort,* Les ports *FTPServerPort, HTTServerPort, UnixServerPort,* ne possèdent pas de vue concrète étant donné qu'ils correspondent à des applications standards.

2.5 Conclusion :

IASA est un modèle de composant spécifique pour la spécification d'une architecture logicielle. Ainsi un architecte travaille directement à l'aide des concepts qu'il retrouve au niveau de son métier. Cependant, il reste difficile pour un architecte de gérer ce que nous appelons les évolutions externes qui correspondent à l'intégration de nouvelles préoccupations au sein d'une architecture logicielle.

La séparation en composants et la hiérarchisation offrent un premier niveau pour modulariser une architecture logicielle et offrir un modèle abordable selon une granularité variable pour l'architecte. Il est alors possible, grâce à la hiérarchie, du modèle d'imaginer une construction incrémentale partant de la spécification des composites de granularité la plus importante vers la spécification des composants primitifs. Un tel découpage ne peut à lui seul être à la base d'un bon processus de construction incrémentale d'une architecture. En effet, certaines préoccupations ne peuvent être correctement modularisées à l'aide d'une telle approche et se retrouvent alors noyées au sein d'une description d'architecture logicielle. Ces préoccupations sont alors très difficiles à intégrer au cours d'une construction incrémentale de l'architecture.

C'est pourquoi nous proposons dans le chapitre suivant ***IASA-AOP*** extension du modèle de composant IASA pour l'intégration de nouvelles préoccupations au sein d'une architecture logicielle.

# Chapitre 3 : Extension du modèle de composant IASA pour l'intégration des aspects Le langage 3ADL

## 3.1 INTRODUCTION:

La conception de logiciels est une étape importante dans le cycle de vie de développement logiciel. La conception Orienté objet (OO) a montré sa force quand il s'agit de la modélisation du comportement commun, Mais la conception OO ne répond pas adéquatement on se qui concerne les comportements qui s'étendent sur de nombreuses classes

Les préoccupations transversales ou se qu'on appelle *crosscutting concerns* en anglais telles que la sécurité et persistance y compris tous les problèmes bien connus qu'elles entraînent, sont présents tout au long du cycle de développement entier et donc entraîner une réduction des avantages attendus de la conception. La programmation orienté Aspect (AOP) aborde ces problèmes au niveau de codage et offre un support de bas niveau pour la séparation des préoccupations comme on peut le trouver, par exemple dans AspectJ , Hyper/J ou de Céasar .

Un manque de soutien de conception conduit à un écart entre la conception et la mise en œuvre qui aggrave les résultats souhaités. Pour profiter des avantages AOP à des stades plus tôt dans le cycle de développement logiciel, les capacités de séparation similaires doivent être fournies également au niveau de la conception donc on

Ce chapitre traite de la spécification des préoccupations transversales au niveau de la conception et de maintenir la séparation des préoccupations plus tôt dans le cycle de vie.
Notre travail peut être considéré comme une étape importante sur la façon de définir les aspects à la phase de conception du développement logiciel orienté aspect (AOSD).

Notre travail porte sur le processus de développement orienté aspect de la conception à la génération de code. En raison du fait que le soutien aspect a été porté principalement au niveau de l'implémentation mise en œuvre, nous nous concentrons sur la conception et de présenter une cartographie automatisée de modèles de conception à modèles de programmation.

Le modèle de composants IASA a été défini au laboratoire LRDSI. Il se présente sous la forme d'une spécification et d'implémentations dans différents langages de programmation comme Java, ArchJava. IASA présentée dans le chapitre précédent est un modèle de composant spécifique pour la spécification d'une architecture logicielle. Ainsi un architecte travaille directement à l'aide des concepts qu'il retrouve au niveau de son métier.

Cependant, il reste difficile pour un architecte de gérer ce que nous appelons les évolutions externes qui correspondent à l'intégration de nouvelles préoccupations au sein d'une architecture logicielle.

La séparation en composants et la hiérarchisation offrent un premier niveau pour modulariser une architecture logicielle et offrir un modèle abordable selon une granularité variable pour l'architecte. Il est alors possible, grâce à la hiérarchie, du modèle d'imaginer une construction incrémentale partant de la spécification des composites de granularité la plus importante vers la spécification des composants primitifs. Un tel découpage ne peut à lui seul être à la base d'un bon processus de construction incrémentale d'une architecture. En effet, certaines préoccupations ne peuvent être correctement modularisées à l'aide d'une telle approche et se retrouvent alors noyées au sein d'une description d'architecture logicielle. Ces préoccupations sont alors très difficiles à intégrer au cours d'une construction incrémentale de l'architecture.

C'est pourquoi nous proposons dans ce chapitre 3ADL (Aspect, Action and Architecture Description Language) L'ADL de l'approche IASA extension du modèle de composant IASA pour l'intégration de nouvelles préoccupations au sein d'une architecture logicielle étape par étape: De l'architecture qui contient la logique métier que dans une architecture globale qui contient des préoccupations transversales et techniques en séparant les différents besoins fonctionnels et préoccupations extra-fonctionnelles (telles que sécurité, fiabilité, efficacité, performance, ponctualité, flexibilité, etc.). Cette approche vient de principes

Aspect Oriented Software Development (AOSD) où les concepteurs définie séparément toutes les facettes d'une application (métier et technique), puis les tisser. En effet, dans l'évolution de démarche, l'architecte logiciel doit intégrer de nouveaux composants pour fournir des fonctionnalités nouvelles.

Ce chapitre sera consacré aux concepts, techniques et mécanismes que nous avons définis pour rendre IASA une approche native d'Architecture Logicielle Orientée Aspect. C'est au niveau de ce chapitre que nous montrerons comment les concepts de port a été exploité pour faire supporter à IASA le concept d'Aspect. Dans ce même chapitre nous avons présenté une nouvelle vue sur l'organisation interne des composants IASA et nous avons proposé l'introduction d'une nouvelle clause dans le langage SEAL. Le Langage SEAL aura après introduction des aspects un nouveau nom : C'est 3ADL pour Architecture, Action an Aspect Description Language.

Nous avons présenté dans le chapitre précédent l'importance du développement logiciel orienté aspect qui s'étale sur toutes les étapes du cycle de vie d'un logiciel, notamment le niveau architectural. Nous avons présenté ensuite un état de l'art sur **les travaux concernant l'intégration des aspects dans les différents modèles de composants**.

Nous nous intéressons dans ce travail de définir une extension du modèle de composant IASA pour l'intégration de nouvelles préoccupations au sein d'une architecture étape par étape, on se basant sur le langage 3ADL un langage de description d'architecture orienté aspect basé sur le **modèle de composant IASA.**

Dans la suite nous présentons en détail les éléments fondamentaux du modèle de composant 3ADL. Nous commençons par Les éléments modélisant la vue externe les concepts de point d'accès et le concept de port, ensuite nous présentons le concept d'enveloppe et le modèle d e la vue interne ainsi que tout les aspects liés au modèle de composant.

3.2 L' architecture logicielle orientée aspect avec 3ADL:

ON a introduit trois artefacts pour développer des applications à base de composants et d'aspect: *composant d'aspect*, *port d'aspect* et *liaison d'aspect* et ASPOAP. Ces trois artefacts n'introduisent pas de concepts supplémentaires dans le modèle IASA.

- Le composant d'aspect est un composant primitif qui implémente le comportement d'un aspect (i.e. le code advice).
- La liaison d'aspect est une liaison entre un composant aspectisable et le composant d'aspect.
- Un port Aspect c'est seulement à travers ce port qu'un advice de composant aspect est accessible et manipulable.

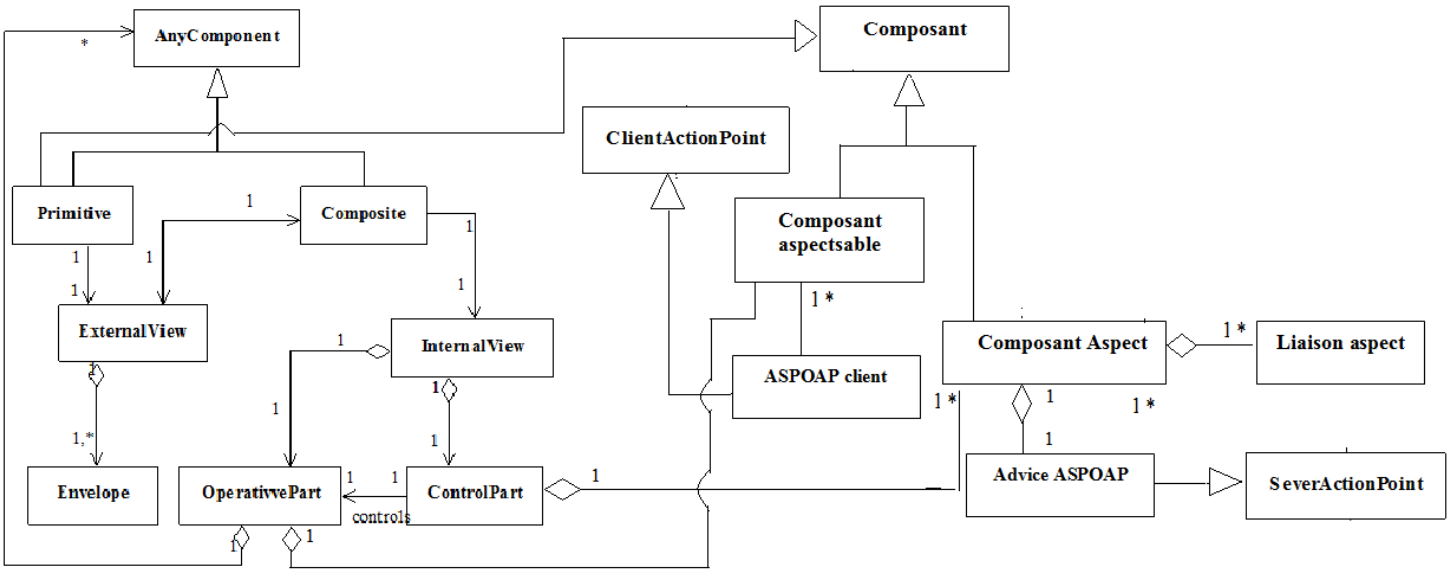



Figure 3.1. méta modèle d'IASA

3.2.1 Le composant aspect:

En utilisant le même concept de la programmation orientée aspect, un aspect est une unité modulaire utilisée pour gérer l'encapsulation des joinpoints, des pointcuts, et des advices que nous détaillerons dans les sections suivantes.

Un composant d'aspect est le seul endroit ou les advices sont spécifié. Un composant d'aspect est instancié uniquement dans la partie contrôle.

Un composant d'aspect dans IASA utilise le même modèle de composant IASA, Un composant Aspect possède un port spécifique dans la vue externe de composant aspect. Ce port est appelé un ***port Aspect.***

**Un composant aspect** est un composant **IASA** fournissant au moins une interface de conseil ou ***Advice_ASPOAP*** (équivalent à la notion d'advice dans L' AOP). Il définit le comportement d'un aspect. Une **interface de conseil(*Advice_ASPOAP)*** – ou interface de code advice – est une interface serveur définissant un code advice de type before, after, ou around, c'est-à-dire un code qui doit être tissé avant, après ou autour une opération de composant interceptée par l'interface de tissage. Elle fournit une réification d'une invocation d'opération de composant en délivrant un certain nombre d'informations liées au contexte d'interception, tel le nom de l'opération interceptée, son composant, ses paramètres et leurs types.

```
package nomDePackage;
import listeDePackage // package de composant et de connecteurs
component nomDuTypêDeComposant {
// Définition de types internes. Pour être instancié dans d'autres composants, un type doit être
//définis à l'extérieur du composant, soit dans un même fichier ou dans un fichier différent.
accesspoint { // Définition de types internes de points d'accès de données }
port { // Définition de types internes de port }
connector { // Définition de types internes de connecteurs. }
component { // Définition de types internes de composant.}
/// Définition des instances
ports{ // définition des ports du composant
}
operativepart{ // description de la partie opérative:
// instance de composant, type internes de composant, et connexions
}
controlpart{// description de la partie contrôle: instance de composant et connexion inter
// composant de la partie contrôle et avec les composants de la partie opérative
}
Aspectpart{// description de la partie aspect: instance de composant et connexion inter
// composant de la partie aspect et avec les composants de la partie opérative
}
behavior{ // Comportement du composant; Ce comportement représente
// celui du composant OpPartController de la partie control
}
properties{ // Spécification des propriétés non fonctionnelles. Pour l'instant, seules les
// sont décrites les informations de déploiements
}
      } // Fin description type de composant
```

Figure 3.2: Les clauses de la description textuelle d'un composant IASA

```
Public component class IASA_Component{
// Instantiation initial des Datapoint || Actionpoint
// Définition des ports
  Public port  NomPortIN{
Provides void NomMethode1(paramètres ){…..}
}
Public port  NomPortOUT{
Provides void NomMethode2(paramètres ){…..}
Requires void NomMethode3(parametres ){…..}
}
Public void NomMethode1(paramètres ){
// définition de la méthode
 }
Public void NomMethode2(paramètres ){
// définition de la méthode
 }}
```

Figure 3.3: L'implémentation d'un composant d'un composant IASA avec Archjava

```
        AspectPart{

Public component class IASAAspectComponent{
// Instanciation des Datapint
// Définition des ports Aspect
 // Instanciation  des Advice ASPOAP
    Public port  NomPortAspect{
            Provides void NomMethode(paramètres ){…..}
            Requires Methode2(parametres )
  }
}

Public port  AdvicePort{

      Provides void AdviceMéthode(paramètres ){

       // Définition de la méthode advice associé aux points de jonction
             }
}

        Pointcuts{

     // Déclaration de coupe


}
}
        }// fin aspectPart
```

Figure 3.4: L'implémentation d'un composant d'un composantAspect IASA avec Archjava

### 3.2.2 Les ports orientés aspects non métiers:

Un port orienté aspect (**aspect port)** est un port de type ***DataPort*** ou ***ServerPort.*** Un port orienté aspect de type serveur (aspect server port) est aussi appelé ***advice port*** c'est seulement à travers ce port qu'un advice de composant aspect est accessible et manipulable.

Un port orienté aspect contient un point d'accès orienté aspect (ASPOAP).

Les ports modélisent la vue externe d'un composant. Un port orienté aspect est toujours instancié comme une partie de la vue externe de composant orienté aspect.

Un advice port apparaît dans la partie gauche en haut dans la notation graphique de composant aspect

Les ports aspects sont spécifiques à une application particulière puisqu'ils servent à tisser l'aspect dans l'application en question.

Un port est un regroupement de points d'accès, étroitement associés dans le contexte de la réalisation d'un objectif commun. Un port possède un attribut renseignant sur le nom de l'instance. Le port représente un espace de nom pour les points d'accès. Chaque point d'accès est identifié de manière unique dans le contexte d'un port. Le composant représente un espace de nom pour un port. Ce dernier est identifié de manière unique dans un composant.

Les ports divulguent les ressources (services et données) requises et fournies d'un composant ainsi que les aspects comportementaux du composant, observables sur ces ports. Les comportements sont décrits dans le langage d'action SEAL qui est une des composantes essentielles de notre approche.

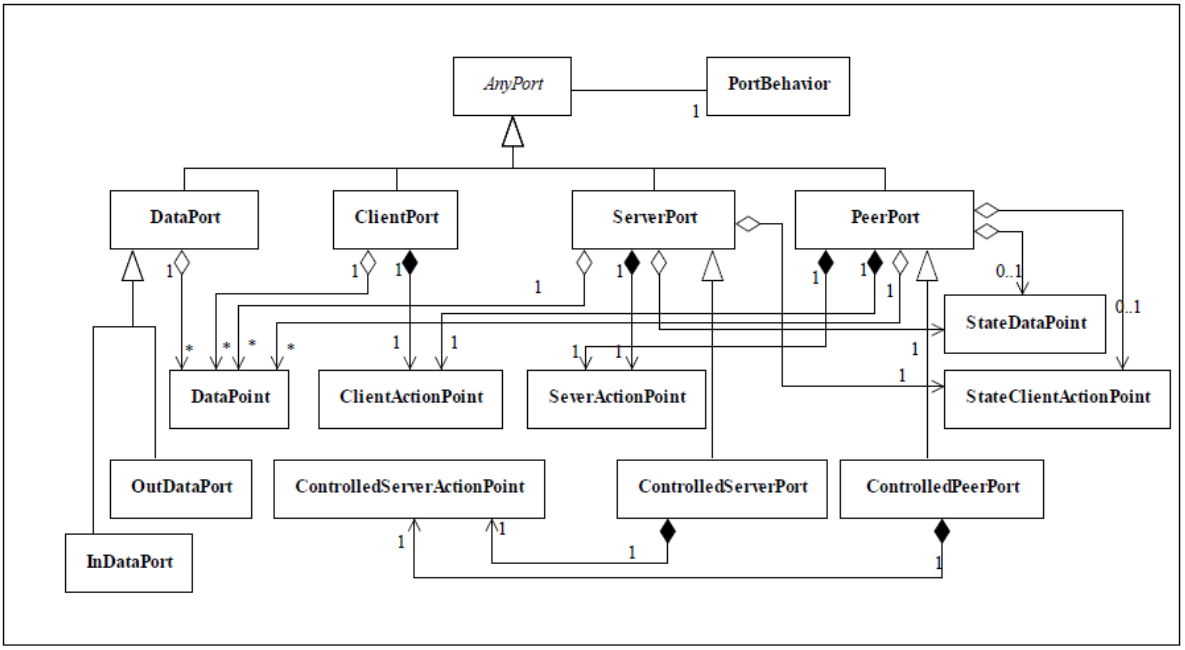



Figure 3.5 Diagramme de classe représente le port

### 3.2.3 Le point d'accès orienté aspect (ASPOAP[12]) :

Un ***ASPOAP*** est un point d'accès de type ActionPoint (ACTOAP) de type client ou serveur associé avec des actions orientées Aspect. Un ASPOAP serveur est aussi appelé ***advice ASPOAP.***

Les actions orienté aspects sont représentées sous forme d'un ensemble d'alias prédéfinies nommés des alias aspects (aspect aliases).

La notion d'***alias*** a été introduite pour rendre compatible des contextes d'actions attachés aux éléments de modélisation. Un alias est une autre forme de présentation d'une action. Dans un même contexte une action peut posséder plusieurs alias.

Chaque aspect alias est associé avec un type spécifique d'advice. Le nombre d'actions associés avec un ***advice ASPOAP*** ne dépendent pas de nombre des types d'advice supportés.

Les types d'advices de type before, after, ou around sont supportés à travers cinq aspects aliases: ***aroundFirstAction, aroundLastAction, proceedAction, beforeAction, afterAction.***

Apres la connexion d'un ***advice ASPOAP*** avec un ASPOAP client, *l'ASPOAP client* est fourni avec un alias aspect étendu on préfixant l'alias aspect

[12] ASPOAP: Aspect Oriented Actions

de base avec le nom de port qui contient l'***advice ASPOAP*** avec un nombre qui spécifie l'ordre des aspects exemple (pAuthAdvice_1_aroundFirstAction). Ce sont ensuite utilisées pour déterminer l'ordre de lancement des advices lorsque deux ou plusieurs aspects sont attachés au même point de jonction.

```
Interface advice_ASPOAP{
public void aroundFistAction (……){ }
publicvoid aroundLastAction (……){ }
public void ProceedAction (……){ }
void beforeAction()(……){ }
void AfterAction()(……){ }
}
```

Figure 3.6: L'implémentation d'un ASPOAP avec Archjava

### 3.2.4 Les points de jonctions/ joinpoints:

Les points de jonction sont des points dans le flot de contrôle d'un programme dans lequel un ou plusieurs aspects peuvent être appliqués. La notion de point de jonction n'est pas suffisante à elle seule pour définir quels points de jonction sont pertinents pour un aspect donné. On a besoin d'une autre notion pour décrire les points de jonction. Cette notion est la coupe.

Les éléments comportementaux d'un port sont considérés comme des ***points de jonction potentiels***. Un élément comportemental inclus les actions attachées aux point d'accès de type ActionPoint (ACTOAP) associés avec des aux point d'accès de type DATA Point ( send, receive, updated, changed).

Un point de jonction est identifié par un nom hiérarchique spécifiant sa position dans l'architecture. Le nom de point de jonction est composé de quatre parties séparés par le caractère point **'.',**

**NomInstanceComposant.NomInstancePort.NomInstancePointD'accés.NomAction**

### 3.2.5 Les coupes /pointcut :

Nous nous intéressons dans cette partie à présenter la spécification d'un pointcut au niveau du langage 3ADL.

Un pointcut est défini comme un ensemble de joinpoints. Un joinpoint est un endroit bien défini dans le code primaire où l'aspect va couper l'application. Les joinpoints sont des candidats utilisés pour accomplir la composition entre la description des aspects et la description de base du système logiciel.

Une coupe sélectionne un ensemble de points de jonction. Comme nous avons déjà mentionné, un pointcut architectural peut être composé d'un ensemble de joinpoints. Pour regrouper ces derniers, nous avons eu recours aux mécanismes de quantification au niveau architectural. Pour ce faire, nous introduisons les opérateurs ET logique (&&), OU logique (‖), (**^**) XOR logique ainsi que les wildcards telque '*' pour décrire un ensemble de joinpoints invoquant le même comportement d'un aspect,

La spécification d'une coupe se base sur les expressions rationnelles GNU (**GNU regexp[13] library).** LA coupe est identifiée par un nom hiérarchique spécifiant sa position dans l'architecture. Le nom de coupe est composé de quatre parties séparés par le caractère point **'.': NomInstanceComposant**‖TypeComposant**.NomInstancePort**‖TypePort**.NomI nstancePointD'accés**‖ruleName**.rule** (si la 3eme partie est un ***rule name***).

| |
|---|
| **PointcutSpecification ::= pointcut** PointcutIdentifier **{PointcutExpression;}** |

Une coupe peut être spécifiée en utilisant le nom complet de nom abstrait de la coupe (les quatre parties de nom hiérarchique de nom de la coupe) ou bien une partie de nom abstrait ( le nom de la coupe sans les autres parties) qui doit se terminer par un '.' pour indiquer qu'il s'agit d'une partie de nom.

[13] Sont une famille de notations compactes et puissantes pour décrire certains ensembles de chaînes de caractères. Elles permettent de rechercher automatiquement des morceaux de texte ayant certaines formes, et éventuellement remplacer ces morceaux de texte par d'autres.

```
/// IASA 3ADL :
/// X25CM component type
Controlpart{…….}
Optionpart{
AspectPart{
AspectComponents {LogCmp logCmp;}
Connectors{description des ports}
Pointcuts{
log_alarm= {alarm.receive};
log_enable={pEnable.*.receive};
log_alaena=log_alarm+log_enable;
// trace pointcut
secFTPTrace={FTPClientPort.getTicketFile.rule}
}}}
```

Figure 3.7. La définition d'une Coupe avec 3ADL

Une coupe peut être définit a partir d'autres coupes par usage des opérateurs union (+) différence (-).

La figure 3.7 montre quelques exemples des spécifications des coupes, la coupe log_alarm et log_enable cible une action de type **receive** .

Les **coupes sur les traces d'exécution** (Trace pointcuts) permettent de capturer des comportements plus complexes dans un système. Elles définissent une **séquence de points de jonction passés**, c'est-à-dire sur un historique d'événements, d'enchaînement d'occurrences dans un Système.

Elles vont plus loin que les coupes strictement structurelles et caractérisant un seul événement. Grâce à ce genre de mécanisme, 3ADL [58] permet d'élaborer des liaisons d'aspect capturant des séquences d'événements. Les coupes sur les traces d'exécution permettent donc d'automatiser ce mécanisme. Elles augmentent le pouvoir d'expression des coupes dans l'ADL 3ADL, **secFTPTrace** (voir figure 3.7) est un exemple d'un point de jonction sur tarce.

Des **coupes** correspondant à la notification d'événements endogènes (invocations de messages sur interfaces POrt, modification des connexions. . .) ou exogènes (grâce à un Framework pour le développement d'applications sensibles au contexte) ;

### 3.2.6 Déclaration des advices:

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

Concrètement, un code advice est un bloc d'instruction qui spécifie le comportement de l'aspect. Un code advice est toujours associé à une coupe ou plus exactement aux points de jonctions sélectionnés par cette coupe. En effet, un code advice n'est jamais appelé manuellement, mais il est invoqué chaque fois qu'un point de jonction, sélectionné par la coupe à la quelle il est associé, survient.

Un code advice peut être exécuté selon trois modes : avant, après, ou autour d'un point de jonction. Lorsqu'il est exécuté autour du point de jonction, il peut carrément remplacer l'exécution de ce dernier, ou bien lui redonner le contrôle.

La syntaxe de chaque advice est de la forme :

**adviceSpecification ::= advice** AdviceDeclaration **{PointcutReference;}**

```
advices{
inject log_alarm.log after log_alaena;
inject LogCmp.log around logFTPTrace;
}
```

Figure 3.8. La définition d'un advice avec 3ADL

### 3.2.7 Mécanisme d'injection d'un Advice:

Le mécanisme d'injection d'un Advice dans un composant métier se fait comme suit:

On injecte un ASPOAP client dans le port client de composant métier se que vas modifie la structure et le comportement de port client conseillé (advised client port) et tous les ports clients connectées au port serveur conseillé.

L'injection de l'advice spécifie:

- Une coupe pointcut
- Un type d'advice (before, after, around)

- Et les interconnections aspectuel (liaison aspects) entre tous les porte contenant le client ASPOAP et le port contenant l'advice ASPOAP en utilisant un connecteur primitif de transport *Primitive transport Aspect.*
- On désigne par un **aspectuel connector** un connecteur qui relie l'ASPOAP client à l'Advice ASPOAP.
- Lorsque on injecte un advice de type **around** on injecte un DAOP avec le ASPOAP client dans le port de composant aspectisable. Si le point de jonction se trouve dans un port serveur ServerPort, tous les ports client connectés a se port vont automatiquement être conseillé (advised). On peut annuler l'injection d'advice pour les ports qui n'ont pas besoin d'être conseillé on utilisant une instruction **remove Advice.**
- Le DAOP additionnel injecté est utilisé pour se renseigner sur le statut de *proceedAction* dans le port advice.

L'injection d'advice se fait de deux manières:

- Avec l'instruction inject;

```
Void Inject (ASPOAP_client, NomComposantAspect, NomComposantIASA, Type d'advice);
```

**Exemple:**

Void **Inject (logCmp.log, logCmp ,GAB.pRetrait, before)**

- Graphiquement avec *IASA studio* en plaçant un ASPOAP client sur le composant IASA graphiquement sur la zone graphique.

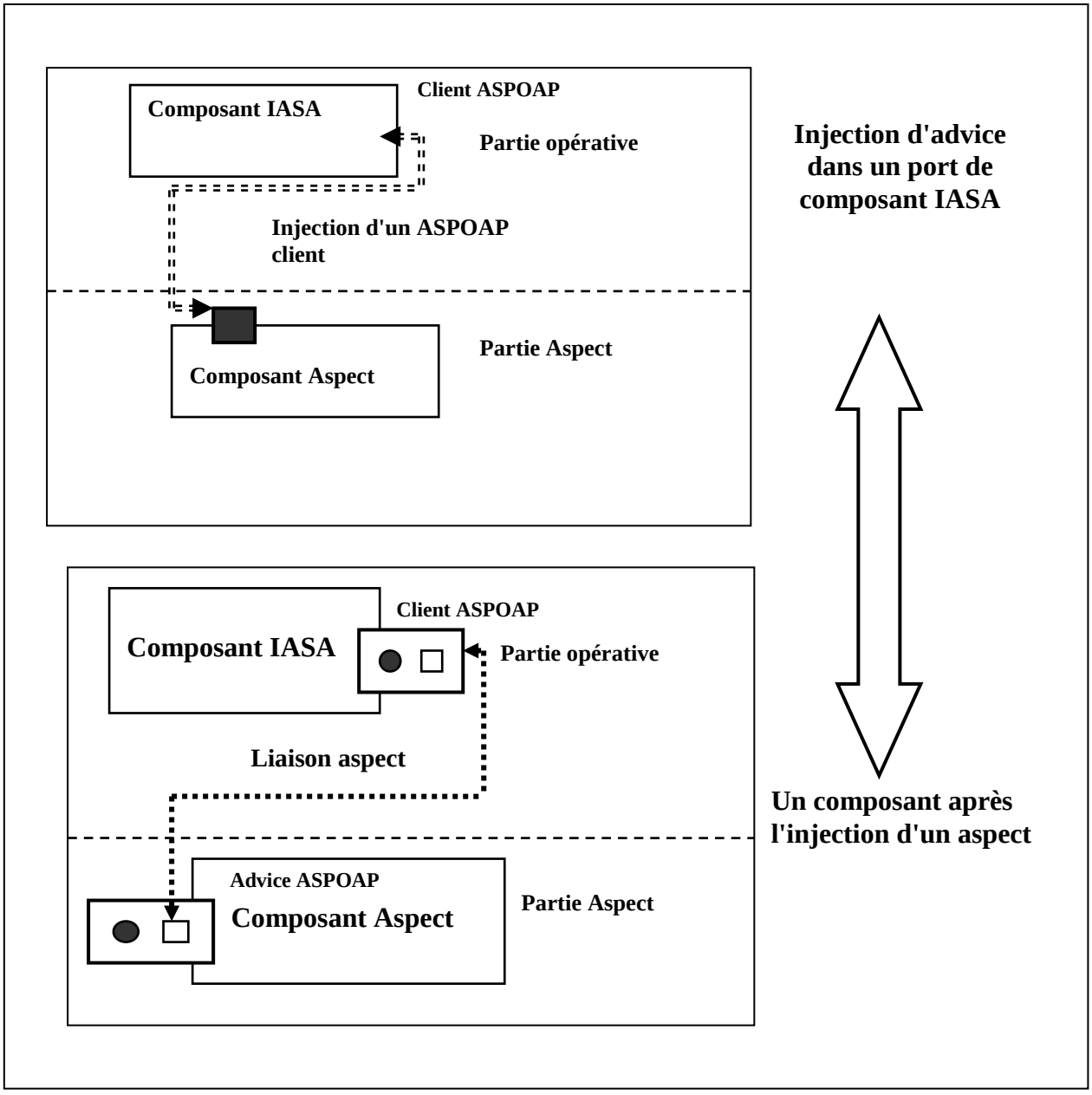



Figure 3.9. Le mécanisme d'injection d'un Advice

3.2.8 Tissage d'aspect:

Les connecteurs constituent un élément architectural au même titre que les composants. Ainsi, ce sont des entités de premier plan dans une architecture. Cette considération identique pour les composants et les connecteurs est l'atout principal des architectures.

Le tissage d'aspect consiste à établir une connexion qu'on appel **Une liaison aspect** entre un composant aspect et un composant aspectisable et plus

précisément la connexion établi sera entre un ASPOAP_advice et l'ASPOAP client injecté au composant aspectesiable.

Le tissage d'aspects au moment de la conception rejoint le paradigme de l'ingénierie dirigée par les modèles. En effet la maîtrise de la complexité des logiciels modernes (tant pour leur production que pour leur validation) passe de plus en plus par la notion de modélisation, c'est à dire de l'utilisation efficace d'une représentation simplifiée d'un aspect de la réalité pour un objectif donné. Si la modélisation peut-être vue comme la séparation des différents besoins fonctionnels et préoccupations extra-fonctionnelles (telles que sécurité, fiabilité, efficacité, performance, ponctualité, flexibilité, etc.) issus des exigences, la conception du logiciel consiste réciproquement à fusionner (ou tisser) ces différents aspects dans la trame fonctionnelle.

A l'exécution, le tissage d'aspects permet de changer le comportement d'une application

Le tissage d'aspect dans IASA s'effectue sois avec 3ADL-ADL, ou a travers IASA Studio. IASA Studio permet également de définir une opération de tissage et a été étendu pour un tel support. En particulier le contrôle de tissage est visible dans la console.

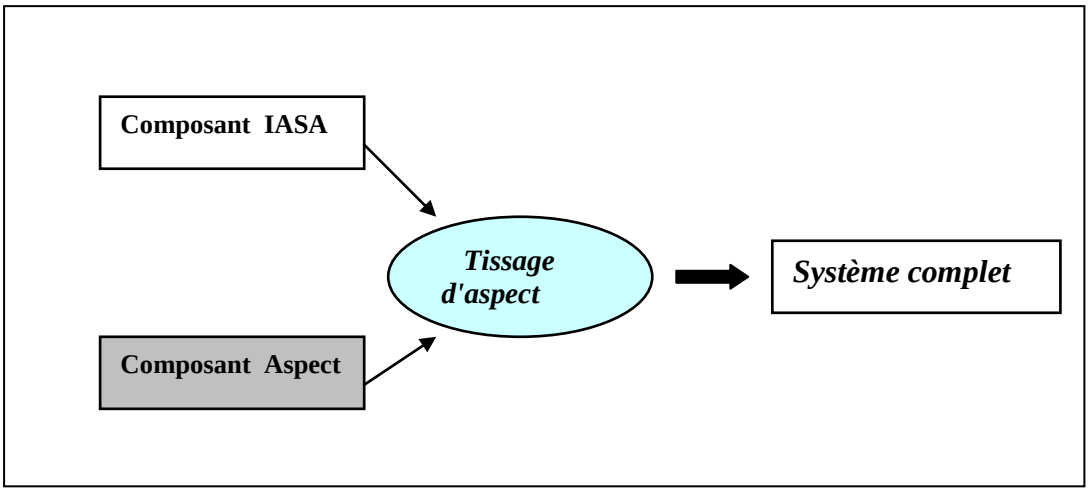



Figure 3.10. Le mécanisme de tissage d'aspect

3.3 La composition et L'ordres d'exécution des aspects

La question de la multiplcité se pose quand l'exécution d'un composant doit être altérée par plusieurs aspects. Ces derniers peuvent éventuellement être en conflit s'ils interviennent à un même point de jonction.

*l'ASPOAP client* est fourni avec un nombre qui spécifie l'ordre des aspects exemple (pAuthAdvice_1_aroundFirstAction). Ce sont ensuite utilisées pour déterminer l'ordre de lancement des advices lorsque deux ou plusieurs aspects sont attachés au même point de jonction.

Lorsque plusieurs composants d'aspects sont tissés, soit directement, soit par une expression de coupe sur le même composant de base, leur ordre d'exécution est celui de leur tissage : c'est à dire les composants d'aspect tissés en premier sont exécutés en premier.

**Un autre mécanisme:**

Est d'implémenter la méthode **GetORder** (**boulean aspect**) (voir figure 3.9).
La file est peut-être la structure la plus immédiate, elle modélise directement les files d'attente gérées selon la politique premier-arrivé, premier-servi. La pile est plus informatique par nature.
Chaque composant apsectisable est muni d'une file d'attente, chaque fois qu'un ASPOAP client est injecté à un composant métier cette file d'attente intercepte l'identifiant de composant aspect responsable de cette injection.

Ainsi, le Tissage suivre la chaîne définit dans la file d'attente.

```
static integer GetOrder(boulean aspect) {
   Fifo f = new Fifo () ;
   int nbAspect = 0, nbFedUp = 0 ;
   int tFree = 0 ;int ID= 0;  // l'identifiant de composant aspect

//On peut limiter le nombre d'aspect injectable a un seule composant métier

   for (int t = 0 ; t < tMax ; t++) {
  // Si un composant est aspectisé (si on a injecté au moins un ASPOAP client cette valeur est a
vrai)
if (occurs(Aspect)) {
      nbAspect ++ ;
      f.add(new ID(t)) ;
    }
        }
  }
   return (integer) nbAspect ;
 }
```

Figure 3.11. La méthode GetOrder

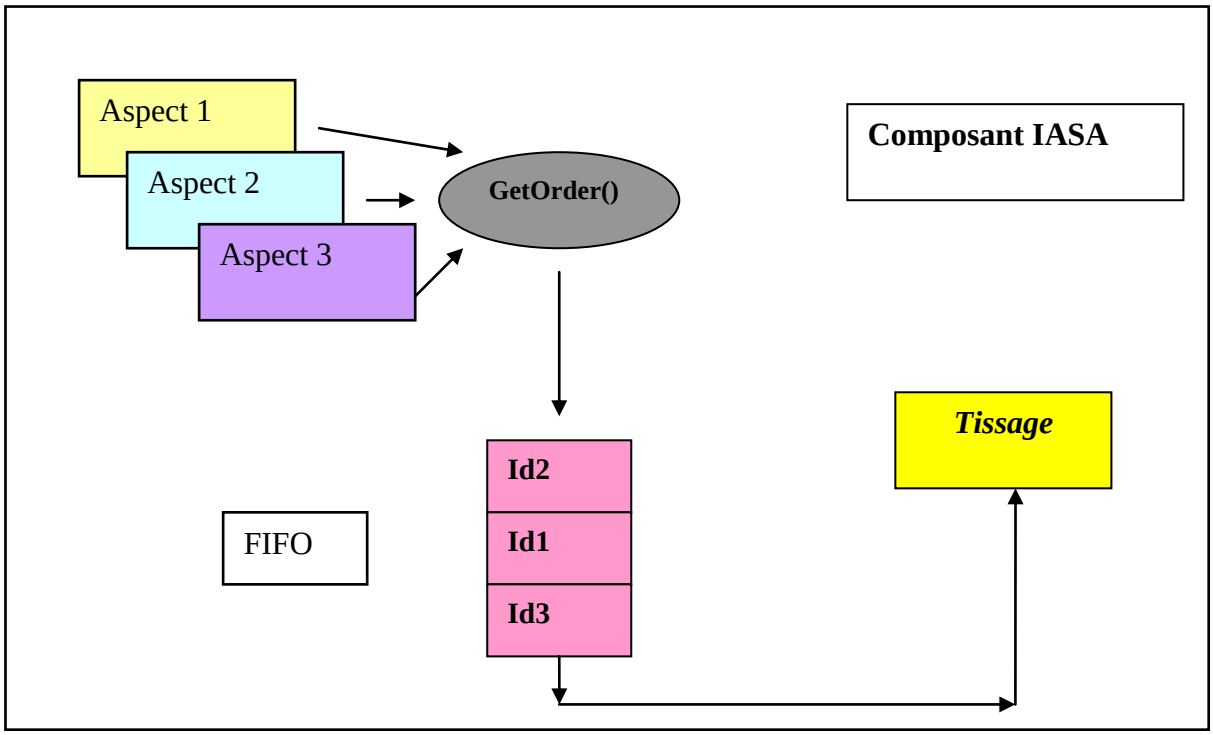



Figure 3.12. Le mécanisme d'ordonnancement d'aspect

3.4 La réutilisation des aspects :

L'idée centrale du développement orienté aspect est de permettre une plus grande agilité vis-à-vis des variations dans les exigences en permettant d'une part une séparation explicite des préoccupations (pouvant alors évoluer indépendamment) et d'autre part l'automatisation de leur composition : c'est ce qu'on appelle le tissage d'aspects. Ce tissage d'aspects peut être effectué à différents moments du cycle de vie du logiciel : la conception, la programmation, ou même l'exécution.

Il s'agit de définir des aspects génériques pour qu'ils puissent être utilisés dans des contextes différents. Cette définition nécessite de découpler les traitements des aspects des points de jonction. Une autre facette de la réutilisation concerne l'étude de la possibilité de composer des aspects existants pour définir de nouveaux aspects.

3.5 Evaluation:

Le modèle de composant IASA permet la définition, la configuration, et la reconfiguration dynamique de composants et offre une séparation claire entre les besoins fonctionnels et non fonctionnels d'une application.

- ✓ Construit comme un modèle de haut niveau, son objectif est de fournir une grande modularité et des possibilités d'extension étendues.
- ✓ Le modèle est récursif : un composant est de type primitif ou composite. Dans ce dernier cas, le composant correspond à un assemblage d'autres Composants primitifs ou composites.
- ✓ Un composant peut également être partagé entre différents composites.
- ✓ La réutilisation **:** Les composants Aspects sont réutilisables puisqu'ils représentent exclusivement le métier de l'aspect sans prendre compte du domaine dans lequel il va être appliqué.

| | |
|---|---|
| **Point de jonction sur trace (Tracecuts)** | **OUI** |
| **Type de tissage : dynamique/ Statique** | **Configuration architecturale ensemble de liaisons aspects entre les ASPOAP advice et les ASPOAP client** |
| **Composition d'aspect** | **possible** |
| **Réutilisation aspect** | **OUI** |
| **Spécification du tissage (Application)** | **Dans la partie configuration architecturale** |
| **Indépendance coupe / aspect** | **OUI** |
| **Spécification Coupe** | **SEAL** |
| **Type de Point de jonction** | **Un ensemble d'actions: les entités comportementales de base qui échangent des flots de contrôle et des flots de données à travers des *ActionPin*.** |
| **Type de Conseil** | **aroundFistAction, AfterAction() aroundLastActionpublic ProceedAction beforeAction()** |
| **Gestion des Aspects au niveau d'un point de Jonction : Ordre d'exécution des aspects** | **Spécifié explicitement on définissant un ordonnancement arbitraire entre aspects à l'aide de interfaces GetOrder ou un mécanisme d'établissement d'ordre** |
| **Type d'advice par point de jonction** | **Hétérogène** |
| **Nombre d'advice supporté par un point de jonction** | **illimité** |

Tableau 3.1 Bilan sur IASA AOP

# Chapitre 4 : IASA Studio

*PRÉAMBULE*

*Dans ce chapitre, nous présentons l'atelier de génie logiciel appelé IASA Studio développé au cours de cette thèse. Cet outil nous sert de plate-forme d'évaluation pour les modèles et les analyses associés à ce travail. Ce chapitre définit les choix architecturaux retenus pour la mise en œuvre de ces multiples modèles, leur analyse et la production de code à partir de ces modèles.*

## 4.1 Introduction:

La dernière contribution de cette thèse concerne le développement d'un atelier de génie logiciel IASA Studio. L'objectif du projet IASA Studio consiste à proposer une chaîne d'outils pour la construction, la spécification avec l'ADL 3ADL d'une architecture logicielle. IASA Studio permet de concrétiser l'ensemble des propositions autour d'IASA.

Ce chapitre présente, tout d'abord, l'architecture d'IASA Studio et décrit les mécanismes et les technologies utilisés pour la création de l'atelier de génie logiciel.

L'accent est mis par la suite sur la mise en oeuvre de la vérification de la cohérence et la génération de code vers des modèles de composant existants. Finalement, nous présenterons comment IASA studio permet un processus de construction incrémentale de nouvelles descriptions d'architecture logicielle.

4.2 Présentation générale d'IASA Studio:

IASA studio peut être décrit comme suit: (voir Figure 4. 1 si dessous).

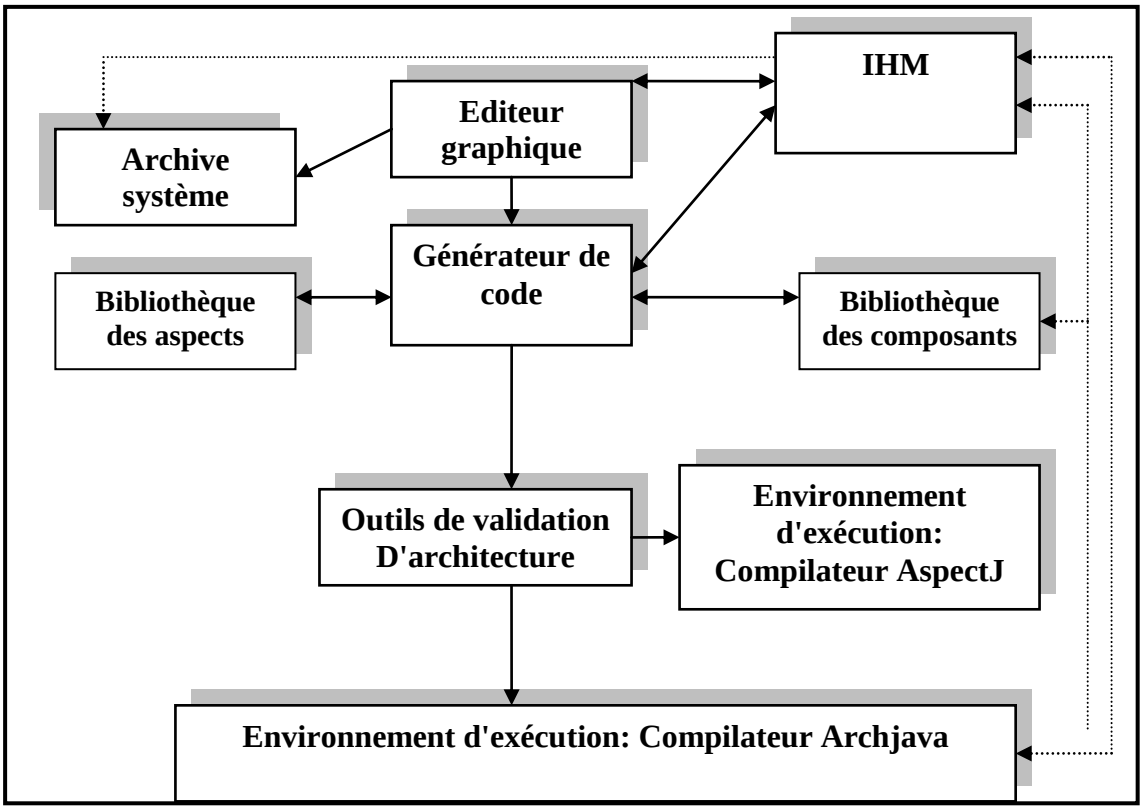



Figure 4.1. L'architecture globale de système

1. Module1: L'éditeur graphique.
2. Module 2: Le générateur de code Archjava.
3. Module 3: Un outil de validation d'architecture logicielle.
4. L'IHM: L'interface utilisateur.
5. Bibliothèques des composants
6. Archive système.

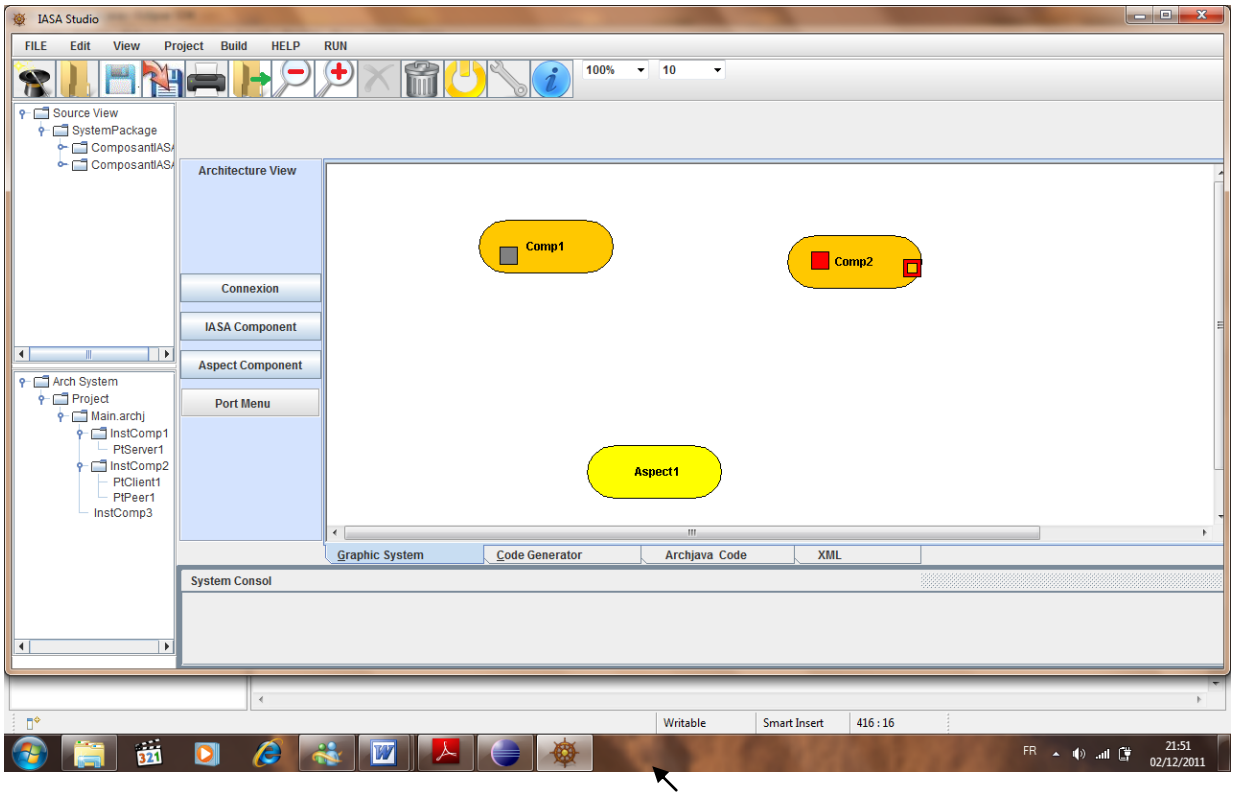



Figure 4.2. IASA studio

### 4.2.1 L'Editeur graphique:

L'éditeur graphique est le composant primordial, essentiel dans l'architecture de ce système, car il constitue le cœur de toute opération de spécification d'architecture logicielle, il offre à l'utilisateur de système la possibilité de spécifier chacun des éléments de son architecture de manière graphique en lui offrant les outils suivants:

- Une palette de dessin qui contient un ensemble de boutons pour dessiner des composants graphiques primitifs métiers ou aspect, des ports, des ASPOAP clients/advices, établir des connexions et des liaisons aspect.

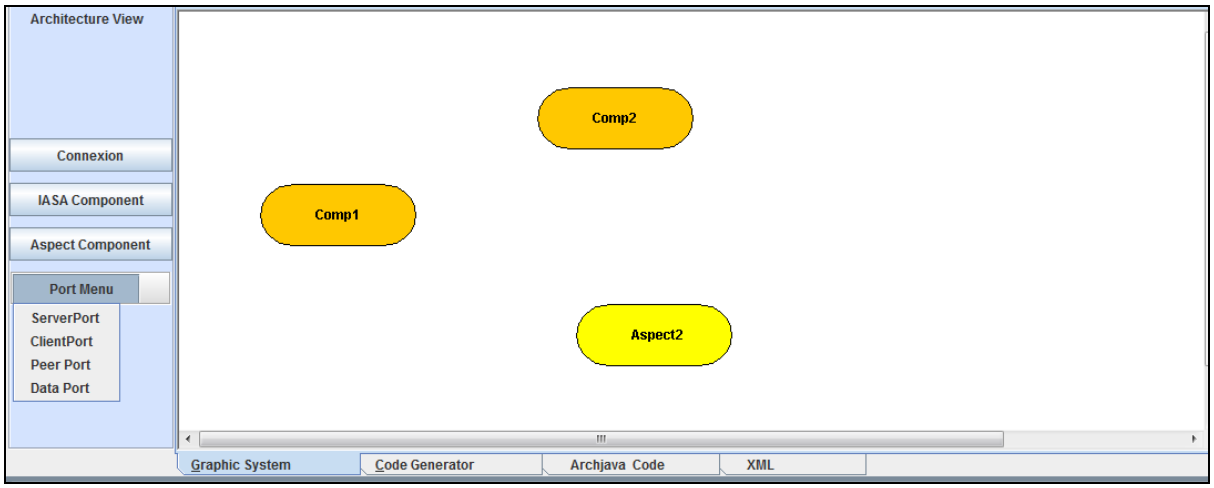



Figure 4.3. La zone graphique

- Une zone de dessin pour que l'utilisateur place les éléments de son architecture (composantIASA, composant Aspect, connecteurs, port….)

Ce module offre plusieurs fonctionnalités qui facilite la tâche a l'utilisateur de système, parmi ces fonctionnalités nous citons:

- La possibilité de déplacer les composants crées tout au long la zone de dessin.
- La possibilité de supprimer des composants et des ports de la zone de dessin.
- La possibilité d'importé des composants de la bibliothèque des composants primitifs ou complexes dans un but de les réutiliser pour spécifier d'autre architecture logicielle.
- La possibilité de réaliser des interconnexions entre les composants primitifs graphiquement.
- La possibilité d'injecter des ASPOAP clients dans des composants métier.
- Réaliser des liaisons aspects entre les ASPOAP clients et les advices ASPOAP.

### 4.2.2 L'arbre *Architecture system:*

Le premier panneau situé en haut à gauche, nommé *Architecture system*, permet d'obtenir une vue arborescente d'un composite. C'est une zone qui présente la liste crée dans la zone graphique, chaque nœud de cet arbre est une instance d'un composant qui fait partie de l'architecture spécifié.

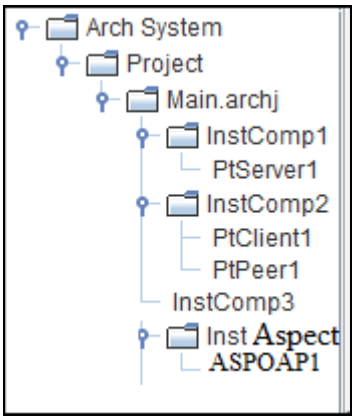



**Figure 4.4. Arbre dynamique Architecture Système**

4.2.3 L'arbre Source View:

L'arbre *Source View* situé en haut bas. Elle sert à donner un aperçu de tous les composants de l'architecture spécifié et d'autre part chaque nœud de cette arborescence sert a une interface entre l'utilisateur et le système pour lui offrir les fonctionnalités telles que:

- L'ajout d'un composant a la bibliothèque afin de permettre l'utilisation.
- Sauvegarde de code correspondant à ce composant.
- La compilation de code Archjava généré.

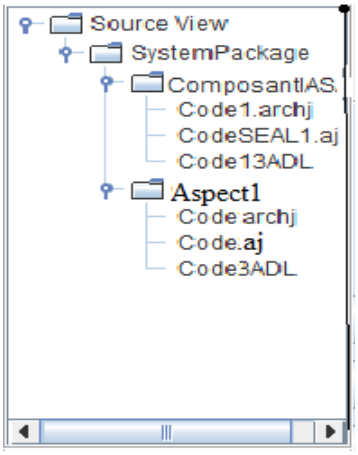



**Figure 4.5. Arbre dynamique Source View**

4.2.4 Le générateur de code:

La partie projection d'une description d'architecture IASA vers ArchJava se place dans le contexte actuel de la transformation de modèles. Notre modèle de départ est le modèle abstrait de composant IASA, le modèle d'arrivée est le

modèle concret ArchJava. Pour ce faire, nous avons utilisé cette fonctionnalité d'IASA Studio.

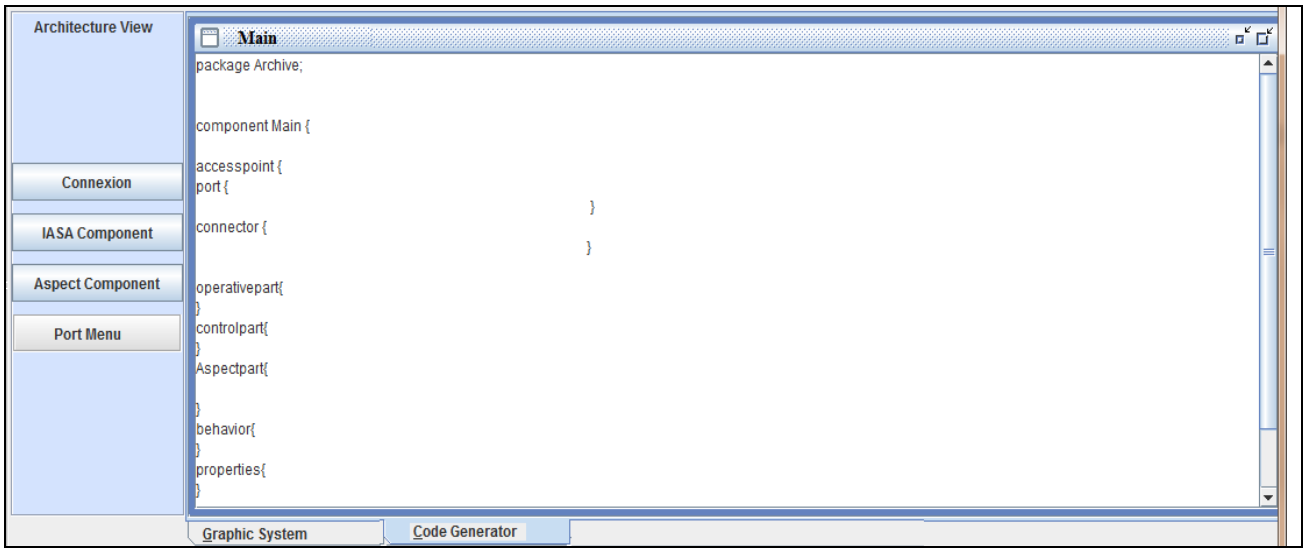



Figure 4.6. Générateur de code

#### 4.2.4.1 Génération de code 3ADL:

Cette outil permet de générer automatiquement un code 3ADL correspondant à l'architecture logicielle crée sur la zone de dessin. Ce module génère de code systématiquement chaque fois un utilisateur ajoute un composant, un port ou établit une connexion.

#### 4.2.4.2 Projection vers ArchJava:

La projection vers ArchJava est la plus facile, la proximité des éléments structurels entre les deux modèles permet de projeter directement les concepts composants primitifs, composites et ports vers ArchJava. En outre, ArchJava distingue pour la notion de connecteur, les notions de classe de connexion qui identifient des ports de types de composant qui peuvent être liés par un connecteur à l'exécution et la notion de connecteur qui lie deux ports de composant.

La génération de code vers ArchJava crée pour chaque composant de l'architecture le code relatif à son interface.

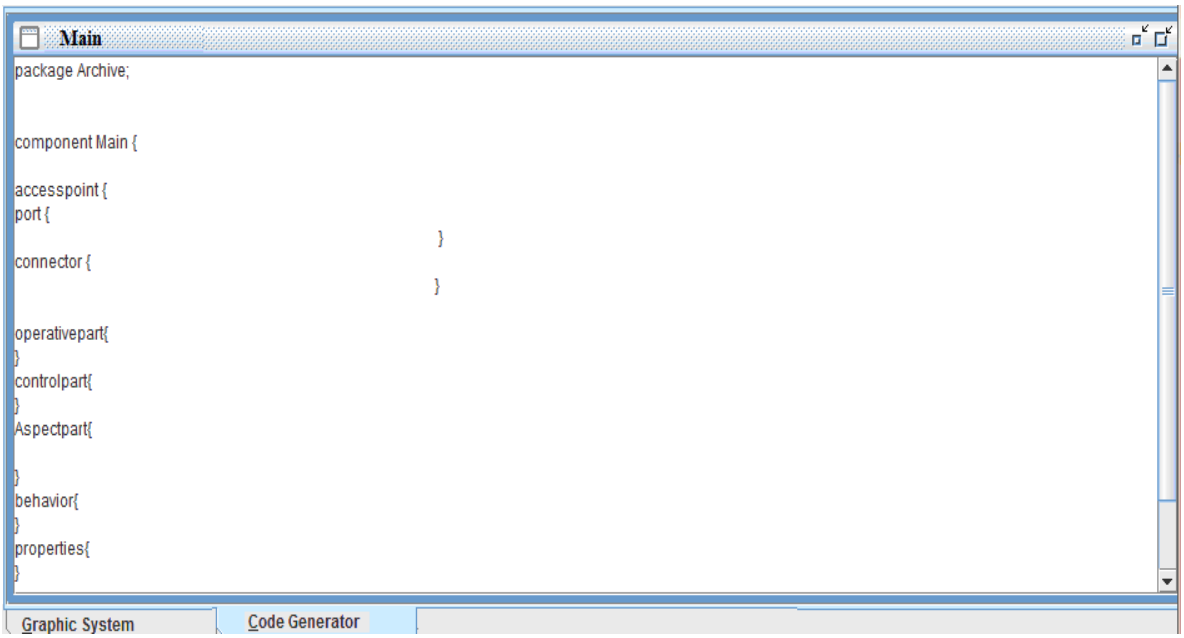



Figure 4.7. Le code Archjava correspondant au composant Main (Enveloppe)

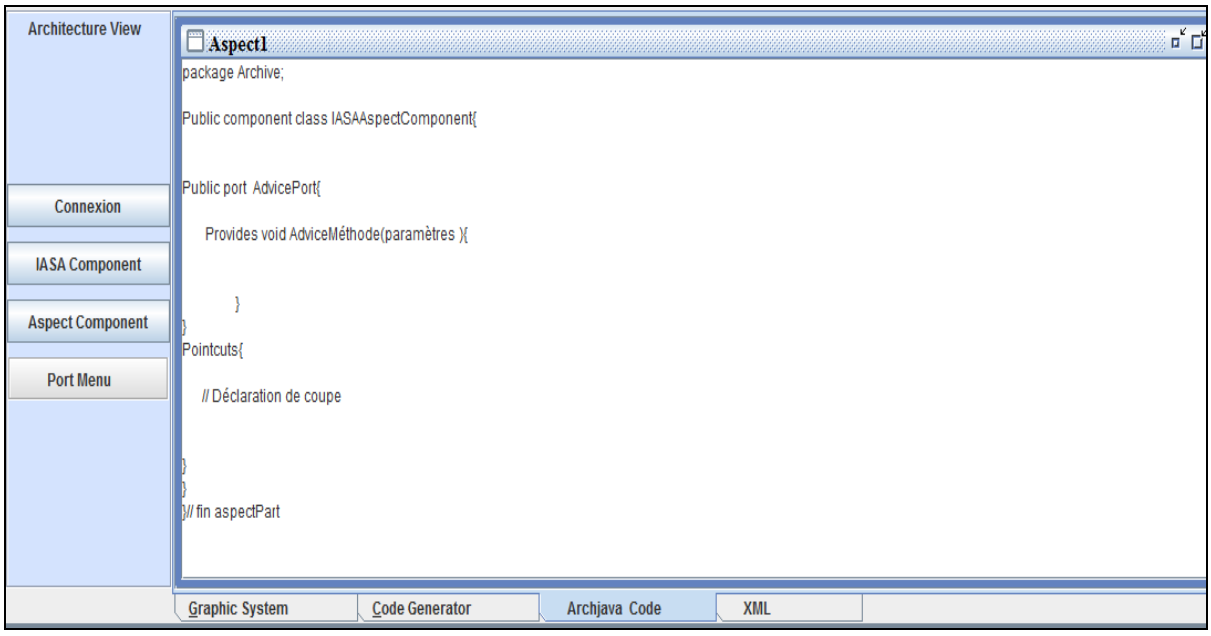



Figure 4.8. Le code Archjava correspondant a un composant Aspect

4.2.4.3 Projection vers java:

La projection vers Java est spécifiquement destinée aux composants IASA métiers afin de pouvoir tisser de code sur ses derniers avec aspectj.

4.2.4.4 Projection vers aspectj:

Lorsque en place un composant aspect sur la zone de dessin l'outil de générateur de code génère de code aspectj pour le composant correspondant afin

de le pouvoir tisser sur les composants métiers (composant IASA aspectisés) avec le compilateur *ajc (pour AspectJCompiler)*.

AspectJ est aujourd'hui une implémentation orientée aspect qui fournit un excellent support pour appréhender les concepts de la programmation orientée aspect.

Sa plus grande force réside dans le fait qu'il est issue des travaux de la même équipe à l'origine de l'orientée aspect. AspectJ est donc une extension orientée aspect du langage de programmation Java. Il permet de déclarer des aspects, des coupes, des codes advices et des introductions. Il offre aussi un tisseur d'aspect appelé *ajc (pour AspectJCompiler)* qui prends en entrée des classes java et des aspects, et produit en sortie des classes dont le comportement est augmenté par les aspects. AspectJ permet de définir deux types de transversalités avec les classes de base : transversalité statique (static crosscutting) et transversalité dynamique (dynamic crosscutting).

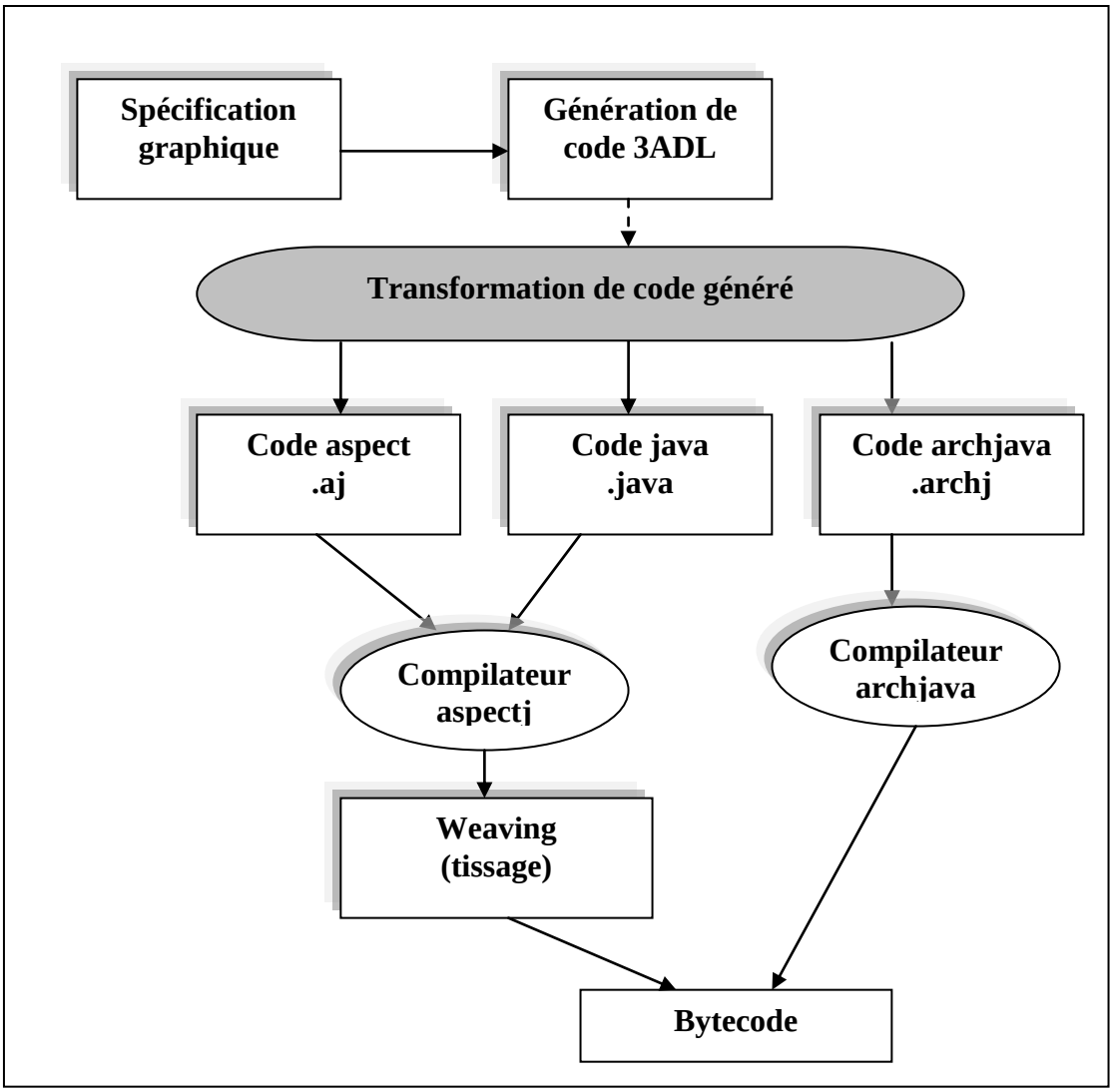



Figure 4.9. Mécanisme de génération de code

4.2.5 Outils de validation de l'architecture logicielle:

Ce module offre un ensemble d'outils qui permettent de valider l'architecture logicielle spécifié, cette validation consiste a :

- Compilation de code archjava généré,
- L'exécution de code archjava généré.
- Tissage d'aspect avec Aspectj.

Parmi les principaux outils offerts par se module:

4.2.5.1 Outil Compiler:

C'est un outil responsable de compiler les différents codes généré par le module générateur de code.

4.2.5.2 Outil Weaver:

C'est un outil responsable de tisser les différents aspects.Le tissage (weaving) est le processus qui prends en entrée un ensemble de composant

aspects et une application de base et fournit en sortie une application dont le comportement et la structure sont étendus par les aspects.

4.2.6 Un consol d'affichage:

C'est un terminal dédié a l'affichage des différents résultats de compilation/ exécution.

4.2.7 Bibliothèque des composants:

C'est une librairie où les composants crées serons stocker pour qu'un utilisateur de système puisse les réutiliser pour spécifier d'autres architectures logicielles en les important tout simplement de cette bibliothèque.

4.2.8 Bibliothèque des aspects:

C'est une librairie où les aspects crées serons stocker pour qu'un utilisateur de système puisse les réutiliser pour les injecter dans d'autres architectures logicielles en les important tout simplement de cette bibliothèque.

4.2.9 Archive de système:

C'est un répertoire "package" où le code des composants sera sauvegardé ainsi que d'autres informations concernant les composants.

# Chapitre 5 : Validation de l'approche 3ADL

## 5.1 INTRODUCTION:

L'approche intégrée a été évaluée dans le cadre de la réalisation d'une application de gestion commerciale d'une banque muni d'un Guichet automatique de Banque.

Notre objectif de choix de cette application peut se résumer en se que suis:

1. Mettre en œuvre les concepts de l'approche intégrée (les modèles de composant, port et connecteur, composant aspect, injection d'advice, liaison aspect etc.

2. Evaluer l'approche intégrée en implémentant l'application cible avec un langage de programmation. Le langage choisi est ArchJava. Le choix de ArchJava est du principalement à sa capacité de représenter les concepts de composants et de connecteurs.

On va présenter la spécification de deux composants composite ***BanK et DistributeurAutomatique.***

## 5.2 Application de mise en place d un Guichet Automatique de Banque (G.A.B.)

La plupart des GAB sont connectés à un réseau interbancaire, ce qui facilite le retrait et le dépôt dans des GAB n'appartenant pas à la banque où le client possède un compte.

De plus, les GAB installés contiennent de moins en moins de circuit intégrer spécialisés et font de plus en plus appel à un PC venant avec un système d'exploitation courant.

☒ Le « Distributeur de billet de banque » est utilisable par trois catégories de personnes :

- Les « clients » qui choisissent la somme désiré, s'authentifient, et retirent les billets distribués ;
- Les « Receveur de banque » qui alimentent le GAB avec des billets;
- Les «employés de maintenance » qui peuvent vérifier le bon état de fonctionnement du distributeur.

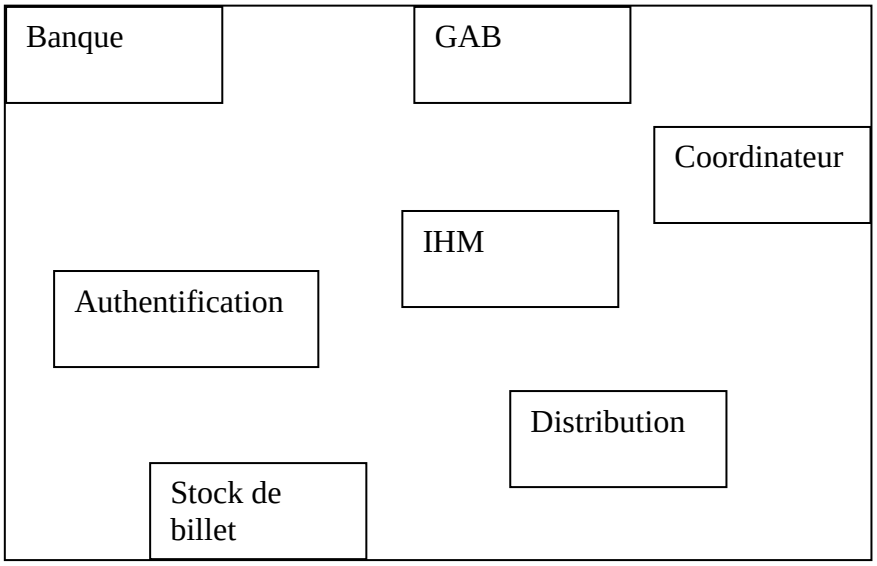



Figure 5.1 Le composite « Distributeur automatique» et ses composants constituants

☒ Un « Distributeur [Figure 5. 1] est formé de composants logiciels de base qui sont :

- Une interface homme-machine («IHM ») permettant aux diverses catégories de personnes de préciser les opérations désirées ;
- Un composant «Authentification » chargé d'identifier chacune des catégories de personnes et de plus, pour le client, de proposer un moyen de paiement adapté à l'utilisation du distributeur;
- Un composant « Stock de billet » pour l'argent à délivrer;
- Un composant « GAB » chargé de prélever de l'argent du stock et de fournir au client ce qu'il souhaite;
- Enfin, UN composant «coordinateur » qui a un rôle de chef d'orchestre relativement aux autres composants du distributeur.

5.2.1 Spécification de composant BanK:

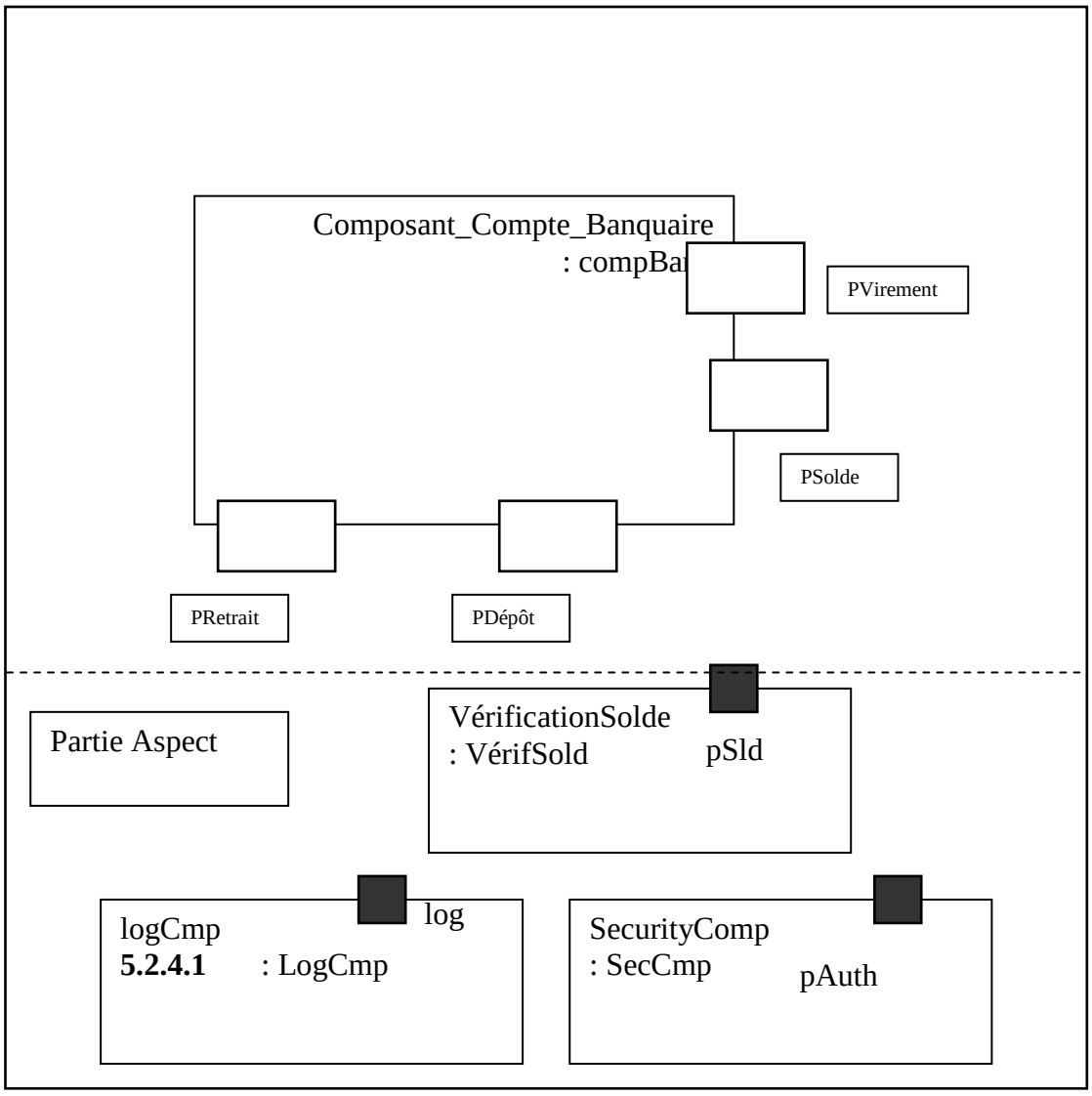



Figure 5.2. Composant Bank

5.2.2.1 La spécification de composant *BANK AVEC 3ADL:*

```
package nomDePackage;
import listeDePackage // package de composant et de connecteurs
component BANK {
accesspoint { // Définition de types internes de points d'accès de données }
port { // Définition de types internes de port }
connector { // Définition de types internes de connecteurs. }
operativepart{ // description de la partie opérative:}

component { // Définition de types internes de composant.}
/// Définition des instances
Composant_Compte_Banquaire : compBank;
connector { // Définition de types internes de connecteurs. }
ports{ // définition des ports du composant}

AspectPart{// description de la partie aspect: instance de composant aspect et connexion
inter composant de la partie aspect avec les composants de la partie opérative
```

```
Aspectcomponent { // Définition de compsant aspects
LogCmp logCmp;
SecCmp SecurityComp;
VérificationSolde VérifSold;
}
connector { // Définition de types internes de connecteurs aspects }

pointcuts{
l_Solde={ psolde.recieve};
log_Solde= {pdêpot.send};
log_compt= pRetrai.receive;
log_Auth={ l_Solde + log_Solde+ log_compt };
log_Security{ log_Solde+ log_compt };
}
advices{
inject psId.VérifSold before log_compt;// vérification de solde avant retrait
inject secCmp.pAuth before log_Auth ;// logging client avant retrait /depot versement
inject logCmp.log after log_Security;//
}
}// fin aspectpart

behavior{ // Comportement du composant; Ce comportement représente
// celui du composant OpPartController de la partie control
}
properties{ // Spécification des propriétés non fonctionnelles. Pour l'instant,
// sont décrites les informations de déploiements
}
} // Fin description type de composant
```

Figure 5.3: La spécification de composant *BANK AVEC 3ADL:*

### 5.2.2.2 Composant_Compte_Banquaire :

```
Public component class Composant_Compte_Banquaire extends IASA_Component{
// Instantiation initial des Datapoint || Actionpoint
  Private int Solde;
// Définition des ports
  Public port  PRetrait{
Provides void Retrait(int SommeRetrait );
}
Public port  PDepot{
Provides void Depot(int SommeDepot );
}
Public port  PSolde{
Provides void GetSolde(int Solde );
}
Public port  PVirement{
Provides void SetSolde(int Solde );
}


Public void Retrait(int SommeRetrait ){
Solde=Solde-SommeRetrait;
}
Public void Depot(int SommeDepot ){
Solde=Solde+SommeDepot;
}
Public int GetSolde(int Solde ){
   return Solde;
```

```
}

Public void SetSolde(int Solde );{
   this.Solde=Solde;
}}// end composant CompteBanquaire
```

Figure 5.4. L'implémentation de composant Composant_Compte_Banquaire avec Archjava

### 5.2.3 Spécification de composant composite *Distributeur de billet* :

```
package nomDePackage;
import listeDePackage // package de composant et de connecteurs

component Distributeur_billet{
accesspoint { // Définition de types internes de points d'accès de données }
port { // Définition de types internes de port }
connector { // Définition de types internes de connecteurs. }
operativepart{ // description de la partie opérative:}
component { // Définition de types internes de composant.}
/// Définition des instances
GAB  gabCmp;
Coordinateur coordCmp    ;
InterfaceIHM ihmCmp    ;
Stock_billet stckCom;
connector { // Définition de types internes de connecteurs. }
ports{ // définition des ports du composant
pUser port;
}
AspectPart{// description de la partie aspect: instance de composant aspectet connexion inter
//composant de la partie aspect avec les composants de la partie opérative
Aspectcomponent { // Définition de compsant aspects
LogCmp logCmp;
AuthentificationComp  AuthCmp:
SecCmp SecurityComp;
VérificationSolde VérifSold;
}
connector { // Définition de types internes de connecteurs aspects }
pointcuts{
l_Solde={ psolde.recieve};
log_Solde= {pdêpot.send};
log_compt= pRetrai.receive;
log_Auth={ l_Solde + log_Solde+ log_compt };
log_Security{ log_Solde+ log_compt };
l_AuthPtCut {   P_EvtAuth. Receive; }
l_AuthPtCut2 {   P_Auth. Receive; }
}
advices{
inject psld.VérifSold before log_compt;// vérification de solde avant retrait
inject secCmp.pAuth before log_Auth ;// logging client avant retrait /depot versement
inject AuthCmp.pEvtAuth before l_AuthPtCut;//
inject AuthCmp.pAuth beforel_AuthPtCut2;//
}}// fin aspectpart
```

```
behavior{ // Comportement du composant; Ce comportement représente
// celui du composant OpPartController de la partie control}
properties{ // Spécification des propriétés non fonctionnelles. Pour l'instant,
// sont décrites les informations de déploiements
}} // Fin description type de composant
```

Figure 5.4**:** La spécification de composant *Distributeur de billet AVEC 3ADL:*

☒ *Administrer un GAB :*

Pour le bon fonctionnement de guichet automatique de billet on doit assurer les conditions suivantes:

• Vérification de la présence de billet dans le distributeur.
• Vérification que le GAB est en connexion permanente avec le système central de la banque.
• Vérification que le GAB est en connexion permanente avec le système central des cartes bleues.
• Vérification de la possibilité d'obtenir systématiquement l'historique papier de chaque GAB.
• Vérification que le GAB est en connexion permanente avec la société de maintenance.
• Détection et rejet d'une carte invalide (carte téléphonique..)
• Avaler une carte après trois tentatives infructueuses de saisi du code personnel
• A partir de la carte, reconnaissance du client : interne à la banque, externe ou technicien de maintenance
• Reconnaissance grâce à l'ordinateur des cartes bleues des cartes en opposition
• Annulation volontaire par le client de la transaction
• Délai d'attente entre 2 actions de l'utilisateur trop long **Application banque:**

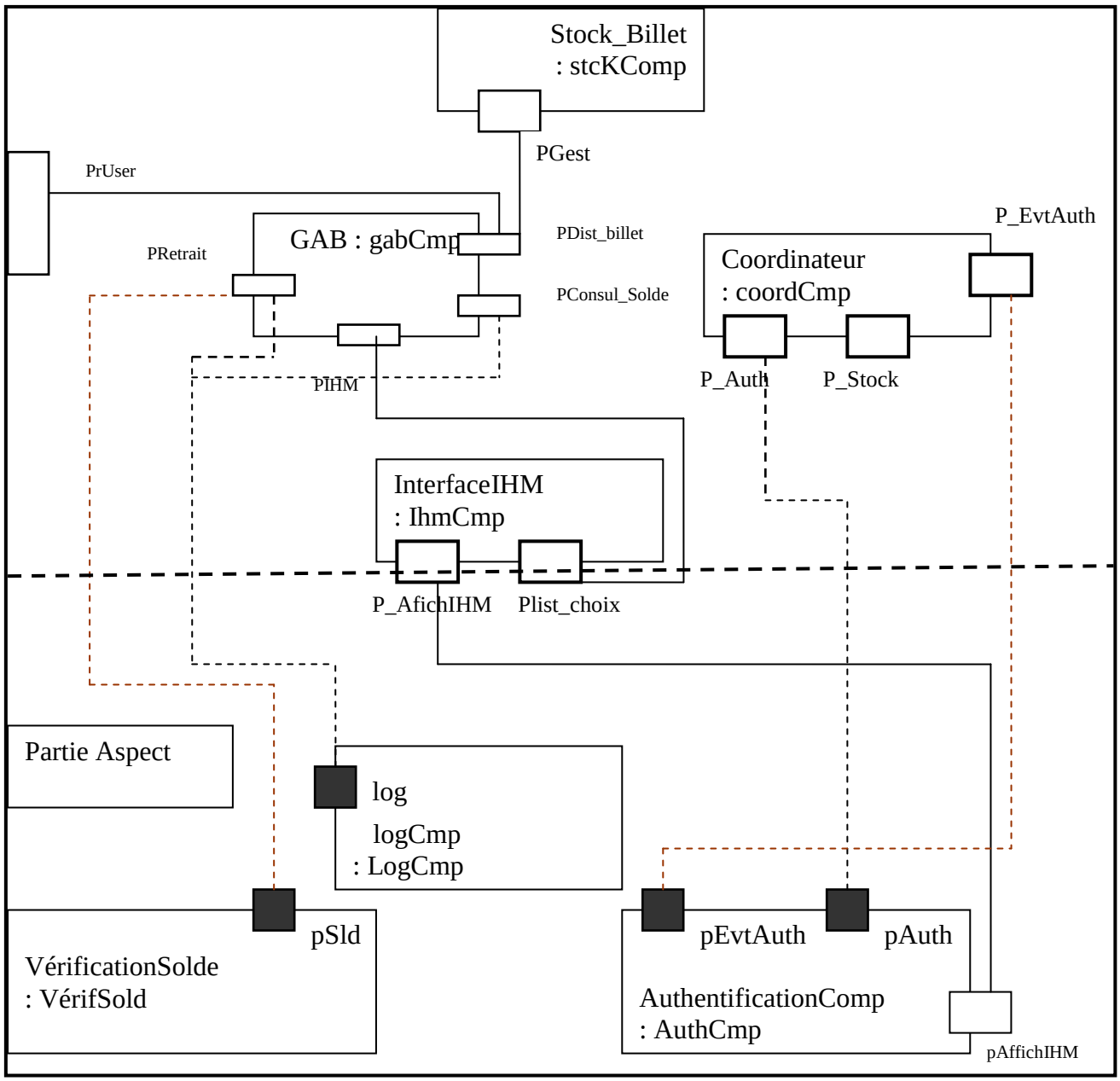



Figure 5.5: Composant Distributeur de billet

### 5.2.3.1 Specification de composant SecurityCom

```
AspectPart{
Public component class SecurityComp extends IASAAspectComponent{

Public void port pAuth{
        Provides void Authentification ();        }
Void Authentification(){
If (isAuthenticated()){ return}
String[ ] userNamePassword= getUserNamePassword();
If (!userNamePassword[0].equals(userNamePassword[1]){
Throw new serviceAuthentificationException("Utilisateur / Mot de passé incorrect");
authentificatedUser.set(UserNamePassword[0]);
}

Pointcuts{
l_AuthPtCut2 {   P_Auth. Receive; }}}
```

Figure 5.6: Composant SecurityComp

### 5.2.3.2 La spécification de composant Aspect VérificationSolde

```
AspectPart{

Public component class VérificationSolde extends IASAAspectComponent{
Private SommeRetrait;
Public void port pSld{
        Provides void VerifiSolde(int Solde);
        Requires int GetSolde (int Solde);
        Requires void Retrait (int SommeRetrait);

      }
}
SommeRetrait= GetSolde();
   Void VérifiSolde(int Solde){
If (SommeRetrait >Solde) { System.out.println("Somme insuffisante");}
Else Retrait(Solde);
}

      Pointcuts{

log_compt= pRetrai.receive;
}
}
      }// fin aspectPart
```

Figure 5.7: Composant aspect VérificationSolde

### 5.2.3.3 La specification de composant Aspect LogCmp

```
import java.util.logging.Level;
import java.util.logging.Logger;

AspectPart{

Public component class LogCmp extends IASAAspectComponent{
Private SommeRetrait;
Public void port  plog{
            Provides void log(int Solde);

          }
}
    Void log(){

try{
  FileHandler hand = new FileHandler("vk.log");
  Logger log = Logger.getLogger("log_file");
  log.addHandler(hand);
  log(Level.INFO, "Test de logger);//
  System.out.println(log.getName());
  }
  catch(IOException e){}
  }//end log

}// fin logcomp

Pointcuts{
log_Solde= {pdêpot.send};
log_compt= {pRetrai.receive};
log_Auth={ l_Solde + log_Solde+ log_compt };
log_Security{ log_Solde+ log_compt };
}
}// fin aspectPart
```

Figure 5.8: Composant aspect LogCmp

### 5.2.3.4 La specification de composant Aspect AuthentificationComp

```
AspectPart{

Public component class AuthentificationComp extends IASAAspectComponent{

Public void port  pEvtAuth{
          Requires void AuthVérification();

        }
Public void port  pAuth{
          Provides void Authentification ();
        }
}

Void Authentification(){
If (isAuthenticated()){ return}
String[ ] userNamePassword= getUserNamePassword();
If (!userNamePassword[0].equals(userNamePassword[1]){
Throw new serviceAuthentificationException("Utilisateur / Mot de passé incorrect");

authentificatedUser.set(UserNamePassword[0]);
}
Public Boolean isAuthenticated{
Return authenticatedUser.get()§=null;
}
Public string[ ] getUserNamePassword(){
Boolean userPrintln=Boolean.getBoolean("aut.run");
String[ ] userNamePassword= new String[2];
BufferedReader in=new BufferdReader(new InputStreamReader(System.in));
Try{

If(userPrintln){  System.out.println("Nom utilisateur :   " };
Else System.out.println("Nom utilisateur :   " };

userNamePassword [ 0]=in.readLine.trim;

If(userPrintln){  System.out.println("Mot de passe :   " };
Else System.out.println("Mot de passe :   " };

userNamePassword [ 1]=in.readLine.trim;

} catch (IOException e){ }

}//fin getUserNamePassword

}//end autentification

}// fin comp

Pointcuts{
l_AuthPtCut {   P_EvtAuth. Receive; }
l_AuthPtCut2 {   P_Auth. Receive; }
}
}// fin aspectPart
```

Figure 5.9. Composant Aspect AuthentificationComp

# Chapitre 6 :
# Conclusions et perspectives

L’objectif de cette thèse est de proposer un approche de conception logicielle qui vise à faciliter la construction d’architectures logicielles et à maîtriser leurs évolutions et de faire supporter les concepts orienté aspect. Nous avons pour cela proposé l'approche IASA (Integrated Approach to Software Architecture) dont l’objectif est à la fois de maîtriser la complexité liée à la construction d’une architecture logicielle et de favoriser la réutilisation. Cette approche se fait fort de respecter les grands principes du génie logiciel à savoir la modularité, l’abstraction, l’anticipation du changement, la construction incrémentale et la réutilisation.

Notre travail s’inscrit dans le contexte du projet IASA (Integrated Approach to Software Architecture) dont l’objectif premier est de mettre sur place une approche architecture logicielle qui doit supporter de manière native les concepts d’aspect. Cette approche dispose d’un langage de description d’Architecture logiciel qui doit permettre de spécifier la structure, le comportement et l’orienté aspect.

Nous avons conçu l'ADL 3ADL (Aspect, Action and Architecture Description Language) qui permet de décrire les propriétés techniques qui recoupent les composants métiers des systèmes logiciels au niveau architectural. 3ADL est une extension du modèle de composant IASA pour l’intégration de nouvelles préoccupations au sein d’une architecture logicielle qui un modèle pour la conception d'une architecture logicielle étape par étape: De l'architecture qui contient la logique métier que dans une architecture globale qui contient des préoccupations transversales et techniques en séparant les différents besoins fonctionnels et préoccupations extra-fonctionnelles.

Cette approche vient de principes Aspect Oriented Software Development (AOSD), Elle respecte les éléments de l'AOSD (coupe, point de jonction, aspect).

Notre approche permet une séparation des préoccupations durant tout les cycles de vie de logicielle de l'étape de conception logicielle revenu a l'étape d'implémentation et d'exécution de logicielle.

**La modularité:** Notre approche ne souffre pas de problème de violation de l'encapsulation en créant des dépendances entre les préoccupations. L'idée est d'ouvrir un programme a l'approche par aspect via des points d'accès ASPOAP (qui sont des points de jonction exportés par le composant aspect) toute en gardant la modularité en cachant les détails d'implémentation des composants métiers, se que permet de préserver le contenu d'un composant.

**La réutilisation:** IASA visent à permettre d'augmenter la productivité au sein de ce canevas de conception en favorisant la réutilisation des composants aspects. Les composants Aspects sont réutilisables puisqu'ils représentent exclusivement le métier de l'aspect sans prendre compte du domaine dans lequel il va être appliqué.

Le modèle est récursif : un composant est de type primitif ou composite. Dans ce dernier cas, le composant correspond à un assemblage d'autres Composants primitifs ou composites.

Un composant peut également être partagé entre différents composites.

Finalement, nous avons montré comment ce canevas de conception d'architecture logicielle pouvait être intégré dans un atelier de génie logiciel ***IASA Studio*** centré sur la notion de composant afin de favoriser la tâche de construction d'une architecture logicielle. Cet atelier de génie logiciel fournit le moyen de concevoir une architecture logicielle par assemblage de composants et tissage.

L'outil ***IASA Studio*** que nous avons développé sous Eclipse sous la plate forme windows. Cet IDE est doté d'un ensemble d'outils pour développer et spécifier une architecture logicielle:

- Graphiquement (créer des composants métiers/Aspects, ports, des connecteurs)
- Génération de code avec l'ADL 3ADL
- L'implémentation de ce code sous Archjava ou aspectj
- Exécuter le code généré (tissage d'aspect)

**PERSPETIVES:**

A terme, une bibliothèque de services techniques serait une contribution très intéressante. C'est-à-dire définir une bibliothèque de services techniques conçue comme des aspects complexes avec IASA.

Nous visons également, à moyen terme, d'intégrer notre travail sous forme de plugin ECLIPSE afin de permettre aux concepteurs des architectures logicielles d'effectuer leurs travaux en utilisant un outil graphique guidé et automatisé.

On vise aussi de régler un certain nombre de problème en particulier les conflits entre deux services techniques; Le problème de composition d'aspect est aujourd'hui encore ouvert dans la communauté de l'approche par aspect.

La gestion des aspects complexes en donnant la possibilité d'extraire toute les préoccupations transverses y compris au sein d'un aspect.

Une autre perspective consiste à continuer à travailler sur l'utilisation d'un moteur de règle pour rechercher des points de jonctions on définissant un langage de coupe et de recherche exhaustive des points de jonction compatible.

**PUBLICATION:**

Se travail à étais publié à la bibliothèque internationale ACM sous le titre 'A new approach to introduce aspects in software architecture**'.**

**CONFERENCE INTERNATIONALE:**

Se travail à étais publié à la conférence internationale ICPE 2011 (International conference on Performance Engineering ICPE 2011), qui a eu lieu à Karlsruhe, Allemagne, de 14 a 16 mars 2011.

"Khider Hadjer, Bennouar Djamal: A new approach to introduce aspects in software architecture. ICPE 2011: 419-420"

# REFERENCES


1. Davidson, E.J., “A high order Crank-Nicolson Technique for Studying Differential Equations”, Computer Journal, V. 9, n° 8, (August 1967), 195 - 197.
2. Maizus, Z.K., Emanuel, N.M. and Denisov, E.T., “Liquid Phase Oxidation of Hydrocarbons”, Plenum Press, New York, (1998), 252 p.
1. J. Baltus, ‘La Programmation Orientée Aspect et AspectJ : Présentation et Application dans un Système Distribué’, Facultés Universitaires Notre-Dame de la Paix, Belgique, Novembre 2007.
2. ‘Programmation orientée aspect’ : URL : http://fr.wikipedia.org, l'encyclopédie libre. Septembre 2007.
3. P. Smacchia, S. Vaucouleur, URL : http://www.dotnetguru.org/articles/dossiers/aop, juillet 2003.
4. L. Seinturier, R. Pawlak, L. Duchien,’ Vers un rapprochement des aspects et des composants’, INRIA Futurs (Jacquard) & LIFL (GOAL), Rennes, mars 2005
5. I. Krechetov, B. Tekinerdogan, A. Garcia, C. Chavez, U. Kulesza, ‘Towards an Integrated Aspect-Oriented: Modeling Approach for Software Architecture Design’, Computing Department, Lancaster University, UK, Décembre 2006.
6. J.Noye, R. Douence, M. Sudholt, ‘Composants et aspects’, Ecole des Mines de Nantes, France, Septembre 2004
7. J. M. Geib, P. Merle, Projet JACQUARD, ‘ Proposition de création d’un projet INRIA : Tissage de Composants Logiciels’, Lille, Septembre 2002
8. H. Fakih, N. Bouraqadi, ‘ Les aspects et les composants logiciels : Etude de cas avec le modèle de composant Fractal’, École des Mines de Douai, France, 2004
9. « Programmation orientée aspect » : URL : http://fr.wikipedia.org, l'encyclopédie libre. Septembre 2007.
10. P. Smacchia, S. Vaucouleur, URL : http://www.dotnetguru.org/articles/dossiers/aop, juillet 2003.
11. D. Talby, “Aspect-Oriented Programming”, Avril 2004
12. P. Gianni, E. De Lamarter, « la programmation orienté aspect », mai 2006.

13. G. Kiczales, “Getting started with AspectJ”, 1997.

14. S. Soares, “Implementing Distribution and Persistence Aspects with AspectJ, et Al. OOPSLA 2002”.

15. G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Videira Lopes, J.-M. Loingtier,J. Irwin, Aspect-Oriented Programming. Springer-Verlag, 1997

16. G. Kiczales, « Les critères d’un bon design AOP » , 1997.

17. V. MARANGOZOVA, « Duplication et cohérence configurables dans les applications réparties à base de composants », Université Joseph Fourier, juin 2003.

18. O. Barais « Construire et Maîtriser l’Evolution d’une Architecture Logicielle à base de Composants », Lille, novembre 2005.

19. L. Quintian, “ JADAPT: Un modèle pour améliorer la réutilisation des préoccupations dans le paradigme objet », Nice, Juillet 2004.

20. O. Hachani, D. Bardou, « Apport de la programmation par aspects dans l'implantation des patrons de conception par objets », Grenoble, Juin 2002.

21. H. KHIDER, Conception d’un IDE pour la spécification de l’architecture logicielle basé sur ArchJava, Université de SAAD D’AHLEB, Blida, Algérie, Octobre 2005.

22. G. KOPOLOUD, Model Checking for Concurrent Software Architectures, PhD thesis, Imperial College of Science, Javier 1999.

23. O. Barais, E. Cariou, L. Duchien, N. Pessemier, and L. Seinturier, “TranSAT: A Framework for the Specification of Software Architecture Evolution” ,Université des Sciences et Technologies de Lille, Novembre 2005.

24. O. Barais, L. Duchien, « TranSAT : Maîtriser l’Evolution d’une Architecture logicielle », Université des sciences de Lille, Octobre 2004.

25. O. Barais, L. Duchien, « Session commune LMO'04, Journées Composants », Lille, mars 2004

26. G. Kiczales1, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, W. G. Griswold, “An Overview of AspectJ”, University of British Columbia, USA, 2001.

27. P. COLLET, « État de l’art sur la contractualisation et la composition », Août 2006.

28. O. Barais, J. Lawall, A. F. Le Meur, L.Duchien, « Safe Integration of New Concerns in a Software Architecture », Jacquard project, INRIA/LIFL, Lille, Novembre 2005.

29. V.Quéma, L. Seinturier, J.-B. Stefani, « Intergiciel et Construction d'Applications Réparties, Chapitre 3 : Le système de composants Fractal», 2006.

30. N. Pessemier, L. Seinturier, L. Duchien, «Une extension de Fractal pour l'AOP », France

31. H. Ossher and P. Tarr. "Software Architecture and Component Technology: State of the Art in Research and Practice, chapter Multi-Dimensional Separation of Concerns and The Hyperspace Approach",Kluwer Academic Publishers, 2002.

32. M. A. Cibrán, B. Verheecke, "Modularizing Web Services Management with AOP System and Software Engineering Lab", Brussels, Europe, 2003.

33. T. Cottenier, T. Elrad, "Validation of Context-Dependent Aspect-Oriented Adaptations to Components", Illinois Institute of Technology, USA,2004.

34. A. Tesanovic, "Towards aspectual component-based real-time system development", Department of Computer and Information Science, Suède , février 2003

35. J. Noye, Remi Douence, Mario S¨udholt, "Composants et aspects", Projet Obasco EMN - INRIA Ecole des Mines de Nantes, France septembre 2004

36. K.LIEBERHERR, D.LORENZ, M.MEZINI, Programming with Aspectual Components, College of Computer Science Northeastern University, Boston, Avril 1999.

37. J. Dedecker, Dynamic Aspect Composition using Logic Metaprogramming ,Vrije Universiteit Brussel – Belgium ,Faculty of Sciences In Collaboration with Ecole des Mines de Nantes France.2001-2002

38. S. Herrmann, M. Mezini, "Combining Composition Styles in the Evolvable Language LAC" ,Technical University Berlin, Germany, 2001.

39. K. LIEBERHERR, D. LORENZ, M.MEZINI ,"Programming with Aspectual Components", Germany,April, 1999

40. F. Loiret, L. Seinturier, E. Gressier,'Fractal, Kilim, JAC: une expérience comparative', 2004

41. D. Bennouar, T. Khammaci, and A. Henni :'Modeling the Component's Interaction Point in The IASA approach', The Mediterranean Journal of Computers and Networks, vol. 4, no. 4, 2008, pp 188-197

42. D. Bennouar: 'The Aspect Oriented Software Architecture in the IASA Approach', 2008, laboratoire LRDSI, Université Saad Dahlab, Blida, Algérie.

43. D. Bennouar, T. Khammaci, A. Henni: 'L'approche IASA d'architecture Logicielle : Modélisation de la vue externe des composants', 2008.

44. « L'Architecture logicielle » : URL : http://fr.wikipedia.org, l'encyclopédie libre. Septembre 2007.

45. R. Sanlaville, 'Architecture logicielle : Une expérimentation industrielle avec Dassault systèmes' mai 2002.

46. N. Manh Tien, 'Programmation Orientée Aspect', Juillet 2005.

47. G. Dufrêne, S. Morvan, L. Duchien' Rapport de Conception d'Applications Réparties' janvier 2006.

48. O. Barais, Ph. Lahire, A. Muller, N.Plouzeau, G. anwormhoudt, 'Évaluation de l'apport des aspects, des sujets et des vues pour la composition et la réutilisation des modèles', mai 2010.

49. S. KEBIR,' Programmation orientée aspect en Java avec AspectJ', février 2010

50. E. Wawszczyk :'introduction à AOP (Aspect-OrientedProgramming) avec le framework Spring', Décembre 2007.

51. S. Hostettler : 'Spring : théorie & pratique', Mars 2007

52. N.Loughran, A.Rashid : 'Framed Aspects: Supporting Variability and Configurability for AOP', Computing Department, Lancaster University UK, juin 2004

53. S. Traumat, 'Tutorial Spring AOP', URL : http://WWW.scub.foundation.org, Avril 2010

54. F. Bodmer, T. Maret, 'AOP Tools Comparison', Universitéde Fribourg, Juin 2006

55. J.dubois, J.P. Retaillé, T.Templier, 'Spring par la pratique : Mieux developper ses applications Java/J2EE avec Spring, Hibernate, Struts', Paris, 2007.

56. R. Laddad, 'Enterprise AOP with Spring Applications', Edition MANNING 2010.

57. R. Johnson, J. Hoeller, A. Arendsen, 'Spring : java/J2EE Application FrameWork', 2008

58. N. Pessemier 'Thèse de doctorat : Unification des approches par aspects et à composants', universit▪ des Sciences et Technologies de Lille, France Juin 2007.

59. C. HERAULT,' thèse : Adaptabilit▪ des Services Techniques dans le Model Composants ', l'Université de Valenciennes et du Hainaut Cambrésis, France février 2006

60. R. Delamare, ‘Thèse de doctorat : Analyses automatiques pour le test de programmes orientés aspect’, Institut de Recherche en Informatique et Systèmes Aléatoires, France décembre 2009.

61. R. Pawlak, J. Ph. Retaillé, L. Seinturier,”Programmation orientée aspect pour Java/J2EE” , Eyrolles, 2004.
62. M. TRANCHANT, ‘ Java WebServer Tomcat, JBoss, JRun, JOnAS ‘ Décembre 2008.

63. B. Wang, B. Ban,’ JBoss Enterprise Application Platform 5.0 Administrations and Configuration Guide’, Red Hat 2009.

64. R. Dubourguais, ‘JBoss Application Server : exploitation et Sécurisation’, Conférence sur les web Shell 2010.

65. L. SEINTURIER,’Réflexivité, aspects et composants pour l'ingénierie des intergiciels et des applications réparties’, l’universit‧ pierre et marie curie, Paris decembre 2005.

66. L. Seinturier, ‘JAC (Java Aspect Components) Frameworkde programmation par aspects’, Projet JACQUARD, janvier 2005.

67. L. Fabrice, ’Sujet de la thèse : Un modele d'assemblage de composants par contrat et Programmation Orientée Aspect’, conservatoire national des arts et métiers, juillet 2005.

68. M. Hariati, D. Meslati, ‘Les Composants Logiciels et La séparation Avancée des Préoccupations : Vers une nouvelle Approche de Combinaison’, Volume 11 - Pages 53 ‧ 68 – ARIMA, CARI 2008.

69. J. Aldrich, Open Modules: Modular Reasoning about Advice, Carnegie Mellon University, Pittsburgh, PA 15213, USA 2004.

70. Thomas Cottenier, Tzilla Elrad ,'Validation of Context-Dependent Aspect-Oriented Adaptations to Components',Concurrent Programming Research Group Illinois Institute of Technology, USA 2005

71. K.Sullivan, W. G. Griswold, Y. Song, Y. Cai, M. Shonle, N. Tewari, H. Rajan,' Information Hiding Interfaces for Aspect Oriented Design',University of Virginia

72. K. Aljasser, P. Schachte, 'ParaAJ: toward Reusable and Maintainable Aspect Oriented Programs',The University of Melbourne, Australia

73. J. Aldrich, ’Open Modules: Reconciling Extensibility and Information hiding’, School of Computer Science Carnegie Mellon University USA.

74. M. Mezini, Klaus Ostermann,’ Conquering Aspects with Caesar’ Darmstadt University of Technology Darmstadt, Germany.

75. M. Mezini and K.Ostermann. Conquering aspects with caesar. In M. Aksit ed., editor, Proceedings of the 2nd International Conference on Aspect-Oriented Software Development (AOSD), pages 90–100, Boston, 2003. USA. ACM Press

76. K. LIEBERHERR, D. LORENZ, M. MEZINI:' Programming with Aspectual Components', College of Computer Science. April 1, 1999

77. E. Truyen, W. Joosen: On the Criteria of Aspectual Component Models Steven Op de beeck, Johan Gr馮oire,Department of Computer Sciencen Leuven, Belgium , March 20–21, 2006, Bonn, Germany.

78. D. Suvée, W. VANDERPERREN, B. Fraine, V. Jonckers, 'Aspect-Oriented Programming using JasCo', System and Software Engineering Lab, Brussels Belgium 2004, p.1-36.

79. D. Suvée, W. VANDERPERREN, V. JONCKERS,'JAsCo : an aspect-oriented approach tailored for component based software development', Proceedings of the 2nd InternationalConference on Aspect-Oriented Software Development (AOSD'03), ACM Press,2003, p. 21–29.

80. W. Vanderperren, D. Suvée, M. A. Cibran, B.Fraine ," Stateful Aspects in JAsCo",Vrije Universiteit Brussel, Pleinlaan 2, Brussels, Belgium, pp. 167–181, 2005

81. Mehdi Hariati,Djamel Meslatin ,'Les Composants Logiciels et La Séparation Avancée des Préoccupations : Vers une nouvelle Approche de Combinaison', Laboratoire LRI, Université Badji Mokhtar-Annaba BP 12, 23000 AnnabaALGERIE. 2008 Pages 53- 68 .

82. K.Viggers, J. Walker, 'An Implementation of Declarative Event Patterns',Department of Computer Science University of Calgary Calgary, Alberta, Canada Decembre 2004

83. K. Viggers, "Improving the Modularity of Context-Sensitive Concerns through the Use of Declarative Event Patterns", , ALBERTA, Canada September, 2005

84. Pichler, R., Ostermann, K., Mezini, M.: On aspectualizing component models. Software Practice and Experience Version: 2002/09/23 v2.2

85. C.Bockisch, M.Haupt, M. Mira, K.Ostermann 'Virtual Machine Support for Dynamic Join Points' Darmstadt University of Technology, Germany mars 2004

86. J. Brichau, M. Haupt," Survey of Aspect-oriented Languages and Execution Models", May 2005.

87. J. Aldrich, "Open Modules: Reconciling Extensibility and Information Hiding", School of Computer Science Carnegie Mellon University, USA 2005.

88. T.Batista, C. Chavez, A. Garcia, U. Kulesza, C. Sant'Anna, C. Lucena, "Aspectual Connectors: Supporting the Seamless Integration of Aspects and ADLs ", 1Computer Science Department, UFRN – Brazil.2006.

89. A. Navasa, A. Miguel. J. M. Murillo, 'AspectLEDA: Extending an ADL with Aspectual Concepts ', Department of Computer Science, University of Extremadura. Spain First European Conference, ECSA 2007 Madrid, Spain, September 24-26, 2007 Proceedings p331- 334

90. J. Benedí,"PRISMA: Aspect-Oriented Software Architectures, Department of Information Systems and Computation Polytechnic University of Valencia, 2006

91. W.Jing,Y, N. YouCong, 'AC2-ADL: Architectural Description of Aspect-Oriented Systems ', International Journal of Software Engineering and Its Applications Vol. 3, No. 1, January, 2009

92. M.Pinto, L. Fuentes. Ao-adl 'An adl for describing aspect-oriented architectures', pages 94-114. 2007.

93. S.LOUKIL, 'Extension d'un langage de description d'architecture pour la programmation orienté aspect', Universit▪ de Sfax école Nationale d'Ingénieurs de Sfax, Tunisie, juillet 2010.

94. H. Khider, D. Bennouar, 'A new approach to introduce aspects in software architecture', ICPE 2011: p419-p420 Karlsruhe, Allemagne, mars 2011.