UNIVERSITE SAAD DAHLAB DE BLIDA

Faculté des Sciences

Département d'Informatique

MEMOIRE DE MASTER

Spécialité : Génie des Systèmes Informatiques

UN FRAMEWORK POUR LA PROGRAMMATION ORIENTE COMPOSANT ET ASPECT EN JAVA

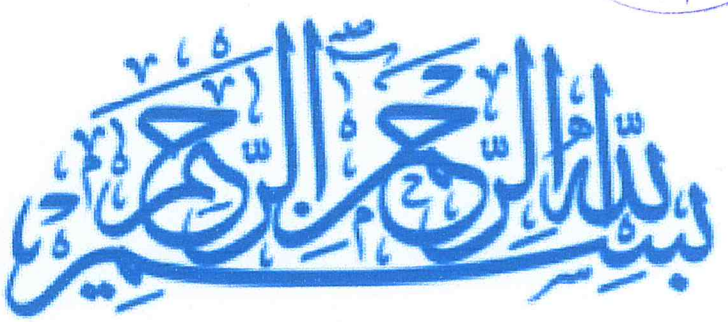
Par

Batoul HOCINE

Devant le jury composé de :

M. SIDOUMOU	Université Saad Dahlab de Blida	Président
D. GUESSOUM	Université Saad Dahlab de Blida	Examinatrice
A. BAOUYA	Université Saad Dahlab de Blida	Examinateur
D. BENNOUAR	Université Saad Dahlab de Blida	Promoteur

Blida, Juin 2013


بسم الله الرحمن الرحيم

Résumé:

Ce travail s'inscrit dans le cadre des architectures logicielles à base de composants. La définition d'une architecture logicielle se fait à l'aide de Langages Description Architecture (ADL). Ces derniers permettent de supporter des raisonnements indépendants du niveau implémentation représenté souvent soit par des langages de programmation ou des framework sous forme d'API.

Notre travail s'intéresse à la proposition d'un Framework, appelé AOCF (Aspect Oriented Component Framework) permettant de supporter directement les concepts fondamentaux de l'architecture logicielle tels que les concepts de composants, ports, connexion et composant composite. Un tel framework permettrait la programmation orienté composant et aspect et pourrait représenter une cible efficace dans un processus de transformation d'une spécification en ADL en une spécification de niveau implémentation.

AOCF unifie les approches à composants et par aspects, il considère les aspects et les composants comme des entités uniformes (Symétrie d'élément) avec une séparation explicite de la définition des points de coupe et le traitement des aspects ce qui permet un haut niveau de réutilisation pour les aspects comme pour les composants. AOCF permet un développement rapide des systèmes basé sur des composants certifiés et dont les préoccupations non fonctionnelles sont factorisées en dehors de ces composants, AOCF permet d'avoir des systèmes évolutifs, adaptatifs et maintenables.

Mots-clés: Architecture Logicielle, LDA, POC, POA, Composant, Connecteur, Port, point de jonction, point de coupure.

Abstract:

This work is part of the field of software architecture based on components, the definition of software architecture is generally described by an ADL (Architecture Description language). These ADLs allows supporting the reasoning's independent of the implementation level represented by programming language or framework in the form of API.

We are interested in this work to propose a Framework called AOCF (Aspect Oriented Component Framework), which allows supporting the basic concepts of software architecture such as: Components, ports, connection and composite component. Such as this Framework enables component and aspect oriented programming and it will be also represented as an effective target in the process of transforming an ADL specification into implementation level.

AOCF unifies the two approaches of components and aspects; it considers aspects and components as uniform entities (Symmetry element) with an explicit separation of the definition of points cut and the code of aspects which enables a high level of reuse for aspects like components. It allows rapid development based on certified components and systems including non-functional concerns are factored outside these components, AOCF allows for scalable systems, adaptive and maintainable.

Key words: Software architecture, ADL, COP, AOP, Component, Connector, Port, join point, point cut.

Remerciements

Je tiens tout d'abord à remercier Dieu le tout puisant et miséricordieux, qui m'a donné la force et la patience d'accomplir ce modeste travail.

La première personne que je tiens à remercier est mon Promoteur Dr. BENNOUAR Djamal, pour l'orientation, la confiance, la patience qui ont constitué un apport considérable sans lequel ce travail n'aurait pas pu être mené au bon port. Qu'il trouve dans ce travail un hommage vivant à sa haute personnalité.

Mes vifs remerciements vont également aux membres du jury pour l'intérêt qu'ils ont porté à notre recherche en acceptant d'examiner notre travail Et de l'enrichir par leurs propositions.

Enfin, je tiens également à remercier l'ensemble de la famille enseignante de l'université SAAD DAHLEB De Blida et toutes les personnes qui ont participé de près ou de loin à la réalisation de ce travail.

Dédicaces

A mon père qui sait sacrifier afin que rien ne m'empêche du bon. Ce travail est le fruit de tes sacrifices que tu as consentis pour mon éducation. Puisse Dieu, le tout puissant, te préserver et t'accorder santé, longue et bonheur.

A ma très chère mère: Tu es l'exemple de dévouement qui n'a pas cessé de m'encourager et de prier pour moi. Puisse Dieu, le tout puissant, te préserver et t'accorder santé, longue et bonheur.

A tous mes sœurs mariées, ces maris et ces enfants: Mes meilleurs vœux de succès dans votre vie.

A mes sœurs Aicha et Noussiba et mon frère Oussama: meilleurs vœux de succès dans vos études.

A toute ma famille

A tous mes amies: Khadidja, Nadjah, Sabrina, Sarah, Linda, Sihem, Nawal, Khawla et Hassiba: je vous remercie de votre patience vous m'a aidée toujours à avancer vous êtes tous des grandes amies si gentilles, merci d'être toujours prés de moi, amies avec lesquels je souris.

A mes enseignats de l'ecole primaire jusqu'à l'université dont les conseils précieux m'a guidée; qu'ils trouvent ici l'expression de ma reconnaissance.

Table des matières

Chapitre I: Généralité

Chapitre II: Concepts fondamentaux

Chapitre III: Approches à composant au niveau implémentation

Chapitre IV: Concepts fondamentaux de l'AOCF

Chapitre V: Implémentation et tests

Chapitre VI: Conclusion et perspectives

VI.1. Conclusion général

VI.2. Perspectives

Annexes

Bibliographies

Liste des figures et tableaux

Liste des tableaux

Chapitre I:
Généralités

Chapitre I: Généralité

I.1. Introduction générale :

Les logiciels, et les systèmes informatiques en général, croissent en nombre et en complexité dans des domaines d'application divers. Cette complexité croissante est caractérisée par l'augmentation de la taille des logiciels, l'hétérogénéité des plateformes à tout niveau (systèmes d'exploitation, inter-logiciels et applications) et la multiplication des normes et des standards.

Face à cette complexité, il est nécessaire d'avoir un niveau d'abstraction élevé et de disposer de modèles qui s'approchent du modèle mental du développeur. Une réponse possible est la définition d'une architecture du système. Une architecture logicielle à base de composants décrit l'ensemble des composants qui la composent, donne la définition de leur assemblage et prend en compte les structures d'accueil nécessaires pour le déploiement et l'exploitation du système résultant. La définition d'une architecture logicielle se fait en générale par des LDA (Langages Description Architecture).

Le développement de logiciel se fait souvent par raffinement successif, à partir d'une spécification très abstraite, pouvant être représentée par une spécification dans un langage formel, en aboutissant à une étape concrète représentée par une spécification à base de composants dits primitifs. Ces derniers sont réalisés soit à l'aide d'un langage de programmation tel Java ou à l'aide d'un ADL de niveau implémentation permettant la programmation orienté composant tel qu'ArchJava **[JCD, 2002a]** **[JCD, 2002b]** ou Fractal **[ETMVJ, 2004]**.

En pratique, les ADLs permettent un raisonnement à un haut niveau d'abstraction. La plupart n'offre pas d'alternative pour transformer la description ADL en une description de niveau implémentation ou ne sont pas couplé à un langage de programmation orienté composant.

I.2. Problématique :

Les ADL sont surtout de haut niveau et nécessite à un moment donnée leur transformation dans une technologie d'implémentation (langage de programmation, utilisation de pattern etc..). Différents langages de description d'architecture, styles et modèles à composants ont émergé au cours de la dernière décennie, au même temps, il y a eu relativement peu de techniques et technologies pour transformer ces architectures logicielles vers une technologie d'implémentation. Pour certains ADL

cette transformation n'existe pas. Ces ADL sont surtout utilisés pour étudier les caractéristiques d'un système au niveau Architecture. Pour d'autre elle est spécifique à un langage de programmation bien particulier et est assurée souvent de manière manuelle.

La difficulté de transformation réside dans le fait qu'au niveau implémentation, nous ne retrouvons pas les concepts du haut niveau (composant connecteur). Malgré la présence au niveau programmation d'un langage orienté composant, en l'occurrence ArchJava, en pratique ce langage ne semble pas avoir été mis en œuvre dans un processus de transformation. Ceci pourrait être principalement du au fait que les connecteurs en ArchJava sont implicites ou nécessitent vraiment une conception a part difficilement réalisable par des personnes non experte en Java et en technologie de communication basée sur Java.

I.3. Objectifs :

L'objectif de ce travail s'inscrit dans une action de recherche a travers laquelle il faut déterminer si l'approche Framework pour l'architecture logicielle, pourrait être assez efficace comme cible d'un processus pour l'implémentation des concepts fondamentaux d'une architecture logicielle à base des composants. Ainsi l'objectif du travail consiste à faire des investigations pour mettre au point un framework orienté conception par composant d'application. Ce framework devra offrir les fonctionnalités pour permettre de spécifier et définir un composant et ses ports, connecter les composants, assembler des applications distantes se trouvant sur des machines différentes et tisser des composants spéciaux (aspects).

Notre objectif est de rapprocher l'approche par composants et l'approche par aspect ç.-à.-d de considéré les composants aspects et les composants métiers comme des entités uniforme, ce qui permet une haute réutilisabilité des aspects.

I.4. Organisation du document :

Le reste du document est organisé en cinq chapitres, qui sont organisés comme suit :

- ***Chapitre II*** : Concepts fondamentaux : nous allons présenter dans un premier temps le domaine Architecture logicielle et les concepts fondamentaux liées à

ce domaine. Nous allons aussi décrit l'approche COP (Component Oriented programming) et l'approche AOP (Aspect Oriented programming).

- ***Chapitre III***: " *Approches à composants au niveau implémentation* **":** Dans ce chapitre nous allons présenter quelques approches liées aux composants au niveau implémentation avec une évaluation de chaque un selon des critères précités qui nous semble important pour un ADL de niveau implémentation.
- ***Chapitre IV :*** *"Concepts fondamentaux de AOCF"* Dans ce chapitre nous allons décrire notre proposition appelée AOCF (Aspect-Oriented Component Framework). AOCF est d'une part d'être une cible de niveau implémentation pour des spécifications en ADL d'architectures et d'autre part permet la programmation orientée composant et aspect.
- ***Chapitre V***: *"Implémentation et test"*, Ce chapitre décrit le détail d'implémentation de notre approche, avec quelques tests pour valider notre réalisation.
- ***Chapitre VI :*** *"Conclusion Général"*, Ce manuscrit se termine par une conclusion générale et de perspectives des travaux en cours.

Introduction :

L'architecture logicielle [Garlan, 1994] est devenue une activité de conception explicite dans le processus de développement. Elle modularise un système sous forme d'une configuration de composants et de connecteurs.

L'architecture logicielle permet d'identifier des entités de connexion appelées connecteurs et de décrire de manière précise les protocoles de communication entre composants [Allen, 1996] [Allen, 1997a]. Les travaux sur les formalismes de description d'architectures logicielles ont donné naissance à plusieurs ADL (Architecture Description Langage) [Medvidovic, 2000].

Dans ce chapitre nous allons présenter les concepts fondamentaux nécessaire pour notre recherche à savoir le domaine d'architecture logicielle et ces concepts de base (la programmation par composant, la programmation par aspect, les ADLs ... ect.) nous a-llons aussi détailler sur l'approche de réutilisation du logiciel « les frameworks » qui représentent l'objectif de notre travail dans ce projet.

II.1. Introduction à l'architecture logicielle à base de composants :

Dans cette section, nous allons présenter une introduction à l'architecture logicielle, les principaux concepts et terminologies, ainsi qu'un aperçu des principaux avantages de son utilisation dans le développement logiciel. Nous donnons ensuite un éventail des approches existantes pour décrire une architecture logicielle.

II.1.1. Les concepts de base de l'architecture logicielle:

Dans cette partie, nous définissons l'ensemble des concepts que nous retrouvons dans la spécification d'architecture logicielle. Ces concepts sont au nombre de trois. Le premier est le composant, le second le connecteur, et finalement le dernier la configuration ou encore la topologie. ***[ACCORD, 2002]***

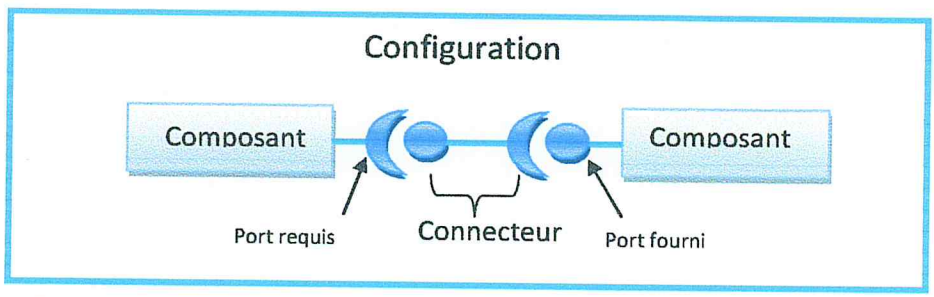


Figure II.1: Les concepts de base de l'architecture logicielle

II.1.1.1. Composants :

A l'heure actuelle, il n'existe pas de normalisation de la notion de composant. Généralement un composant réutilisable est défini comme « *Une unité de conception (de n'importe quel niveau d'abstraction) identifiée par un nom, avec une structure définie et des directives de conception sous la forme de documentation pour supporter sa réutilisation* ». ***[Per et al, 2000]***

La documentation d'un composant illustre le contexte dans lequel il peut être utilisé en spécifiant les contraintes et les autres composants dont il a besoin pour offrir sa solution.

Nous adoptons cette définition générale du concept de composant car elle admet la possibilité d'utiliser le concept de composant dans des niveaux d'abstraction autres que le niveau d'implantation.

Un composant peut être ***primitif*** ou ***composite***. Un composite est un réseau de composants interconnectés selon une topologie permettant d'atteindre les objectifs fonctionnels du composite.

Les caractéristiques globales d'un composant définies par Medvidovic et Taylor ***[Reiko,2004]*** sont les suivantes :

- l'interface,
- le type,
- la sémantique,
- les contraintes,
- les propriétés non fonctionnelles.

➢ **L'interface d'un composant**

L'interface d'un composant est la description de l'ensemble des services offerts et requis par le composant sous la forme de signature de méthodes, de type d'objets envoyés et retournés, d'exceptions et de contexte d'exécution. L'interface est un moyen d'expression des liens du composant ainsi que ses contraintes avec l'extérieur. L'interface peut englober une description comportementale, souvent référencée par le terme typage comportemental. Le typage comportemental indique les règles de mise en œuvre d'une interface.

➢ **Le type d'un composant**

Le type d'un composant est un concept représentant l'implantation des fonctionnalités fournies par le composant. Il s'apparente à la notion de classe que l'on trouve dans le modèle orienté objet. Ainsi, un type de composant permet la réutilisation d'instances de même fonctionnalité soit dans une même architecture, soit dans des architectures différentes. En fournissant un moyen de décrire, de manière explicite, les propriétés communes à un ensemble d'instances d'un même composant, la notion de type de composant introduit un classificateur qui favorise la compréhension d'une architecture et sa conception.

➢ **La sémantique d'un composant**

La sémantique du composant est exprimée en partie par son interface. Cependant, l'interface telle que décrite ci-dessus ne permet pas de préciser complètement le comportement du composant. La sémantique doit être enrichie par un modèle plus complet et plus abstrait permettant de spécifier les aspects dynamiques ainsi que les contraintes liées à l'architecture. Ce modèle doit garantir une projection cohérente de la spécification abstraite de l'architecture vers la description de son implantation avec différents niveaux de raffinements.

➢ **Les contraintes d'un composant**

Les contraintes définissent les limites d'utilisation d'un composant et ses dépendances intra composants. Une contrainte est une propriété devant être obligatoirement vérifiée sur un système ou une de ces parties. Si celle-ci est violée, le système est considéré comme un système incohérent et inacceptable. Elles permettent ainsi de décrire de manière explicite les dépendances des parties internes d'un composant comme la spécification de la synchronisation entre composants d'une même application (dépendance intra composant).

➢ **Les propriétés non fonctionnelles d'un composant**

Les propriétés non fonctionnelles (propriétés liées à la sécurité, la performance, la portabilité, etc.) doivent être exprimées à part, permettant ainsi une séparation dans la spécification du composant des aspects fonctionnels (aspects métiers de l'application) et des aspects non fonctionnels ou techniques (aspects transactionnel, de cryptographie, de qualité de service).

II.1.1.2. Connecteur:

Les connecteurs représentent les interactions entre les composants et les règles qui régissent ces interactions correspondent aux lignes dans les descriptions de type "boites-et-lignes". Ce sont des entités architecturales qui lient des composants ensemble et agissent en tant que médiateurs entre elles. Les exemples de connecteurs incluent des formes simples d'interaction, comme des pipes, des appels de procédure, et l'émission d'événements. Les connecteurs peuvent également représenter des interactions complexes comme un protocole Remote Procedure Call (RPC) de client/serveur ou un lien de SQL entre une base de données et une application contrairement aux composants. ***[Adel, 2011]***

Un connecteur peut être un connecteur ***d'assemblage*** ou de ***délégation***. *Un connecteur d'assemblage* permet de connecter un composant qui fournit des services à un composant qui les utilise, on peut en déduire une relation *d'utilisation*. Tandis que *le connecteur de délégation* permet de déléguer la réalisation ou le requiert d'un service à un de ses sous composants, on en déduit une relation de *composition* entre les participants.

Six caractéristiques importantes définies par Medvidovic et Taylor ***[Reiko,2004]*** sont à prendre en compte pour spécifier de manière exhaustive un connecteur. Ces caractéristiques sont les suivantes :

- l'interface,
- le type,
- la sémantique,
- les contraintes,
- l'évolution,
- les propriétés non fonctionnelles.

➢ **L'interface :**

L'interface d'un connecteur, appelée aussi rôles dans certains langages de description d'architecture tel que Wright *[Sylvain,2010]*, définit les points de connexions entre connecteurs et composants. Elles servent à déclarer les participants à l'interaction décrite par le connecteur. Comme celles des composants. Néanmoins, à la différence des composants, les interfaces ne décrivent pas de services fonctionnels mais des mécanismes de connexion. Elles décrivent également le rôle de chacun des composants impliqués.

➢ **Le type** :

Le type d'un connecteur correspond à sa définition abstraite qui reprend les mécanismes de communication entre composants. Il permet la description d'interactions simples ou complexes de manière générique et offre ainsi des possibilités de réutilisation de protocoles. Par exemple, la spécification d'un connecteur de type RPC qui relie deux composants définit les règles du protocole RPC.

➢ **La sémantique**

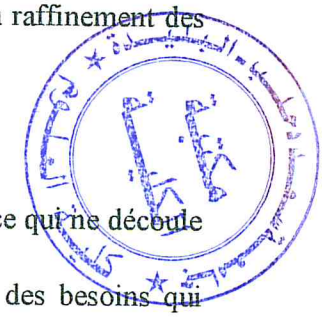
Comme pour les composants, la sémantique des connecteurs est définie par un modèle de haut niveau spécifiant le comportement du connecteur. A l'opposé de la sémantique du composant qui doit exprimer les fonctionnalités déduites des buts ou des besoins de l'application, la sémantique du connecteur doit spécifier le protocole d'interaction. De plus, celui-ci doit pouvoir être modélisé et raffiné lors du passage d'un niveau de description abstraite à un niveau d'implantation.

➢ **Les contraintes**

Les contraintes permettent de définir les limites d'utilisation d'un connecteur, c'est-à dire les limites d'utilisation du protocole de communication associé. Une contrainte est une propriété devant être vérifiée sur un système ou sur l'une de ses parties. Si celle-ci est violée, le système est considéré comme un système incohérent et inacceptable. Par exemple, le nombre maximum de composants interconnectés à travers le connecteur peut être fixé et correspond alors à une contrainte.

➢ **L'évolution d'un connecteur**

Le changement des propriétés (interface, comportement) d'un connecteur doit pouvoir évoluer sans perturber son utilisation et son intégration dans les applications existantes. Il s'agit de maximiser la réutilisation par modification ou raffinement des connecteurs existants.



➢ **Les propriétés non fonctionnelles**

Les propriétés non fonctionnelles d'un connecteur concernent tout ce qui ne découle pas directement de la sémantique du connecteur. Elles spécifient des besoins qui viennent s'ajouter à ceux déjà existants et qui favorisent une implantation correcte du connecteur. Par exemple, elles peuvent concerner la performance ou la sécurité. La spécification de ces propriétés est importante puisqu'elle permet de simuler le comportement à l'exécution, l'analyse, la définition de contraintes et la sélection des connecteurs.

II.1.1.3. Configuration:

Les configurations architecturales représentent les graphes de composants et de connecteurs et la façon dont ils sont reliés entre eux. Cette notion est nécessaire pour déterminer si les composants sont bien reliés, si leurs interfaces s'accordent, si les connecteurs correspondants permettent une communication correcte. La combinaison de leurs sémantiques aboutit au comportement désiré, qui vient en appui des modèles de composants et de connecteurs. Les descriptions des configurations permettent l'évaluation des aspects distribués et concurrents d'une architecture, comme par exemple, la possibilité de déterminer des verrous, de connaître le potentiel de performance, de fiabilité, de sécurité... etc. Le rôle clé des configurations est de faciliter la communication entre les différents intervenants dans le développement d'un système. Leur but est d'abstraire les détails des différents composants et connecteurs. ***[Adel, 2011]***

Ainsi, elles décrivent le système à un haut niveau d'abstraction qui peut être potentiellement compris par des personnes avec différents niveaux d'expertise et de connaissances techniques.

II.1.2. Les modèles à composant et leurs implantations:

La classification des modèles à composants est dictée par la modélisation trois tiers des systèmes d'information. Ainsi on distingue couramment deux grands types de modèles à composants : les modèles à composants de présentation (ex : JavaBeans) et les modèles à composants métiers (ex : EJB, CCM, .NET), la partie donnée étant gérée par les services techniques des composants métiers. Cependant, un troisième type de composants est récemment apparu : les modèles à composants qu'on pourrait dénommer "génériques" (ex : Fractal, ArcticBeans, Avalon).

II.1.2.1. Composants de présentation :

Les modèles de composants de présentation sont principalement dédiés au développement rapide d'interfaces graphiques et n'implantent pas les notions de conteneur. Le représentant principal de ce type de modèle est le modèle des JavaBeans [Medvidovic, 2003]: principalement dédiés aux applets et aux applications côté client en Java, ils sont connectés dynamiquement sur un modèle d'émetteur/récepteur d'événements et garantissent les propriétés de persistance, d'introspection et de configuration. Ces composants sont développés selon un moule assez restrictif mais qui permet de les manipuler facilement avec des outils visuels.

II.1.2.2 Composants métiers :

Les composants métiers sont des composants logiciels gros grain, destinés aux systèmes d'information distribués. Afin d'en faciliter le développement, la distribution et la réutilisation de ces composants, leur conception sont basées entre autres sur le principe de séparation code applicatif (ou fonctionnel) du code technique (ou non-fonctionnel). Cela se traduit par le fait que lors de l'exécution d'une méthode d'un composant, des services techniques tels que la distribution (ORB pour les communications synchrones ou bus à message pour les communications asynchrones), la persistance, la sécurité et d'autres sont mis en œuvre par l'intermédiaire du conteneur de façon transparente. Ce code technique étant développé à part du code applicatif et étant manipulé par le conteneur, l'application écrite par le développeur ne contient pas ou peu de code relatif à ses services.

On distingue principalement trois modèles à composants métiers les plus utilisés dans l'industrie : EJB, CCM et .NET. Chacun de ces modèles correspond à une vision particulière de la programmation : les composants Corba Component Model (CCM) de l'OMG sont destinés au développement d'applications à grande échelle et fiables pour les entreprises. Les composants Entreprise JavaBeans (EJB) de Sun proposent

une solution pour un développement rapide d'applications déployées sur Internet. Quant aux composants .NET de Microsoft, ils permettent de développer des applications "de bureau".

II.1.2.3. Composants génériques :

Un nouveau type de modèle à composants émerge. On peut qualifier les modèles de ce type de modèles à composants génériques car ils ont pour but initial de fournir des composants pour la conception et le développement à la fois d'intergiciels, d'applications d'entreprise et de services web. Des modèles tels que Fractal **[ETMVJ, 2004]** résultent du retour d'expérience des premiers modèles à composants industriels. Ainsi, ces modèles fournissent des composants à plusieurs niveaux de complexité, plus évolutifs grâce une grande flexibilité du conteneur. De plus, ces modèles introduisent la notion de composants hiérarchiques : composants qui se composent eux mêmes d'autres composants. Par extension, on peut concevoir des composants applicatifs métiers, utilisant des composants techniques plutôt légers.

II.1.3. Les langages de description d'architecture logicielle:

Plusieurs outils existant pour décrire la description d'une architecture logicielle parmi ces outils nous avons les ADL (Architecture Description Langage), la grande majorité de ces ADL se composent de trois idées principales :

- les composants,
- leurs connecteurs,
- et la configuration de l'architecture.

Les ADLs fournissent des notations formelles pour décomposer un système en composants et connecteurs (i.e., des mécanismes d'interaction), et pour spécifier comment ces éléments sont combinés afin de former une configuration ***[Reiko, 2004]***

II.1.3.1. Fractal :

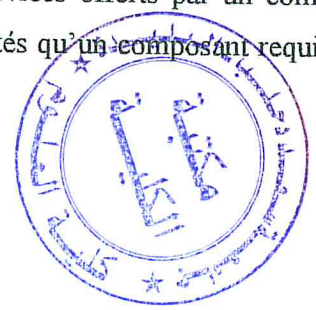
Fractal **[ETMVJ, 2004]** c'est un ADL Basé sur XML, ce langage définit une syntaxe abstraite indépendante de tout langage de programmation pour la description de l'architecture en termes de composants, d'interfaces, de liaisons et d'attributs. Fractal est un ADL extensible. Il permet d'intégrer facilement de nouveaux concepts au niveau de la description de l'architecture en modifiant la grammaire du langage. Cette grammaire est en effet construite à partir d'un ensemble de modules prenant en

charge chacun une caractéristique du modèle de composant comme les interfaces, les liaisons ou les attributs. Il est possible de définir son propre module pour ces propres caractéristiques ou simplement redéfinir des caractéristiques existantes pour définir la syntaxe de son langage de description d'architecture. On retrouve au niveau du modèle concret de composant Fractal les concepts de composant, connecteur et configuration.

- **Le composant :**

  Les composants possèdent une membrane qui délimite clairement leur contenu de leur environnement extérieur afin de structurer l'application. Cette membrane dispose d'interfaces externes (resp. internes) permettant la communication des composants avec l'extérieur (resp. au sein de leur contenu). Leur contenu consiste en un ensemble fini de sous-composants. Le modèle est donc hiérarchique permet ainsi une construction récursive qui s'arrête aux composants primitifs qui sont directement programmés dans un langage de programmation donné (Figure I.2).

Les interfaces jouent un rôle central dans Fractal. Elles appartiennent à deux catégories distinctes : les interfaces métier et les interfaces de contrôle (la membrane correspond à l'ensemble de ces interfaces de contrôle). L'ensemble des interfaces métier et de contrôle d'un composant définit son type. Les interfaces métier sont les points d'accès externes au composant alors que les interfaces de contrôle prennent en charge des propriétés non fonctionnelles du composant comme la gestion de son cycle de vie ou de ses liaisons. Une interface Fractal est composée d'un nom, d'une signature et d'un type. Dans la projection du modèle en Java, la signature d'une interface est une interface Java. Une interface Fractal métier est du type client ou serveur. Une interface serveur identifie les services offerts par un composant alors qu'une interface client spécifie les fonctionnalités qu'un composant requiert pour son fonctionnement.



➢ **Le connecteur :**

Fractal ne possède pas de notion de connecteur explicite avec une sémantique de processus. Cependant, il utilise la notion de liaison pour spécifier les interactions entre composants. Une liaison Fractal est définie comme un lien orienté entre une interface client et une interface serveur. Ce lien permet aux composants d'interagir. Le modèle étant fortement typé, le type d'une interface serveur doit être obligatoirement du type ou du sous-type de l'interface cliente à laquelle elle est reliée. La liaison est assimilable à un connecteur implicite, elle ne possède pas de comportement propre.

La membrane possède à la fois des interfaces externes, qui sont accessibles de l'extérieur du composant, et des interfaces internes, accessibles seulement par son contenu. Dans le modèle Fractal, une interface interne ne peut exister que symétriquement à une interface externe. Ce mécanisme d'interface interne sert principalement à permettre les traversées de membranes des composites en conservant la sémantique de la liaison. Ainsi la liaison permet de définir à la fois les interactions entre deux composants mais permet aussi de définir les liens de délégation entre les interfaces d'un composite et les composants de son contenu.

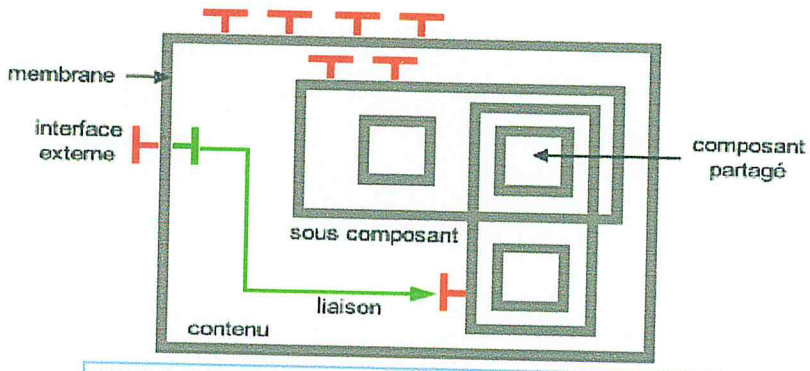


Figure II.2: Structure d'un composant fractal

➢ **La configuration :**

Fractal est un modèle hiérarchique, un composant peut posséder au sein de son contenu d'autres composants. Il est identifié dès lors comme un composite. Le contenu d'un composite définit une configuration pour la mise en œuvre des services définis au niveau de ses interfaces métiers.

Enfin, et c'est une spécificité du modèle Fractal, un composant peut appartenir au contenu de deux composites qui ne sont pas imbriqués l'un dans l'autre. Il est alors dit *partagé*.

- **Avantages:**

  - Fractal est un modèle simple et léger. Sa prise en main est aisée pour les programmeurs et les architectes issus de l'objet.
  - Fractal permet de construire des applications par composition et son modèle de composant est indépendant de toute technologie,
  - Il fournit plusieurs implémentations du modèle dans divers langages.

- **Inconvénients :**

  - Fractal n'intègre pas explicitement la notion de port. Elle est intégrée dans la notion d'interface cela provoque bien souvent des ambiguïtés,
  - Le support des connecteurs via les binding components semble assez primitif et peu étudié,
  - Fractal ne permet pas une séparation claire entre les propriétés fonctionnelles et non fonctionnelles prises en charge par la membrane.

- **Exemple :** L'exemple ci-dessous montre une application très simple constituée d'un composant composite contenant deux composants primitifs. Le premier composant primitif est *"server"* et qui fournit une interface *s*. Le deuxième composant primitif est *"client"* lié à l'interface du serveur précédent (Figure II.3).

```
//Le composant Serveur
@Component(provides=@Interface(name="s",signature=Service.class) )

public class ServeurImpl implements Service {
   public void print( String msg ) {
     System.out.println(msg);
     }
 }
//Le composant Client
@Component(provides=@Interface(name="r",signature=Runnable.class) )
public class ClientImpl implements Runnable {
@Requires(name="s")
private Service service;
     public void run() {
        service.print("Hello world!");
      }
}
//L'assemblage
<definition name="HelloWorld">
<interface name="r" role="server" signature="java.lang.Runnable" />
<component name="client" definition="ClientImpl" />
<component name="serveur" definition="ServeurImpl" />
<binding client="this.r" server="client.r" />
<binding client="client.s" server="serveur.s" />
</definition>
```

Figure II.3: Exemple de description fractal

II.1.3.2. Wright

Wright est un LDA, construit autour des trois éléments suivants : les «composants», les «connecteurs » et leur « configuration ».

- **Le modèle de base :** Un composant contient une « interface » et un comportement. L'interface est constituée d'un ensemble de « ports ». Chaque port représente une interaction à laquelle le composant participe. La « computation » (calcul) donne une description plus précise de la fonction du composant ; et spécifie si un port est requis ou fourni.
- **Les connecteurs :** Les connecteurs, qui sont explicites en Wright, se composent d'un ensemble de « rôles » et d'une *glue* ; chaque rôle spécifie le comportement d'un seul «participant » à l'interaction. La *glue*, spécification du comportement des composants, décrit leur interaction. La configuration est un ensemble d'« instances » de composants combinées via des connecteurs. Ces liaisons se font par les *attachments* qui connectent un port d'un composant à un rôle d'un connecteur.
- **L'architecture :** Wright donne aussi la possibilité de définir des « sous-composants » et des « Styles » d'architectures qui possèdent des « types

d'interface », des «contraintes » et des « paramètres». Le comportement et la coordination des composants sont spécifiés en « CPS » (*Communicating Sequential Process*), ce qui permet de définir des « events », des « process » et surtout des « contraintes sémantiques ».

- **Avantages :**

  - Permet de spécifier une architecture logicielle de manière formelle et totalement abstraite (composant, connecteur, configuration),
  - Présence d'un modèle abstrait de composant et d'un modèle abstrait de connecteur. Les deux modèles sont indépendants.

- **Inconvénients :**

  - Difficile à assimiler,
  - Wright n'apporte pas de solution pour la hiérarchisation et la structuration de l'application,
  - Wright ne possède pas de moyen de projeter l'architecture logicielle vers un système concret (le passage de l'abstrait au concret est difficile),
  - Peu de moyens pour séparer les spécifications fonctionnelles et non fonctionnelles.

II.1.3.3. ACM: Aspectualizing Component Model *[Hannousse,2012]*.

Ce travail a été proposé *à L'Université Nantes Angers Le Mans* consiste à introduire un ADL qui s'étend ADLs actuels par une définition explicite des comportements des composants et d'aspects. Cette approche introduit un langage déclarative de points de coupure (VIL) dédié au modèles à composants, il défini les règles de tissage et de composition d'aspects et aussi la détection et la résolution des interférences d'aspects lorsque plusieurs aspects sont tissés à un système à composants. Chaque règle utilise des expressions VIL afin de décrire déclarativement où les aspects vont être tissés.

[Hannousse, 2012] fournis dans son approche un ensemble de règles de transformation

pour obtenir la spécification formelle des composants et des aspects à partir de l'ADL, il utilise le model checker Uppaal **[Larsen, 1997] [Behrmann, 2004]** pour la détection des interférences possibles entre les aspects.

➢ **Exemple:**

Nous prenons l'exemple suivant pour illustrer comment l'ADL de **[Hannousse, 2012]** décrit une architecture logicielle.

```
System CmpSalam{
   Interface  comm { @Syn get_message();}
   Primitive CmpServer {
         provides comm;
         String get_message(){
               return "Assalamu Alaykum";
               }
         }
   Primitive CmpClient {
         Requires comm;
         // behavior
         }
   binding client client.comm server server.comm;
}
```

Figure II.4: Architecture d'un système à base de deux composants et leur description en l'ADL de **[Hannousse,2012]**

➢ **Avantages:** L'approche de **[Hannousse,2012]** combine les deux approches composant et aspect:

- Permet un tissage/détissage dynamique,
- Possède un modèle de composant hiérarchique les différents composants peuvent être encapsulé par un composant composite. Il possède un langage de coupe approprié aux composants qui définit des points dans l'architecture à composants.

➢ **Inconvénients:** Cet ADL est à haut niveau et ne possède pas des outils pour transformer ses architectures vers l'implementation.

II.2. La conception par aspect:

La POO s'avère particulièrement limitée dans l'expression des fonctionnalités dites horizontales ou transverses, celles exprimant les aspects techniques de l'application. En effet de telles fonctionnalités tendent à s'étaler sur un grand nombre de modules (objets dans la POO) souvent sans rapport entre eux.

Les chercheurs se sont penchés sur ce sujet et ont conçu la programmation orientée aspect (POA) afin de pallier cette faiblesse de la POO dans l'expression des fonctionnalités transverses.

II.2.1. Fondements de la Programmation Orientée aspect:

Nous pouvons inférer des problèmes évoqués ci-dessus qu'il serait intéressant de modulariser l'implantation des préoccupations transverses. On parle alors de séparation de ces préoccupations. La Programmation Orientée aspect (POA) est une approche (parmi d'autres) permettant d'atteindre ce but.

La POA permet d'implanter les préoccupations transverses indépendamment les unes des autres et de les combiner ultérieurement pour produire le système final. L'unité de modularité en POA est appelée un Aspect tout comme l'unité de modularité en POO est appelée une Classe.

En général, le cycle de développement en POA se fait en trois étapes :

1) **La décomposition aspectuelle :** consiste à décomposer les besoins afin d'identifier et séparer les problématiques transverses et métiers. Cette phase est souvent comparée au passage d'un rayon de lumière à travers un prisme afin de séparer ses différentes composantes chromatiques.
2) **Implantation des préoccupations :** consiste à implanter chaque problématique séparément. Les problématiques métiers sont implantés moyennant les techniques conventionnelles de la POO alors que les problématiques transverses sont implantées moyennant les techniques de la POA.
3) **Recomposition aspectuelle :** consiste à construire le système final en intégrant ou recoupant les problématiques métiers avec les problématiques transverses. Cette phase est appelée Tissage (weaving en anglais). Un tisseur (weaver) utilise des règles spécifiées par le concepteur de l'application afin de recouper correctement les problématiques entre-elles. Par analogie, nous pouvons comparer cette étape à un nouveau passage des composantes chromatiques dans un prisme qui les combine pour faire sortir un rayon de lumière unique. Le tissage peut être statique ou dynamique :
    - **Le tissage statique :** est effectué par le compilateur qui prend en entrée le code de base de l'application ainsi que les aspects et fournit en sortie l'application étendue avec les aspects. Le tissage statique permet

d'avoir de bonnes performances d'exécution cependant il ne permet pas aux aspects d'apparaître de façon distincte dans l'application finie. On ne peut donc plus ajouter, retirer ou changer les aspects après compilation. AspectJ fournit un tissage statique.

- **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. JAC **[PDFS, 2002]** et JBoss AOP **[Jboss, 2002]** fournissent un tissage dynamique.

II.2.2. Concepts et terminologies POA:

De nouveaux concepts sont introduits avec la POA afin de permettre aux développeurs de spécifier et implanter les préoccupations transverses:

1. **Point de Jonction (joinpoint en anglais):** Endroit précis dans l'exécution du programme. Par exemple, un appel à une méthode, à un constructeur…

   Les Conseils sont insérés au niveau des Points de Jonctions.
2. **Coupe (pointcut en anglais):** Constitue le moyen de spécifier un ensemble de Points de Jonctions particuliers. Une Coupe est souvent une expression régulière.
3. **Conseil (advice en anglais):** Fragment de code à insérer au niveau des Points de Jonction. Implante une préoccupation transverse.
4. **Aspect:** C'est une unité de regroupement :
   - D'une ou de plusieurs définitions de Coupes.
   - D'une ou de plusieurs définitions de Conseils.
   - D'une ou de plusieurs associations de Coupes à des Conseils.
5. **Tisseur (weaver en anglais):** Est un outil spécial permettant d'appliquer les aspects au code de base.

II.2.3. Bénéfices de la POA:

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 et d'assurer une meilleure réutilisation du code.

De plus, les études montrent que la surcharge introduite par les approches orientées aspect est relativement faible. Enfin, les implantations orientées aspect ont des niveaux d'adaptabilité et de réutilisation plus élevés que les implantations uniquement objet.

II.2.4. Exemples d'implémentations de la POA:

De nombreux efforts ont été menés a fin de construire des outils permettant d'adopter les principes de la POA. AspectJ étant le plus connu, Il fut conçu au Xerox PARC par l'équipe de Gregor Kiczales, Il s'agit d'une extension du langage Java, qui prend en charge les concepts cités précédemment.

Actuellement il existe une panoplie d'outils de développement d'applications orientées Aspect, on peut citer à titre d'exemple JAC Java Aspect Components **[PDFS, 2002]**, JBOSS **[Jboss, 2002]** et Spring AOP **[RJ, 2006]**. Toutes ces implémentations permettent d'introduire les concepts de base de la POA, avec une différence mineure dans la terminologie utilisée, dans la conception, et dans l'approche utilisée pour tisser les aspects, l'approche ***Langage*** consiste à étendre un langage orienté objet pour prendre en charge la POA, comme c'est le cas pour AspectJ qui nécessite un compilateur spécial, cette approche agit lors de la compilation pour tisser les aspects. L'approche ***Framework*** n'a pas besoin d'un compilateur vu qu'elle est utilisée avec un langage orientée objet comme Java par exemple, elle fournit un framework avec une API qui permet de représenter les aspects et de les tisser au moment du chargement ou lors de l'exécution, en se référant à des fichiers de configurations souvent écrits en XML, cette technique permet une meilleure flexibilité , on peut modifier la configuration des aspects même lorsque l'application est en exécution.

II.3. La conception par combinaison aspect/composant :

Les aspects peuvent apporter aux composants logiciels un support pour les propriétés transverses d'un système à base de composants. Réciproquement, les composants apportent aux aspects des propriétés structurantes, ce qui permet de gagner en modularité ainsi qu'en abstraction à l'aide de la vue architecturale proposée par les langages d'architecture.

La combinaison de l'approche basée composant et l'approche aspect permet un développement rapide des systèmes.

II.3.1. Classification des approches de combinaison :

Les approches actuelles de combinaison, peuvent être classées en deux catégories: les ***approches symétriques*** et les ***approches asymétriques***. Notre choix s'est orienté vers les travaux les plus connus dans le domaine de la combinaison des deux paradigmes. Chacune des approches considérées traite la combinaison d'une manière plus ou moins différente des autres, ce qui permet de voir la combinaison selon des points de vue variés, et d'identifier le maximum de problèmes potentiels liés à cette combinaison.

- **Approches symétriques :** dans les approches symétriques, les aspects et les composants sont considérés comme entités uniformes. Ex : FuseJ **[Suvée et al, 2006]**, FAC **[Nicolas, 2007]** ... etc.
- **Approches asymétriques :** Les approches asymétriques traitent les aspects et les composants du système de façon différente. Ex : JasCo **[Suvée, 2003]**.

Conclusion :

Dans ce chapitre nous avons situe le cadre général de notre travail. Il aborde le domaine des architectures logicielles. Nous avons exposé les concepts, les définitions et la terminologie des architectures logicielles. Nous avons aussi étudié dans ce chapitre les caractéristiques principales des approches à composants et des approches par aspects. Dans le chapitre suivant nous allons présenter les principales approches à composants au niveaux implémentation avec une évaluation pour chaque un.

Chapitre III:
Les approches à composants au niveau implémentation

Chapitre III . Approches à composants au niveau implémentation

Introduction :

Dans le chapitre précédent nous avons cité le domaine d'architecture logicielle et les principes fondamentaux liés aux approches à composants et celle par aspects, dans ce chapitre nous allons présenter les principales approches à composants au niveau implémentation avec une évaluation de notre part pour chaque approche du point de vue composant et architecture et du point de vue aspect.

Nous abordons les approches suivantes.

- ArchJava **[JCD, 2002a] [JCD, 2002b]** (Section **III.1.1**).
- JBoss AOP **[Jboss, 2002]** (Section **III.1.2**).
- JasCo **[Suvée, 2003]** (Section **III.1.3**).
- CAM/DAOP **[M. Pinto, 2005] [M. Pinto, 2003]** (Section **III.1.4**).
- Spring AOP **[RJ, 2006]** (Section **III.1.5**).
- FuseJ **[Suvée et al, 2006]** (Section **III.1.6**).
- FAC **[Nicolas, 2007]** (Section **III.1.7**).

III.1. Présentations et évaluations des approches à composants:

III.1.1. ArchJava :

C'est un langage permettant de décrire l'architecture logicielle de manière formelle créé par l'université de Washington ' Département of Computer Sciene and Enginering' USA dont les concepteurs principaux sont Jonathan Aldrich et Craig Chambers *[JCD, 2002a] [JCD, 2002b].*

III.1.1.1. Présentation :

ArchJava se situe à la frontière entre le langage de description d'architectures et les modèles de construction d'applications à composants. ArchJava a pour objectif d'améliorer la compréhension des programmes, de garantir l'architecture de l'application, il étend le langage Java afin d'incorporer les éléments d'architectures à l'intérieur du code.

Le modèle de composant d'ArchJava définit les trois concepts de composant, connecteur et configuration. Un composant est la seule unité d'encapsulation définie dans le langage ArchJava, il communique avec d'autres composants via des ports. Un port définit un ensemble de méthodes, au sens Java classique, qui définissent des points d'accès à un composant.

ArchJava reprend ces concepts (composant, connecteur et configuration), comme élément de structure du code de l'application. Il ajoute de nouveaux éléments de syntaxe au langage Java pour gérer les composants, les connecteurs et les ports. Les composants composites sont les unités de configuration. En effet, ce sont eux qui contiennent la description des composants d'une application ainsi que leurs interconnections. Une application est donc un composant composite. La communication entre les composants se fait par appel de méthodes.

ArchJava garantit que les communications entre composants respecteront les connexions définies au niveau de l'architecture et qu'aucune autre interaction entre composants ne sera possible. La vérification de cette propriété est réalisée à la compilation, évitant l'apparition impromptue de messages d'erreur à l'exécution. Ainsi, les interactions entre composants connectés et entre un parent et ses sous-composants immédiats sont autorisées. Par contre, les appels externes à un sous

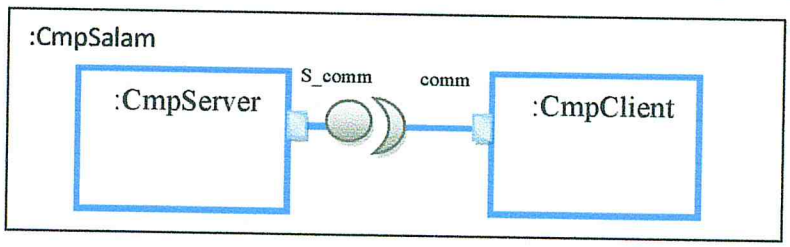
composant et les appels entre composants non connectés sont interdits. ArchJava interdit aussi les appels violant la structure hiérarchique de l'architecture.

ArchJava permet enfin de définir des classes de connexion appelées plus communément connecteurs complexes. En effet, lorsque le programmeur désire découpler l'implémentation des composants de la façon dont ils communiquent, il peut transformer les connexions entre les composants en objets à part entière. Des objets dont le seul but est de gérer la communication en tant que telle.

III.1.1.2. Exemple :

Pour illustrer la spécification en ArchJava, nous proposons de reprendre l'exemple de code "Assalamu Alaykum" en ArchJava Le composant ***Serveur*** offre un service get_message au sein de l'interface ***s_comm*** et composant ***Client*** requis un service get_message au sein de l'interface comm. Voir Figure (III.1):

(a)




```
Component class CmpServer{
    private String message= new String("Assalamu Alaykum");
    protected String getMessage(){return message};
    public port s_comm(){
            provides String get_message(){return this.getMessage();}
}
Component class CmpClient{

    public port comm(){
            requires String get_message();
}
Component class CmpSalam{
    private final CmpServer server=new CmpServer();
    private final CmpClient client=new CmpClient();
    Connect server.s_comm, client.comm;
}
```

Figure III.1: Architecture d'un exemple à base de deux composants (a) et leur description en ArchJava (b)

III.1.1.3. Evaluation :

ArchJava est une extension de Java dont la syntaxe est largement connue ce qui permet un apprentissage rapide et une bonne lisibilité du programme. La complexité de ArchJava réside dans le fait est que les connecteurs sont implicites et nécessitent vraiment une conception a part difficilement réalisable par des personnes non expert en java et en technologie de communication basée sur java, Ce qui rend ce langage ne semble pas avoir été mis en œuvre dans un processus de transformation d'une architecture logicielle vers l'implémentation.

La communication entre les composants se fait qu'à l'aide d'un appel de méthode cela ne correspond pas forcément avec les violentés des architectes, aussi dans ArchJava il y'a manque de séparation claire entre les propriétés fonctionnelles et non fonctionnelles c.-à-d. qu'il n'est pas ni de prés ni de loin un langage de programmation orienté aspect, toute la gestion des propriétés non fonctionnelles d'une application est à la charge du programmeur.

Un *composant ArchJava* peut être utilisé comme une classe Java, donc il est possible d'utiliser l'*héritage* entre *composants* ou de définir des *composants* abstraits par exemple. Un composant peut aussi implémenter des interfaces Java.

III.1.2. JBoss AOP :

JBoss AOP ***[Jboss, 2002]*** est un framework pour aspects qui peut aussi bien s'utiliser individuellement ou conjointement avec le serveur d'application JBoss.

III.1.2.1 Présentation :

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

liaison à la base sont détachées, caractéristique d'une symétrie de placement. JBoss AOP, en plus d'offrir un mécanisme d'interception classique des approches par aspects, offre également un mécanisme d'introduction, ainsi que quelques mécanismes de mixins[1] que nous ne détaillons pas plus car non considérés dans notre étude. Les points de jonctions pris en compte par JBoss AOP sont les accès aux attributs, les invocations de méthodes, ou encore un constructeur d'objet. Lors d'une interception de méthode, il est possible d'intervenir du côté de l'appelant ou de l'appelé. L'interception ne s'arrête pas aux frontières d'une classe, il est possible d'atteindre les éléments publics tout comme privés ou protégés.

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

Les codes advice sont écrits en pur Java. Une interface de programmation permet d'accéder au contexte d'interception par réflexion. Il est donc possible de connaître l'appelant, l'appelé, la méthode interceptée, etc.

III.1.2.2. Exemple : puisque JBoss AOP utilise un mécanisme de liaison qui encore une fois fonctionne soit par annotations soit dans les descripteurs XML en prend un exemple pour le cas XML :

[1] Un **mixin** est une classe abstraite. C'est un cas de réutilisation d'implémentation. Chaque mixin représente un service qu'il est possible de greffer aux classes héritières.

❖ **JBoss AOP :** exemple d'aspect utilisant les descripteurs XML :

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

Figure III.2: exemple d'un aspect en JBoss AOP

III.1.2.3. Evaluation:

En résumé, JBoss AOP est avant tout une approche par aspects qui s'applique au niveau objet, et donc au niveau de l'implantation des ***composants EJB***. L'approche reste donc éloignée en termes de symétrie, de modèle général et hiérarchique. Les aspects sont en pur Java mais ne sont pas des composants ce qui rend l'approche asymétrie d'élément. Les aspects sont donc appliqués au niveau de l'implantation des composants et non comme concept de premier ordre. Le modèle sépare clairement la définition d'un aspect de sa liaison aux composants (placement symétrique) qui peut être définie par annotations dans le code ou à l'aide des descripteurs XML de l'application.

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

III.1.3. JasCo : est un modèle de composants conçu à la Vrije Universiteit Brussel. JasCo étend le modèle Java bean.

III.1.3.1. Présentation :

Dans cette approche il y'a deux concepts : les aspects beans et les connecteurs. Le premier décrit le comportement qui interfère avec l'exécution d'un composant en utilisant des classes internes, appelées hook, dont la spécification est réutilisable indépendamment du contexte. Le second est utilisé pour déployer un ou plusieurs hook dans un contexte spécifique. Dans ce travail, un nouveau modèle de composant a été proposé en se basant sur la

notion de trappes intégrées qui permettent d'interférer avec l'exécution normale d'un composant. JasCo est compatible avec le modèle de composant Java beans et permet l'ajout/retrait dynamique des aspects. JasCo résout partiellement le problème d'interaction des aspects conflictuels en permettant de les ordonnancer et de décrire des combinaisons d'aspects explicites et réutilisables.

III.1.3.2. **Evaluation :** 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. JasCo laisse la priorité des aspects varier selon les applications, permet aussi l'ajout/retrait dynamique des aspects, le projet JasCo se continue actuellement dans le cadre du projet de FuseJ [Suvée et al., 2006].

III.1.4. CAM/DAOP :

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

III.1.4.1. Présentation :

CAM est un nouveau modèle à composants et par aspects et s'inspire des standards EJB et CCM. Deux entités de premier ordre existent dans CAM : les composants et les aspects. Cela la place dans la catégorie des approches asymétriques (asymétrie d'éléments). Les composants et les aspects sont des unités de composition au sens de la définition de Szyperski [Szyperski, 2002], c'est-à-dire des entités auto-contenues à gros grain, qui peuvent être déployées indépendamment. Tout comme CCM [OMG, 2002], CAM utilise un langage de description d'interface qui permet de décrire les services fournis et requis d'une interface. Ces descriptions d'interfaces font partie du langage d'architecture DAOP-ADL[2] utilisé pour décrire les composants et les aspects.

[2] 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.

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

Les communications dans CAM se font uniquement par message, DAOP étant une 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.

Les points de jonction supportés par CAM sont les interfaces publiques d'un composant. Le contenu d'un composant ne peut être affecté par des aspects pour préserver la propriété d'encapsulation des composants. Les points de jonction considérés sont donc les messages émis et reçus entre les composants ainsi que la création ou destruction d'un composant. Les coupes sont définies au niveau du langage d'architecture DAOP-ADL, donc extérieurement aux aspects respectant ainsi une symétrie de placement (symétrie de relation). Les aspects ont accès à un contexte d'interception permettant de connaître le composant émetteur ou récepteur, ainsi que le message.CAM/DAOP est un modèle général au sens où il a été conçu pour être indépendant de tout langage de programmation. Pour le moment, CAM/DAOP est implanté en Java et Java RMI est utilisé pour la communication entre composants.

III.1.4.2. Exemple : Application de chat [1]

- **CAM/DAOP** exemple d'un composant en DAOP-ADL

```
<component role="chat">
   <providedInterface>ChatProv.xml</providedInterface>
   <requiredInterface>
      <fromTargetComponent role=chat/>
      <requiresMessage>ChatReq.xml</requiresMessages>
   </requiredInterface>
   <requiredInterface>
      <fromTargetComponent role=awarenessList/>
      <requiresMessage>AwarReqInt.xml</requiresMessages>
   </requiredInterface>
   <implementations>
      <implementation>
         <name>chat1</name>
         <language>java</language>
         <class>Chat.class</class>
      </implementation>
   </implementations>
</component>
```

Figure III.3: exemple de définition d'un composant en CAM/DAOP

- **CAM/DAOP** exemple de définition d'un aspect en DAOP-ADL : Tout comme un composant, un rôle est attribué ainsi qu'une interface dite évaluée qui définit le type de point de jonction.

```
<aspect role="persistence">
  <evaluatedInterface>
       <joinpoint>BEFORE_SEND</joinpoint>
       <capturedMessages>PersistenceEval.xml</capturedMessages>
  </evaluatedInterface >

<implementations>
       <implementation>
         <name>persistence1</name>
         <language>java</language>
         <class>LDAPPersistence.class</class>
       </implementation>
       <implementation>
         <name>persistence2</name>
         <language>java</language>
         <class>OraclePersistence.class</class>
       </implementation>
</implementations>
</aspect>
```

Figure III.3.a: exemple de définition d'un aspect en CAM/DAOP

- **CAM/DAOP** exemple de définition d'une composition en DAOP-ADL : L'exemple suivant 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.

```
<compositionRules>
   <componentCompositionRules>
      <compositionRuleFor role="chat">
      <compositionRule>
      <formatRole>awarenessList</formatRole>
      <realRole>userList</realRole>
      </compositionRule>
      </compositionRuleFor>
   </componentCompositionRules>
   <aspectEvaluationRules>
      <createComponent role="chat">
        <BEFORE_NEW>
          <concurrent>
            <aspect role="authentification"/>
          </concurrent>
        </BEFORE_NEW>
      </createComponent>
      ...
    </aspectEvaluationRules>
</compositionRules>
```

III.1.4.3. Evaluation :

CAM/DAOP se distingue par son modèle qu'est indépendant de tout langage de programmation et de son langage d'architecture, l'interconnexion se fait par envoi de messages. 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. Il sépare clairement la définition d'un aspect de sa liaison aux composants (placement symétrique).

III.1.5. Spring AOP:

Spring est un modèle à conteneur léger qui permet une intégration des services J2EE. Les modèles à conteneurs légers marquent un retour vers le paradigme objet, là où les serveurs d'applications ont prôné l'utilisation de composants EJB à gros grain, difficiles à configurer et à maintenir, et parfois inadaptés aux besoins. Spring dans sa seconde version offre un support pour le langage AspectJ c'est le module Spring AOP.

III. 1.5.1. Présentation :

La partie aspect de Spring est en pur Java et implante donc des interfaces du canevas Spring. Ces interfaces existent pour les coupes et les codes advice, ou encore les *advisor* qui encapsulent les deux à la fois. De manière similiaire à JBoss AOP, les aspects Spring peuvent s'appliquer sur les ***composants beans*** qui sont de toute manière des objets. Les aspects peuvent à la fois atteindre les objets hébergés par un conteneur à inversion de contrôle ou des objets extérieurs. Contrairement à JBoss AOP, Spring AOP supporte uniquement l'interception et

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

III. 1.5.2. Exemple :

Pour bien illustre l'objectif de ce framework on prend l'exemple suivant : Soit l'application Calculatrice qui permet de faire l'addition ou la soustraction de deux entier. Le principe est d'intercepter les entrées/sorties des méthodes "addition(int,int)" et "soustraction(int,int) " et de les logger à l'aide d'une classe externe "CalculatriceLogger".

- **La classe calculatrice :** La classe Calculatrice à tracer est la suivante:

```
public class Calculatrice {
	public int addition(int a, int b) {
		return a + b;
		}

		public int soustraction(int a, int b) {
		return a - b;
		}
```

- **La classe CalculatriceLogger :** La classe CalculatriceLogger contient les deux méthodes "logMethodEntry()" et "logMethodExit()" qui vont être appelées par Spring-AOP pour tracer les appels des méthodes interceptées. Le code de cette classe est le suivant:

```
public class CalculatriceLogger {
	public void logMethodEntry(JoinPoint joinPoint) {
	Object[] args = joinPoint.getArgs();
	String name = joinPoint.getSignature().toLongString();
	StringBuffer sb = new StringBuffer(name + " called with: [");
	for (int i = 0; i < args.length; i++) {
	Object o = args[i];
	sb.append("'" + o + "'");
	sb.append((i == args.length - 1) ? "" : ", ");
	}
	sb.append("]");

	System.out.println(sb);}
	public void logMethodExit(StaticPart staticPart, Object result) {
	String name = staticPart.getSignature().toLongString();
   System.out.println(name + " returning: [" + result + "]");

	}
```

- Le fichier de définition des beans est le suivant :

Figure III.4.a: exemple de définition d'un aspect en Spring AOP

```
<bean name="calculatrice" class="org.insat.aop.Calculatrice" />
<!-- Debut de la configuration AOP -->
<aop:config>
<aop:pointcut id="servicePointcut"
expression="execution(* org.insat.aop.Calculatrice.*(..))" />
<aop:aspect id="loggingAspect" ref="calculatriceLogger">
<aop:before method="LogMethodEntry" pointcut-ref="servicePointcut"/>
<aop:after-returning method="LogMethodExit" returning="result"
   pointcut-ref="servicePointcut" />
</aop:aspect>
</aop:config>

<bean id="calculatriceLogger" class="org.insat.aop.CalculatriceLogger"/>
<!-- Fin de la configuration AOP -->
</beans>
```

Figure III.4.a: exemple de définition d'un aspect en Spring AOP

III. 1.5.3. Evaluation :

Spring propose un canevas léger pour le support des services J2EE, en ajoutant des propriétés intéressantes, comme l'abstraction réalisée au niveau des fabriques. Il reste indépendant des spécifications et laisse le choix à l'utilisateur de l'implantation ou de l'outil qu'il désire. Cependant cela reste un canevas J2EE donc avec les défauts que nous avions soulignés pour l'approche JBoss AOP : pas de notion de hiérarchie de composants. Du point de vue aspect nous considérons que l'approche est symétrique simplement par le fait que Spring marque un retour à l'utilisation des POJOs, même pour les composants beans. Les aspects étant eux mêmes écrits en pur Java, l'approche est dite symétrique. La symétrie de placement est également respectée, car les descriptions de coupes sont séparées des codes advice.

III. 1.6. FuseJ :

FuseJ [Suvée et al., 2006] est un modèle pour composants et aspects qui propose une unification des deux paradigmes (composant et aspect). De ce fait l'approche est dite symétrique.

FuseJ introduit un nouveau modèle à composants pour ses besoins ainsi qu'une terminologie associée pour la définition des interfaces fournies et requises qui sont des portes (*gates*).

III. 1.6.1. Présentation : L'objectif du projet de recherche FuseJ est donc d'étudier les caractéristiques de la symétrie d'éléments (appelé aussi unification). L'approche fait le choix de ne faire aucune distinction entre un aspect et un composant. Seule l'interaction entre deux modules dont l'un joue le rôle d'aspect

requiert un nouveau mécanisme : une composition orientée aspect. Les composants et les aspects sont des beans Java dans FuseJ. La composition par aspects de FuseJ permet de les tisser sur d'autres beans. Le modèle à composant est très simple et reste très proche du langage Java. Ainsi, un composant est une classe Java et le principe d'interfaces fournies et requises est mis en oeuvre par des classes Java. Les services d'un composant sont ensuite spécifiés dans un nouveau langage, que l'on peut rapprocher des langages d'interfaces (*IDL – Interface Description Language*). Une spécification de services fournit donc une liste des interfaces offertes et implantées par le composant et des interfaces requises ou attendues par ce même composant.

III. 1.6.2. Exemple :

- **FuseJ :** Exemple de spécification d'un service

```
interface TransferI {
        byte[] getFileFragment(String aFileName);
        FileFragementInfo findFileFragment(String aFileName);
}
interface NetworkI {
        void send(String host, String info);
        byte[] get();
}
service TransferS {
        provides TransferI;
        expects NetworkI;
}
```

Deux interfaces sont spécifiées, qui sont des interfaces Java classiques (TransferI et NetworkI). Ensuite un service est vu comme un ensemble d'interfaces fournies ou requises. Le service ***TransferS*** déclare l'interface ***TransferI*** comme une interface fournie et l'interface ***NetworkI*** comme une interface requise.

- **FuseJ :** Maintenant l'exemple suivant montre comment le service *TransferS* est représenté par l'interface Java TransferS implantée par la classe TransferC représentant le composant *TransferC*.

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

Figure III.5: exemple de définition d'un Composant en FuseJ

Dans ce même exemple, les appels aux méthodes send et get sont des appels aux opérations requises de l'interface NetworkI. Cela signifie que le code d'un composant utilisant des services requis ne peut compiler sans être complété a posteriori par FuseJ qui injectera les bonnes dépendances. Ceci constitue une première embûche à l'utilisation de FuseJ.

- **FuseJ :** l'exemple suivant montre une interaction orienté aspect entre deux composants (***TransferNetC*** et ***LoogerC***) :

```
configuration LoggedTransferNetC configures
 TransferNetC, LoggerC as TransferNetS {

        Linklet log {
        Execute:
          LoggerC.log(String st);
        Before:
          TransferNetC.*(..);
        Where:
          st = Source.getMethodSignature();
        }
}
```

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

III. 1.6.3. Evaluation :

Le principal atout du projet de recherche FuseJ est son choix marqué d'unifier composants et aspects. De ce fait l'approche est symétrique du point de vue des éléments et des relations, aussi bien la portée que le placement. Aucune distinction n'est observée entre un composant et un aspect. Tout service d'un composant peut être potentiellement employé comme un code advice. Toute la composition dans FuseJ intervient au niveau de son langage de configuration et des *linklet*. Ces *linklet* peuvent à la fois spécifier des compositions classiques de type client/serveur entre composants ou alors des compositions par aspects.

Le modèle FuseJ n'est pas vraiment hiérarchique du fait de la complexité de la composition de composants entre eux. En effet, chaque composition dans FuseJ produit un composite, qui à nouveau, peut être

composé avec un autre composant en déléguant toutes ses opérations. Nous sommes loin, dans ce cas de figure, des modèles hiérarchiques classiques. Enfin, notons que FuseJ est encore à l'état de prototype et que l'instrumentation du code est encore réalisée manuellement.

III. 1.7. FAC :

Une extension du modèle FRACTAL qui unifie les principes des approches à composants et par aspects [Nicolas, 2007].

III. 1.7.1. Présentation : FAC intègre la notion d'aspect. Le but est de capturer les propriétés transversales du système. Dans FAC, les aspects sont représentés par des composants Fractal. Ils sont appelés des composants d'aspect (Aspect Components ou AC). Ils sont dotés d'une interface serveur spécifique qui implémente l'API AOP Alliance. Les points de coupe sont définis à travers des expressions régulières au niveau du connecteur. Un point de coupe sélectionne les composants, les interfaces et les méthodes sur lesquels l'AC sera appliqué. FAC permet des points de coupe structurels ou comportementaux. Un AC est tissé et détissé à l'exécution. FAC définit une API pour gérer le tissage (voir figure III.6). Le processus de tissage est très similaire au processus de connexion entre une interface client fonctionnelle et une interface serveur dans Fractal. Est appelé connecteur, cette interaction entre un ensemble de composants et un AC. Ces connecteurs transverses sont ainsi définis par une API ou à travers un niveau d'ADL. FAC travaille sur plusieurs niveaux et précisément sur plusieurs paradigmes (composants, langages). Certains problèmes ont été soulevés comme celui de la cohérence entre les différents niveaux d'expression. Appliquer les modifications au moment de l'exécution ou au moment de la conception représente également un problème. Appliquer des modifications à l'exécution en utilisant des transformations requiert des précautions particulières.

```
interface AspectController {
/** Méthode de tissage. */
void weave(Component root, AspectComponent ac,
ItfPointcutExpr pcd, String acName );
/** Méthode de détissage. */
void unweave( Component root, Component ac );
/** Réordonnancement des aspects tissés sur le composant courant. */
String[] changeACorder( String acName, int newPosition );
/** Méthodes d'introspection de coupes. */
String[] listACNames();
Component[] listAC();
Component[] listCrosscutComps( Component root, Component ac );
Pointcut aspectizableComps( Component root, ItfPointcutExp pcd );
}
```

Figure III.6: API de tissage de FAC

III. 1.7.2. Exemple :

Comanche et la journalisation: Comanche est un serveur HTTP minimal conçu à l'aide de composants FRACTAL, qui consiste à accepter des connexions TCP et à traiter ces requêtes par l'envoi du fichier demandé ou d'une erreur. Nous allons à présent étudier l'application des principes de FAC sur cet exemple de Comanche. La figure (III.7) fournit une illustration conceptuelle de ce qui est attendu pour cet exemple.

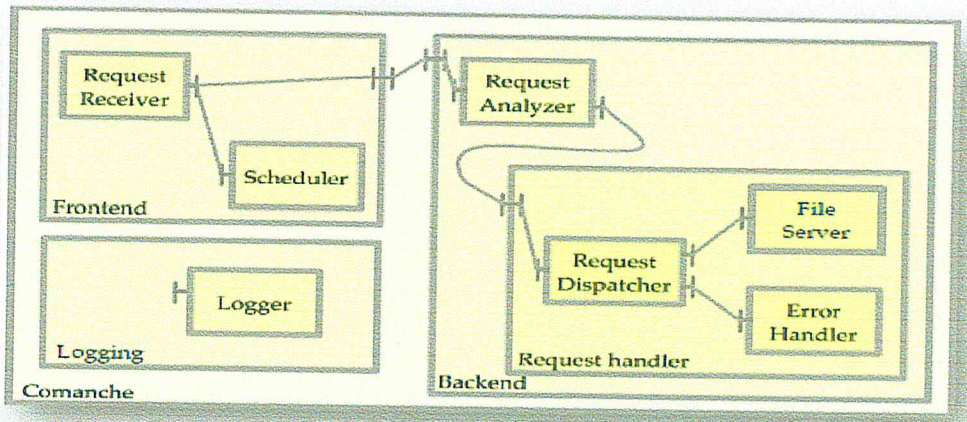


Figure III.7: Exemple « comanche » : la journalisation

❖ **FAC** : Définition de la propriété transverse: le composant ***Logger***

```
/** Interface de journalisation */
@Interface(name="logger")
public interface ILogger { void log (String msg); }

/** Composant de journalisation sur la sortie
standard */@FractalComponent
public class Logger implements Logger {
 public void log (String msg) {
System.out.println(msg); }
}
```

Figure III.8 Exemple de définition d'un aspect en FAC

❖ **Définition du/des composant(s) d'aspect :** consiste à définir un connecteur pour adapter le composant *Logger* à une utilisation transverse. Autrement dit, un composant d'aspect doit être défini pour capturer le contexte d'interception des composants sur lequel la journalisation va s'appliquer. Le composant d'aspect de journalisation ici se contente d'être connecté, par une liaison standard, au composant *Logger*.

❖ **Tissage avec Fractal_ADL :**

Le code suivant est la description de l'assemblage de l'exemple Comanche de la Figure (III.7) :

```
<definition name="comanche.Comanche" >
<interface name="r" signature="java.lang.Runnable"   role="server"
cardinality="singleton" contingency="mandatory"/>
<component name="frontend" definition="comanche.FrontendComposite" />
 <component name="backend" definition="comanche.BackendComposite" />
 <component name="logging" definition="comanche.LoggingComposite" />
 <binding client="this.r" server="frontend.r" />
 <binding client="frontend.rh" server="backend.rh" />
</definition>
```

Figure III.9.a Exemple de description d'assemblage en FAC

Le code suivant permet l'assemblage du composite Logging de la figure (III.7)

```
<definition name="comanche.LoggingComposite" >
 <component name="logger">
 <interface name="logger" signature="comanche.ILogger"
role="server"/>
 </component>
 <component name="loggingAC">
 <interface name="aspectComponent"
signature="fac.AdviceInterface" role="server" />
 <interface name="logger" signature="comanche.ILogger"
role="client"/>
 </component>

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

Figure III.9.b Exemple de description d'assemblage en FAC

III. 1.7.3. Evaluation : FAC offre cinq contributions originales à l'approche aspect. D'abord, il définit un aspect encapsulé dans un composant logiciel. Il définit un ordre d'exécution des greffons au niveau de chaque point de jonction. Ensuite, il rend possible l'extraction des préoccupations transverses. De plus, il permet la sûreté de l'intégration des aspects par le contrôle des aspects agissant sur un point de jonction par une interface particulière de tissage. Enfin, il autorise le contrôle de l'intérieur des composants composites. FAC est spécifique pour le modèle Fractal donc ce n'est pas un modèle général. Fractal ne possède pas de notion de connecteur explicite avec une sémantique de processus. Cependant, il utilise la notion de liaison pour spécifier les interactions entre composants. Une liaison Fractal est définie

comme un lien orienté entre une interface client et une interface serveur. Ce lien permet aux composants d'interagir.

III.2. Récapitulatif et évaluation des approches à composants

Dan cette section nous allons évaluer et récapitulé tous les approches présenté dans ce chapitre.

III.2.1. Les critères d'évaluation:

Pour l'évaluation des diverses approches nous avons défini un ensemble de critères que nous avons organisé en deux catégories : Les critères relatifs aux concepts fondamentaux de l'architecture logiciel, et les critères relatifs à la manière dont l'orienté aspect est supporté dans ces diverses approches.

III.2.1.1. Les critères relatifs au concept de l'architecture logicielle :

Dans ce contexte nous avons retenus les critères suivant pour l'évaluation des diverses approches du point de vue composant et architecture:

- ***La réutilisation par Héritage de composant.***
- ***La notion de hiérarchie dans la composition:*** par hiérarchisation, les différents composants peuvent être encapsulés par un composant composite. À notre avis, un composite ne doit pas être seulement une abstraction et une encapsulation d'un ensemble de composants, il doit avoir son propre comportement vis-à-vis des appels entrants et sortants de ses interfaces externes. Cela permet aux aspects de s'intégrer au niveau composite et d'être appliqués à tous leurs composants internes.
- ***Le type de communication entre composants*:** appel des méthodes, envoi des messages... etc.
- ***Les contraintes d'interconnexions des composants***: pour déterminer quel sont les connections qui sont permises/refuses.
- ***La faisabilité de la transformation d'un ADL.***
- ***Ajout/suppression dynamiques des composants.***
- ***Ajout/suppression dynamique des connecteurs.***

III.2.1.2. Les critères relatifs au concept d'aspect avec les composants:

- ***Tissage/Détissage dynamique des aspects:*** permettant l'ajout et la suppression des aspects au moment de l'exécution.
- ***Symétrie d'éléments*** : consiste à ne pas faire de distinction entre un composant et un aspect (aussi appelée unification).
- ***Symétrie de placement*** : permet de séparer clairement la définition d'un aspect de sa coupe. Cette symétrie permet d'améliorer la modularité des aspects car la coupe définit où l'aspect doit être tissé, là où sa définition spécifie son comportement, qui peut alors être réutilisé lorsque ces deux parties sont séparées.
- ***Définition de langage de coupure:*** pour déterminer si l'approche possède une expression de points de coupures spécifique aux composants qui définissent des points dans les architectures à composants.
- ***Ordre d'exécution des greffons au niveau de chaque point de jonction.***

III.2.2. Evaluation

Le tableau (III.1) récapitule les approches selon les critères du point de vue composant et architecture:

critère / *L'approche*	*Héritage de composant*	*La notion d'hiérarchique* ***(-) signifie un seul niveau d'hiérarchie***	*Le type de communication entre composants*	*La faisabilité de la transformation d'un ADL.*	*Ajout /suppression dynamique des composants et connecteurs*
ArchJava	Oui	Oui	Appel de méthodes et connecteurs complexes	Oui, avec la difficulté est la synthèse de connecteur complexe	oui
Jboss AOP	Non	Non	Appel de méthodes	Non	Non
JasCo	Non	Non	Appel de méthodes	Non	Oui
CAM/ DAOP	Non	Oui (-)	Envoi de messages	Non	Oui

Spring AOP	Non	Non	Appel de méthodes	Non	Non
FuseJ	Non	Oui	Appel de méthodes	Non	Oui
FAC	Non	Oui	Appel de méthodes	Non	Oui

Tableau III.1: Bilan (1) des approches présentées dans le chapitre

Le tableau (III.2) récapitule les approches selon les critères du point de vue aspect

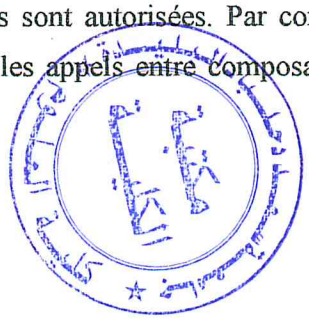
critère / L'approche	*Tissage / détissage dynamique*	*Symétrie d'élément*	*Symétrie de placement*	*Langage de la coupe*	*Ordre d'exécution des greffons*
ArchJava	Non	Non	Non	Non	Non
Jboss AOP	Oui	Non	Oui	Non	Oui
JasCo	Oui	Non	Non	Non	Oui
CAM/ DAOP	Non	Non	Oui	Non	Oui
Spring AOP	Oui	Oui	Oui	Non	Oui
FuseJ	Non	Oui	Oui	Oui	Non
FAC	Oui	Oui	Oui	Oui	Oui

Tableau III.2: Bilan (2) des approches présentées dans le chapitre

III.3. Synthèse:

Nous venons d'étudier un certain nombre d'approches à composants. Le Tableau (III.1) propose une synthèse des approches abordées dans ce chapitre du point de vue composant et architecture et le tableau (III.2) synthèse du point de vue aspect. Nous tirons les conclusions suivantes de cette étude

- Sauf ArchJava parmi les approches citer dans ce chapitre permet d'utiliser l'***héritage*** entre ***composants***.
- Seul **FAC** et **ArchJava** sont des modèles réellement hiérarchiques et **CAM/DAP** permet de définir un seul niveau de hiérarchie où tous les composants figurent à la manière d'OpenCOM.
- La communication entre les composants pour la plupart des approches se fait par appel de méthodes sauf **CAM/DAP** qui possède la communication par envoi de messages.
- Aucune approche ne possède une faisabilité de transformation d'un ADL sauf ArchJava qui est plus proche du code, mais la difficulté de son utilisation réduit leur flexibilité.
- Les approches **FAC**, **Jboss AOP, JasCo**, **Spring AOP** permettent l'ajout/suppression dynamique des aspects au moment de l'exécution.
- Seul **FuseJ** et **FAC** propose une symétrie des éléments.
- La symétrie de placement se vérifie pour chaque approche sauf bien sure **ArchJava** puisque il ne possède pas l'approche aspect et JasCo de sa nature est un approche purement asymétrique, pour le reste des approches de notre étude, ils offrent la possibilité de séparer la définition de la coupe des codes advice.
- Sauf **FuseJ** et **FAC** qui possèdent un langage de coupe appropriée aux composants.
- Pour le critère qui concerne les contraintes d'interconnexion des composants, la plupart des approches académiques ex: FAC, ArchJava, FuseJ permise les interactions entre composants connectés et entre un parent et ses sous-composants immédiats sont autorisées. Par contre, les appels externes à un sous-composant et les appels entre composants non connectés sont interdits.



Conclusion :

Pour conclure sur ce chapitre, nous observons que d'une manière générale aucun approche à composant ne satisfait nos conditions complètement surtout la faisabilité de transformation des ADLs et l'héritage de composants. Le chapitre suivant détaille les concepts fondamentaux de notre approche : AOCF 'Aspect Oriented Component Framework'.

Chapitre IV:
Concepts fondamentaux de l'AOCF

Introduction :

Dans le chapitre précédent nous avons présenté les différentes approches à composants au niveau implémentation existants comme ArchJava, Spring AOP, JBoss AOP, FAC ... etc. nous avons évalué chaque approche selon des critères d'un point de vue composant et architecture et d'un point de vue aspect. A la fin du chapitre nous avons conclus qu'aucune approche ne répond de manière satisfaisante à nos objectifs.

Dans ce chapitre nous allons présenter notre approche « un framework pour la conception orienté composant et par aspects » AOCF : Aspect Oriented Component Framework qui permet d'une part d'être une cible l'implémentation des spécifications ADL d'architectures et d'autre part permet la programmation orientée composant aspect.

IV.1. Principe générale de l'AOCF:

Comme nous l'avions présenté les ADLs (Architecture Description Language) permettent de décrire une architecture logicielle selon un certains nombre des concepts fondamentaux (***composant***, ***connecteur***, ***configuration*** *et* ***interface***). En pratique, ces ADLs permettent un raisonnement à un haut niveau d'abstraction et pour la plupart n'offre pas d'alternative pour transformer la description ADL en une description de niveau implémentation ou ne sont pas couplé à un langage de programmation orienté composant. Le Framework (AOCF) que nous proposons permet l'implantation des concepts fondamentaux de l'architecture logicielle en langage JAVA indépendamment d'un ADL particulier. Son objectif est de prendre en charge les aspects communs à la conception par composants.

Puisque l'approche à composants souffre d'un manque pour le support des propriétés transverses, il promet la réutilisation, mais elle est sujette aux problèmes de dispersion et de mélange de code des propriétés transversales. L'application de la programmation par aspects (AOP) sur les composants logiciels permet de faire face à ces problèmes.

Malheureusement, la plupart des travaux actuels sur la programmation par composants visent à implanter les concepts d'AspectJ **[Kiczales, 2001]** tels quels dans les modèles à composants ignorant la particularité des composants et des systèmes à

composants (e.g. points de coupures définissant des points dans les architectures composants).

AOCF qui est notre proposition unifie les approches à composants et par aspects. Elle permet un développement rapide des systèmes basé sur des composants certifiés et dont les préoccupations non fonctionnelles sont factorisées en dehors de ces composants, AOCF permet d'avoir des systèmes évolutifs, adaptatifs et maintenables. Les composants aspects et les composants métier sont considérés comme entités uniformes (***Symétrie d'élément***), avec une séparation explicite de la définition de points de coupe et de traitement des aspects (advice) (***Symétrie de placement***).

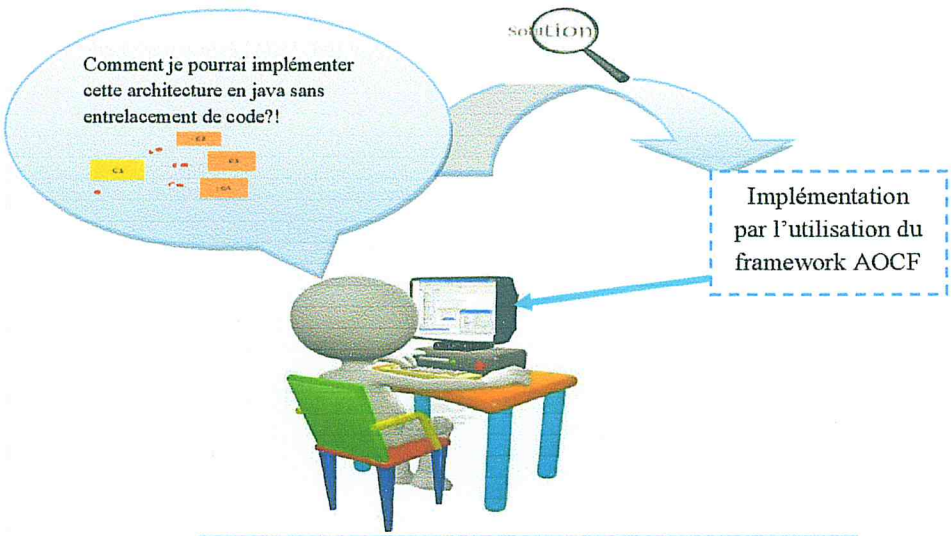


Figure IV.1: Principe générale de l'approche proposé

AOCF part du principe qu'un système est formé de ***composants métiers***, *de* ***composants aspects***. Alors que les composants métiers ont pour but d'assurer les fonctionnalités métiers du système, les composants aspects assurent les services extra fonctionnels.

Le rôle principal de notre framework est l'implémentation des éléments d'une architecture logicielle à savoir les composants et la gestion de leur combinaison (l'interconnexion) et l'injection des aspects sur un ensemble de points de jonctions défini par une coupure.

L'intégration de la programmation orienté aspect (AOP) avec les composants logiciels revient à revoir les concepts liés au paradigme aspect dans un contexte où les applications sont constituées de composants. Ceci consiste à:

1. Définir des points de jonction (points particuliers du flot d'exécution) pour les composants et la façon permettent de les identifier (coupe).
2. Définir une structure d'aspect comprenant des traitements *(Advices)* et des règles de tissage des aspects avec les composants (*before, after et around),*
3. Définir le mécanisme de tissage des aspects avec les composants. Au-delà de la simple intégration de l'application, ce mécanisme doit gérer la multiplicité des aspects liés à un même composant.

Dans ce qui suit nous présenterons ces trois points selon notre approche, nous commencerons premièrement par le modèle de composant d'AOCF et nous détaillerons les concepts sur lesquels est basée les fonctionnalités fournis par ce dernier.

IV.2. Le Modèle de composant en AOCF :

Notre approche impose son propre modèle de composant, nous choisissons de construire un modèle qui *unifie* l'approche à composants et l'approche par aspects, l'avantage de cette unification entre les aspects et les composants réside dans la simplification des interactions entre les composants et les aspects ainsi que leur évolution.

Ce modèle décrit un système comme une configuration de composants communiquant. Chaque composant possède un ensemble de ports requis et un ensemble de ports fournis qui sont défini à l'aide d'une interface java ordinaire, chaque port requis d'un composant possède un ensemble des références vers les composants qu'ils le fournissent des services requis.

Notre modèle introduit la notion de composants ***hiérarchiques*** : composants qui se composent eux-mêmes d'autres composants. Nous appellerons *composite* ce type de composant.

Un composite comprends deux parties:

- **La partie des composants métiers :** Regroupe les opérations principales pour le fonctionnement de l'application.

- **La partie des composants aspects :** C'est la partie qui possède des fonctionnalités techniques, pour séparer les définitions de points de coupure et de traitement des aspects (***symétrie de placement***) les points de coupes sont des éléments de connecteur d'aspect (Aspect Assembling), ce connecteur permet de tisser les codes des greffons dans les composants qui conviennent.

IV.3. Déploiement des composants en AOCF:

La spécification du déploiement consiste à définir pour un type de composant des cas de déploiement et des plans de déploiement. Les cas de déploiement renseigne sur les formes réelles que le composant pourra avoir une fois en exécution (i.e. tache, processus). Le plan de déploiement indique comment un cas de déploiement est appliqué (i.e. tache dans un processus fonctionnant sur la machine locale). **[Bennouar, 2009].**

Nous identifions les cas de déploiement: *PROCESS, MAINTHREAD, THREAD.* Chaque cas de déploiement doit être spécifié avec son environnement. A titre d'exemple, pour le cas *PROCESS*, il est nécessaire d'indiquer l'adresse Internet de la machine et le port TCP.

AOCF définit les trois types de composants suivants chaque type définis des cas et des plans de déploiement particulier:

- **Component:** sera un composant qui sera déployé comme faisant partie de *MAINTHREAD.*
- **TComponent:** qui sera un *THREAD* dans l'application en cours.
- **PComponent:** qui sera un processus à part.

En AOCF, par défaut ***Component*** et ***TComponent*** seront déployé dans l'application en cours, mais on peut déployer des ***Component*** et des ***TComponent*** dans un ***PComponent***.

Lorsque nous demandons la connexion d'un port à un autre port, le connecteur qui sera généré dépendra de la position de chaque composant vis à vis de l'autre (c.-à-d. ça dépend de la distance entre les deux composants).

De par la nature même des composants, le modèle d'AOCF est hiérarchique. Par exemple, un processus (PComponent) peut contenir des processus légers (TComponent).

Par ces cas de déploiement fourni par l'AOCF, nous pouvons développer facilement des applications multithreads et multiprocessus.

IV.4. L'interconnexion entre composants en AOCF:

La communication se fait à l'aide des connecteurs, le type de connecteur dépend de la distance qui sépare les composants. Cette dernière définis selon le cas de déploiement des deux composants connectés.

IV.4.1. Concept de distance entre composants :

Lorsque deux instances de composants sont connectées, le connecteur qui les relie ne sera pas le même pour tous les cas de déploiement des deux instances. Dépendant du cas de déploiement associé à chaque instance, une distance sépare les composants connectés. Selon cette distance le connecteur aura une structure particulière.

Selon les cas de déploiement associés aux instances connectés, la distance devient plus ou moins importante, nous exploitons les types des distances suivants définis dans **[Bennouar, 2009]** pour définir nos modèles de connecteurs supportés par notre Framework AOCF:

- ***La distance 'directe'*** : La distance est dite directe lorsque les deux instances opèrent dans une même tâche.
- ***La distance Locale*** : La distance est dite locale si les ports connectés appartiennent à des composants qui évoluent dans deux taches différentes d'un même processus.
- ***La distance Étendue***: Elle est étendue lorsque les deux composants appartiennent à des processus différents, qui évoluent dans un même environnement (même système d'exploitation, même serveur d'application) ou sur des machines différents.

Le tableau suivant résume les cas de déploiement, cas de distances entres les composants et les types de connecteurs corresponds dans AOCF:

Déploiement	Distance	Type de connecteur
Main thread (MAINTHREAD)	Directe	Connecteur Direct→Appel directe de méthodes
Thread (THREAD)	Locale	Connecteur Local→Appel de méthodes avec synchronisation
Process (PROCESS)	Etendue	RMI (Lipe-RMI)

Tableau IV.1: résumé des cas de déploiements/type de connecteurs supporté par AOCF selon la distance entre les composants

- Lorsque la distance est directe, les deux composants opèrent dans une même tâche (cas de déploiement *MAINTHREAD*). Le connecteur est alors dit direct et est réalisé par des techniques locales à la tâche. Ces techniques ne permettent pas d'aller au delà de la tâche. Les techniques les plus usuelles sont l'appel de procédure et les variables partagées, dans notre cas nous utilisons *l'appel direct de méthode*.
- Dans une même tâche, les deux composants ne peuvent pas évoluer en parallèle. Si une ressource (un port) devait être accédée par les divers composants, l'environnement de déploiement par sa nature résous le problème d'accès. Cependant, puisque il est très possible que les composants utilisés, pourraient avoir été développés par des sources différentes, il n'est pas évident que les ports interconnectés soient totalement compatibles du moins au niveau des noms. Une couche d'adaptation devient nécessaire pour représenter un connecteur direct.
- Si les composants sont séparés par une distance locale, les deux composants opèrent dans deux taches différentes (cas de déploiement *THREAD*) dans un même processus. Le connecteur est dit local. L'accès à une ressource partagée (représenté par un port) doit être régulé. Dans ce cas le connecteur local ne sera pas uniquement un simple appel de méthode ou un simple accès à une variable. Une logique de synchronisation des accès doit être implémentée par le connecteur. Une telle logique ne doit pas faire partie du port du composant, car elle deviendrait encombrante et inutile dans le cas ou les deux autres instances des deux composants son connectés par un connecteur direct.

- Lorsque les composants sont distants, il devient nécessaire d'utiliser des mécanismes de communication fournis par des environnements différents. Des environnements distincts peuvent offrir des mécanismes de communication incompatibles. dans notre cas nous utilisons l'API Lipe-RMI [5] pour connecter les composants distants.

IV.4.2. Les connecteurs en AOCF :

Les interactions entre les composants sont matérialisées par la définition de connexions entre leurs ports. Une connexion permet de relier deux ports. Une vérification est faite qu'il y a bien conformité de type et de sens entre les ports connectés.

L'interaction entre les composants peut se faire par l'intermédiaire des connecteurs. Comme noua avons présenté dans la section précédente est que au niveau implémentation, les connecteurs correspondent à un ou plusieurs mécanismes logiciels bien précis (i.e. appel de procédure, mémoire partagée, pipe, socket, etc.) ou à une technologie d'interconnexion dans le contexte d'un protocole standard de communication (FTP, HTTP, SOAP, IIOP).

La technologie d'interconnexion qui sera mise en œuvre dépend des composants à interconnecter, des types de ports, de la distance entre composants déduite des propriétés de déploiement et bien sur des choix technologiques de l'architecte. Dans notre approche pour l'implémentation des connecteurs nous utilisons trois mécanismes d'interactions qui sont :

- *Appel direct de service (méthode);*
- *Appel direct avec synchronisation;*
- *Appel distant (par l'utilisation de Lipe-RMI);*

IV.4.2.1. Appel direct de méthode :

Dans ce cas l'appelant et l'appelé sont dans un même processus et dans une même tache. Dans un environnement local, les connexions sont souvent réalisées par un appel de méthode.

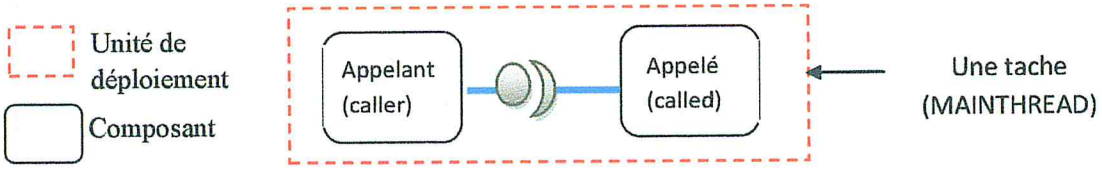


Figure IV.2: connexion par appel direct de méthodes

Cette communication est fiable, lorsqu'un message est envoyé du côté d'un appelant sa réception est toujours garantie du côté de l'appelé. C'est une communication simple, point-à-point. Nous désignerons ce type d'interaction comme une connexion *simple.*

IV.4.2.2. Appel direct avec synchronisation :

Les composants en communication sont dans des taches distinctes dans un même processus

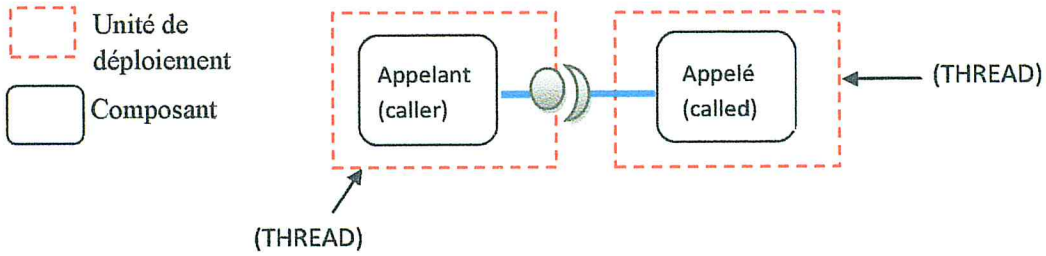


Figure IV.3: connexion par appel de méthodes avec synchronisation

Ce type de connecteur implémente un logique de synchronisation *'synchronisation de threads'*. Ce mécanisme du langage Java permet d'organiser les threads de manière à ce que, pour certaines parties du programme, plusieurs threads ne soient pas en même temps en train d'exécuter ces parties de programme.

IV.4.2.3. Appel distant par Lipe-RMI:

Les deux composants en communication sont dans des processus distant. L'interaction entre le client et le serveur se fait généralement comme un appel de méthode à distance. Ici nous utiliserons Lipe-RMI.

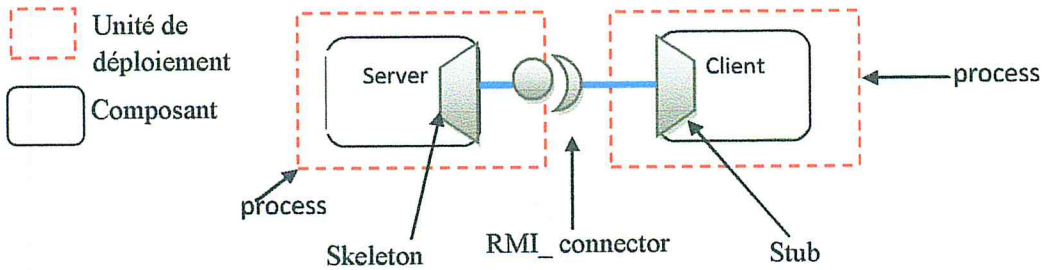


Figure IV.4: connexion par appel distant

- RMI (Remote Method Invocation, invocation de méthodes distantes): Spécialement pour traiter les communications entre les applications JAVA, est une interface de programmation (API) pour le langage Java qui permet d'appeler des méthodes distantes, sur le principe des ORB[3].
  Selon la terminologie RMI, l'objet dont la méthode effectue un appel à distance est appelé l'*objet de client*. L'objet distant est appelé l'*objet de serveur*. Le composant qui exécute le code Java qui appelle la méthode distante est le client pour cet appel et le composant qui héberge l'objet qui traite cet appel est le serveur pour cet appel.

- **Lipe-RMI:** Light Weight Internet Approach for Remote Method Invocation **[5]**: est une nouvelle implémentation de RMI (la première version en 2006 et la version disponible actuellement c'est 0.4.2), elle est totalement indépendamment de RMI et permet les appels par internet, cette API a été conçu pour remédier aux problèmes vue dans RMI ordinaire, le programmeur n'a pas besoin d'un compilateur comme ***rmic***. Le code stub est dynamiquement produit par Lipe-RMI. Puisque Lipe-RMI a été conçu pour travailler dans un environnement internet, les services *Namming et Registry* n'ont pas beaucoup de sens et jusqu'à présent n'ont pas été mis en œuvre par Lipe-RMI, la seule façon d'atteindre un serveur est de connaitre leur adresse IP ou bien le nom de l'hôte. (pour plus de détails sur cette API voir Annexe A).

IV.5. La partie aspect en AOCF:

Dans cette section nous présentons la partie aspect, et comment notre Framework se chargera de tisser/Détisser les aspects dynamiquement.

IV.5.1. Les composants aspects en AOCF :

Un aspect représente la définition d'une préoccupation transverse. Un aspect est définit dans AOCF par héritage de la classe '***Component***' de manière uniforme aux composants métiers, ces composants définissent exclusivement et uniquement les

---

[3] Un ORB est un ensemble de fonctions (classes Java, bibliothèques C++...) qui implémentent un « bus logiciel » par lequel des objets envoient et reçoivent des requêtes et des réponses, de manière transparente et portable : il s'agit de l'activation ou de l'invocation à distance par un objet, d'une méthode d'un autre objet distribué - en pratique les objets invoqués sont souvent des services.

traitements relatifs à un et un seul aspect, ils sont génériques et donc réutilisable, il peut être définit à l'aide d'un composant primitif ou un composite. Nous disposons trois manières d'associer un advice avec un point de jonction: Avant (before), après (after), autour (around).

- ❖ L'advice de type ***Avant (Before)*** est exécuté juste avant l'exécution du point de jonction.
- ❖ L'advice de type ***Après (After):*** est exécuté juste après l'exécution du point de jonction.
- ❖ L'advice de type ***Autour (Around):*** un advice de type 'Around' encapsule un point de jonction.

IV.5.2 Les points de jonction d'AOCF :

Dans AOCF, toutes les ressources situées au niveau d'un port sont considéré comme des points de jonction potentiels.

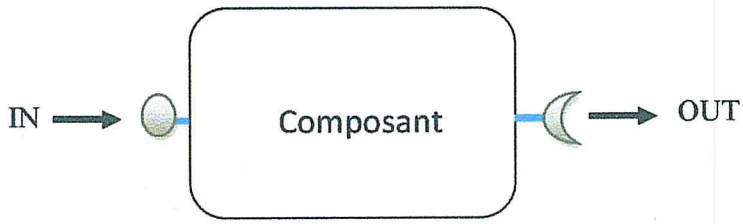


Figure IV.5: Le modèle de point de jonction en AOCF

Ce choix est motivé par le fait que nous considérons que l'approche AOSD se trouve dans un monde de composant. Puisque les composants sont des boîtes noires, il est assez naturel de considérer les points de jonctions que sur les éléments visibles de l'extérieur i.e. les ports requis et fournis. De cette manière notre approche ne briserait pas l'encapsulation des composants.

Dans **AOCF** l'injection effective d'aspect, ou tissage a toujours lieu au niveau des interfaces requises.

IV.5.3 Les points de coupure (pointcut) d'AOCF :

Une coupe est un regroupement de points de jonctions qui permet de précise l'endroit où les greffons (Advices) seront insérés.

Le langage de points de coupure que nous utilisons pour sélectionner les points de jonction est basé sur une expression pointcut spécifique aux composants qui définissent des points dans les architectures à composants, elle est sous forme une expression régulière composée de trois parties qui affectent le nom des composants (le nom de la classe et le nom des objets), les noms d'interfaces (port) et les signatures de méthodes séparées par des slaches "/" (figure IV.6) :

pointCut:= NomDeClasseDeComposant: NomDeInstanceDeComposant/ NomDePort /NomDeMethode

Figure IV.6: La grammaire du langage de pointcut en AOCF

- ✓ **NomDeClasseDeComposant:** si c'est *, ce sont toutes les composants du composite.
- ✓ **NomDeInstanceDeComposant:** c'est le nom de l'instance de composant, si c'est * ce sont tous les instances du composant.
- ✓ **Exemple:** `pointCut = "Compute:.*/Iop/add(int):int"` ; Dans cet exemple tous les appels de la méthode 'add' sur l'interface 'Iop' à partir de tous instances du composant dont le nom de la classe est 'Compute'.

IV.5.4 Tisseur d'aspects introduit par AOCF:

Notre Framework enrichit un assemblage initial de composants métiers avec un assemblage implémentant des préoccupations transverses. En bref, il est possible de définir des mécanismes de tissages. A ce stade nous allons besoin de caractériser en plus la notion d'aspect qui incarne une préoccupation transverse aussi un mécanisme de tissage de ces préoccupations transverses à la partie métier.

Comme nous avons indiqué dans la section précédente que le tissage d'aspects avec notre AOCF se fait au niveau de composant client, si le pointcut concerne un composant sercveur l'aspect corresponds doit être appliqué à tous ces composants clients connecté sur le port indiqué dans le pointcut.

Nous prenons l'exemple illustré dans la figure (IV.7.a) un composite contient trois sous composants (server1 instance du composant 'Server', client1 et client2 qui sont des instances du composant 'Client'), les composant '*client1*' et '*client2*' sont connectés avec le composant 'server1' via son interface 'IS ' qui dispose un service m1.

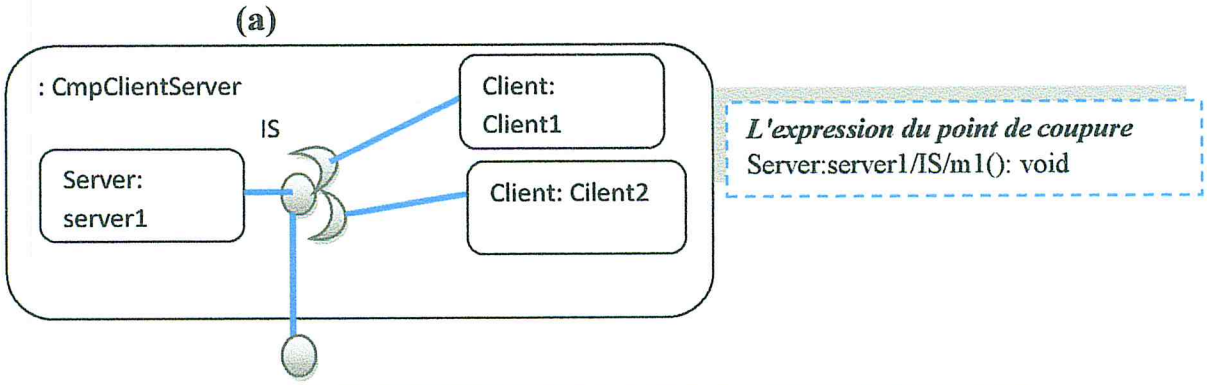


Figure IV.7: Exemple du pointcut en AOCF (b) les composant à aspectiser (a)

Si on a un aspect 'A1' qu'il doit être tissé au niveau du composite '*CmpClientServer*' selon le point de coupure définis en (IV.1.b), cet aspect sera donc tisser sur tous les composant clients connectés sur l'interface 'IS' et qui demande le service 'm1' dans notre exemple ce sont les composant '*Client1*' et '*Client2*'.

Notre modèle de tissage utilise le mécanisme de réflexion de JAVA pour détecter quelles méthodes qui sont actuellement besoin d'être greffé.

Une méthode définie par notre Framework (la méthode ***weave***) permet de trouver toutes les interfaces et méthodes qui respectent la coupe définie et tisser les greffons (advices) correspond selon son type (before, after, around).

Pour réaliser cette insertion nous utilisons l'API Javassist [3], Il s'agit d'un API pour la manipulation du bytecode Java, il permet aux programmes Java de définir une nouvelle classe à l'exécution et de modifier un fichier de classe avant leur chargement par le JVM. (pour plus de détails sur cet API voir annexe B).

IV.5.4.1. Utilisation de l'API javassist dans le tisseur AOCF:

Notre utilisation de cet API concerne seulement la création dynamique des classes java, implémentation dynamique des interfaces et l'ajout des nouvelles méthodes au niveau de cette classe.

Dans un premier temps grâce à la méthode "***weave*** " introduit par le framework, qui se chargera premièrement d'extraire à partir de l'expression qui représente le point de coupure la liste des composants concerné par l'aspect courant, un composant pour chaque un sera créé dynamiquement par l'AOCF, un composant c.-à-.d une classe java créé dynamiquement grâce à l'API javassist qui implémente une interface et étends de la classe ***Component,*** ce composant créé dynamiquement sera une sorte de proxy, et les appelles de méthode sera rediriger vers ce proxy (figure IV.8) illustre ce logique:

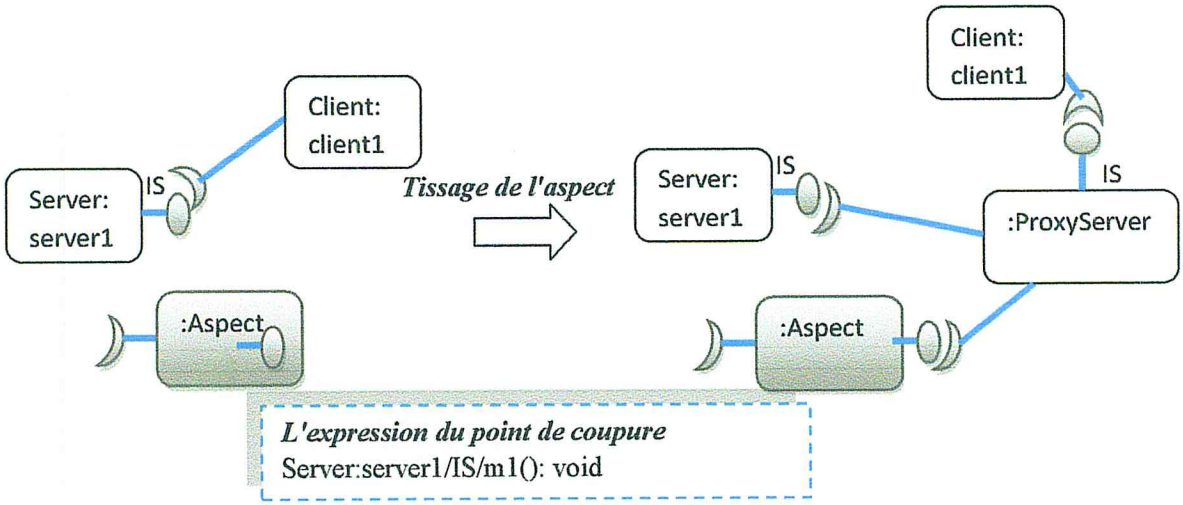


Figure IV.8: principe du tissage en AOCF- greffons de type After ou before

Dans cet exemple (IV.8), l'expression du point de coupure contient l'instance 'server1' du composant 'Server' et son interface fournis 'IS', donc l'aspect sera tisser sur tous les composant clients connectés avec 'server1' sur l'interface 'IS', le composant 'ProxyServer' c'est celle qu'est créé dynamiquement par l'AOCF après le tissage l'appelle de la méthode *m1* par *client1* suivre les étapes suivants:

1. *Client1* appelle la méthode *m1*.
2. L'appelle rediriger vers *'ProxyServer'* .
3. *'ProxyServer'* de son tours redirige l'appelle vers *'server1'* ensuite *'Aspect'* si l'advice est de type *'after'* ou dans le sens contraire dans le cas où l'advice est de type *'before'*.

Pour le cas où le type de l'advice est *'around'* la logique de tissage sera un peut changer, le code de l'aspect permet de lancer l'exécution de la méthode et ainsi de réaliser des traitements avant, pour par exemple conditionner l'invocation de la méthode et des traitements après.

IV.5.5 Tissage de plusieurs composants d'aspects:

Lorsque plusieurs composants d'aspects sont tissés sur le même composant de base, leur ordre d'exécution est celui de leur tissage : i.e. les composants d'aspect tissés en premier sont exécutés en premier. A titre de travail futur, nous prévoyons de compléter notre prototype pour permettre 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 ou JAC).

IV.6. Les contraintes d'interconnexion des composants en AOCF:

Notre Framework intègre aussi la vérification la spécification de connexion et l'insertion d'aspect. Par exemple si on veut lis un port requis à un composant (donc à une interface fournie de ce composant), il faut que ce composant dispose de cette interface fournie, sinon la connexion ne serait pas correcte.

Les interfaces des composants connectés doivent disposer du même nom et des mêmes opérations, les opérations (les méthodes) doivent être compatible : même nom, même type de retour et même type de paramètres. Cependant, puisque il est très possible que les composants utilisés, pourraient avoir été développés par des sources différentes, il n'est pas évident que les ports interconnectés soient totalement compatibles du moins au niveau des noms. A titre de travail future , nous prévoyons ajouter à notre AOCF une couche d'adaptation d'interfaces (ports liées aux composants connectés) pour représenter un connecteur direct, avec une indication de correspondance des méthodes c.-à-d. d'indiquer la méthode x d'une interface elle est correspond à la méthode y de l'autre interface.

Autre contrainte d'interconnexion notre AOCF interdit les appels entre deux sous composants dans deux composites différents.

IV.3. Architecture du Framework:

Dans cette partie nous allons résumer tous ce que nous avons précité dans ce chapitre par un schéma décrivant l'architecture de notre Framework.

La figure IV.9 décrit le schéma globale de notre AOCF par une représentation UML2.0

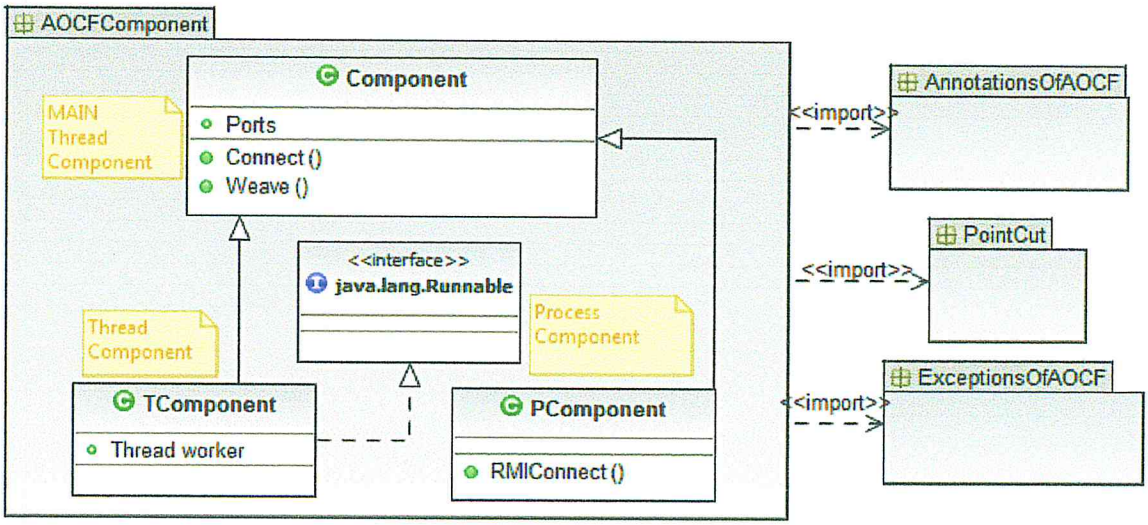


Figure IV.9: Architecture du Framework AOCF par UML2.0

Notre AOCF contenant un ensemble de classes assurant les fonctionnalités citées ci-dessus. Il contient entre autres une *classe* '***component***'.

Force est de constater que tous les éléments du système quelque soit composants métiers ou composants aspects sont reliés au Framework (par héritage) voir figure (IV.5), en effet, ce dernier gère l'ensemble des interactions entre composants métier et composants métiers grâce à la méthode ***connect***, puis entre composants aspects et composants métiers grâce à la méthode ***weave***, et enfin celles liant composants aspects et composants aspects.

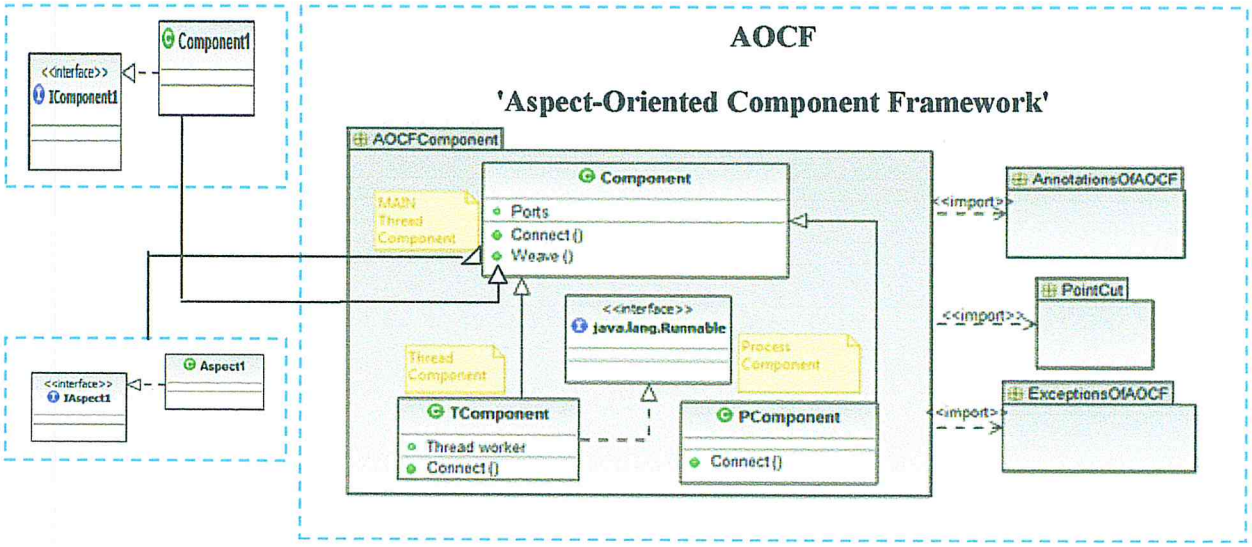


Figure IV.10: Définition des composants métiers/aspects en AOCF

Le rôle du framework dans une application à base des composants est primordial, il assure les la transformation d'une architecture logicielle vers l'implémentation en langage JAVA, Il gère toutes les liaisons pouvant exister entre les différents éléments du système ; entre un composant aspect et un composant métier, entre les composants métiers, et entre les composants aspects, il permet aussi l'ajout/suppression dynamiques des composants et aspects durant l'exécution.

IV.4. Les modules ouverts et AOCF:

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

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

IV.4.1. Principe des modèles ouverts:

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

Un point de jonction devient accessible lorsqu'il est déclaré comme point d'accès d'un module. L'interface d'un module peut donc déclarer : des points de jonction, des fonctions, ou des coupes prédéfinies sur un ensemble d'éléments internes. Un module vérifie les propriétés définies ci dessous, que nous établissons comme règles pour pouvoir s'y référer plus facilement dans la suite de notre discussion.

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

Dans des travaux plus récents, les principes gouvernant les modules ouverts ont été appliqués à AspectJ [Ongkingco et al., 2006]. Dans leur étude, les auteurs ont ajoutés de nouveaux concepts, qui sont souvent assez spécifique à AspectJ. Cependant, nous

en avons relevé deux qui nous semblent généralisables à notre application des modules ouverts aux composants. Le premier concept intéressant est la possibilité de ne pas seulement réduire et contrainte l'accès à un module, mais de pouvoir moduler cet accès : pouvoir ouvrir et fermer l'accès de certains points. Ceci peut être particulièrement utile dans un scenario d'aspect de déverminage. Il peut ainsi être intéressant de fermer certains accès après cette période de déverminage. Un second concept intéressant de cette étude est la possibilité de désigner les aspects pouvant accéder l'interface du module.

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

IV.4.2. Support des règles des modules ouvert en AOCF:

La première similitude frappante réside dans la notion de module et celle de composant. Un module est une collection de classes qui partagent des points d'accès aux aspects par une interface. Un composant est une entité contractuelle qui fournit et requiert des services par le biais de ports (ie.interfaces). Un composant est une «boîte noire» qui naturellement cache ses détails d'implantation comme le définit la Règle 3 des modules ouverts. Dans AOCF les points de jonction sont des éléments de ports externes (fournies et requises) des composants ouverts. Cette définition s'applique parfaitement aux définitions des Règles 1, 2 et 3. La Règle 2 correspond à la définition même de notre modèle de point de jonction, les points d'accès sont accessible par les aspects. La Règle 3 interdit l'interception des appels internes, ce qui est le cas également avec AOCF.

La création des composants qui sont sorte de proxy comme nous avons indiqué dans la section (IV.5.4.1) permet de préserver le comportement interne des composants et le verrouillage de l'accès aux opérations.

IV.5. Caractéristique de l'AOCF :

- AOCF est une cible d'implémentation des spécifications ADL, il permet de définir un composant et ces ports, interconnecter les composants et ajouter des composants dans un composant qu'est alors un composite pour construire une hiérarchie de composants.

AOCF permet entre autre d'implémenter facilement un processus d'élaboration d'application homogène par raffinement successifs. Les ADL actuels, passent à un moment donné à un autre modèle que le modèle de composant lors de la transformation ADL vers une cible (langage de programmation ou framework)

- D'autre part AOCF permet la programmation orientée composant aspect, il propose un modèle d'adaptation logicielle en combinant CBSD (component-based software development) et AOSD (Aspect-Oriented Software Development) qui sont deux approches complémentaires, afin d'augmenter la modularité et l'évolution des systèmes.
- Notre 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 (Aspect Oriented Programming) dans COP (Component Oriented Programming). Le terme sûr est pris dans le sens où l'intrusion d'AOP est finalement gérée au niveau de chaque composant.
- Notre approche est symétrique, en d'autres termes elle considère les aspects et les composants comme des entités uniformes.
- Elle permet au système de s'adapter au contexte et à l'environnement, en permettant l'ajout et la suppression des aspects au moment de l'exécution. Cette caractéristique introduit plus de souplesse dans la gestion des applications.

IV.5. Les apports de l'approche AOCF :

- Il considère les aspects et les composants comme des entités uniformes (approche symétrique), il sépare explicitement la définition des points de coupe et de traitement des aspects ce qui permet un haut niveau de réutilisation pour les aspects comme pour les composants. En outre, les aspects du système ne perdent pas leurs identités ce qui facilite grandement la maintenance et favorise l'évolution du système.
- Le développement du système est relativement facile, car le développeur se concentre seulement sur la sélection des composants et des aspects.

- Pour tout résumer, notre approche de combinaison cherche à rassembler à la fois, les avantages des travaux existants et les solutions portant remède à leurs insuffisances.

Le tableau suivant (IV.2) résume les contraintes assurées par notre framework qui sont relatif aux concepts de l'architecture logicielle:

critère / *L'approche*	*Héritage de composant*	*La notion d'hiérarchique*	*Le type de communication entre composants*	*La faisabilité de la transformation d'un ADL.*	*Ajout /suppression dynamique des composants et connecteurs*
AOCF	Oui	Oui	Appel de méthodes local, appel de méthodes distantes	Oui	oui

Tableau IV.2: Bilan (1) sur l'AOCF

Le tableau (IV.3) récapitule notre approche selon les critères du point de vue aspect

critère / ***L'approche***	***Tissage / détissage dynamique***	***Symétrie d'élément***	***Symétrie de placement***	***Langage de la coupe***	***Ordre d'exécution des greffons***
AOCF	Oui	Oui	Oui	Oui	Selon leur ordre de tissage

Tableau IV.3: Bilan (2) sur l'AOCF

Conclusion :

Dans ce chapitre nous avons présenté les concepts fondamentaux de notre approche proposée Aspect-Oriented Component Framework **AOCF** un Framework pour la conception orienté composant qui unifié les deux approches composant/aspect qui est un cible pour la transformation d'un ADL vers l'implémentation. Nous avons essayé de fonder toute notre action sur une approche qui consiste à rassembler les avantages des travaux existants tout en tentant de remédier à leurs insuffisances.

Le chapitre suivant représente le détail d'implémentation notre Framework en JAVA et quelques tests pour valider notre réalisation.

Chapitre V: Implémentation et Tests

Introduction :

Après avoir présenté dans le chapitre précédent les concepts fondamentaux de notre proposition AOCF (Aspect-Oriented Component Framework), nous arrivons dans ce chapitre à présenter les détails d'implémentation de notre Framework en JAVA avec quelques tests sur des applications java.

Nous commencerons dans un premier temps par la présentation de notre environnement de travail, ensuite nous passons vers le détail de démarche d'implantation et quelques tests pour valider noter approche.

V.1. L'environnement de développement:

Pour l'implantation de notre approche, nous avons opté pour la plateforme éclipse [4], connue pour ses principes de modularité et d'extensibilité, mis en pratique par le concept de plug-ins.

La liste suivante indique clairement l'environnement de développement de notre framework AOCF:

- **Système d'exploitation:** Windows 7 Edition Familiale Premium.
- **Type de Système:** Système d'exploitation 64 bits.
- **Langage de programmation:** JAVA (JDK 7, et l'IDE Eclipse 3.7.1 indigo).

V.1.1. Les bibliothèques ajoutées:

Nous avons ajouté à notre environnement de travail une bibliothèque pour l'API ***javassist*** (version 3.18.0)[3] qui nous aide pour réaliser la partie aspect de notre framework, En effet pour faire supporter l'orienté aspect à notre framework, il est nécessaire d'intervenir au niveau Byte Code Java, Nous devons en effet créer à la volée, durant l'execution, de nouvelles classes puis les compiler et ensuite les intégrer dynamiquement a notre programme. Cette opération est en elle-même un projet à long terme. De ce fait nous avions cherché des outils orientés manipulation du bytecode java. Après une brève recherche, nous avons déterminé l'existence de plusieurs outils, plus exactement des API, tels que JAVASSIST [3], ASM[6] et BCEL [5]. Notre choix s'est porté sur JAVASSIST. Ce choix est motivé d'une part par la souplesse d'utilisation de JAVASSIST et d'autre part par l'existence de versions plus récentes

que les autres. A titre d'exemple la dernière version de JAVASSIST date de Juin 2012, alors que celle de BCEL et ASM datent respectivement de 2009 et de 2010.

Nous avons aussi ajouté la bibliothèque LIPE-RMI [2] qu'est une nouvelle implémentation de RMI ordinaire, notre utilisation de cette bibliothèque est pour réaliser les connecteurs entre les composants distants situés sur deux JVMs différents.

V.2. Les package AOCF:

Afin de bien organiser les fonctionnalités fournis par notre framework, nous utilisons les packages. Chaque package contient un ensemble de classes qui participent à une tache donnée: tous les éléments d'un même package doivent donc avoir la même finalité:

- ***Le package "aocf.component"*** c'est le package métier de notre framework contient les classe 'Component' , 'TComponent' et 'PComponent'.
- ***Le package "aocf.annotation":*** contenant tous les annotations fournis par AOCF:

  Par exemple l'annotation ***@Requires*** pour annoter les méthodes requis par un composant client.
- ***Le package "aocf.exception" :*** contient l'ensemble des exceptions généré par l'AOCF qui concerne les fautes de conception faite par le développeur.

V.3. L'implémentation d'un ADL à l'aide d'AOCF:

- Dans un premier temps le concepteur défini une architecture en instanciant un ensemble de type de composants qu'ils connecte par la suite.
- Un composant est une classe java ordinaire qui étends (héritage) l'une des classes suivantes de AOCF
  - La classe ***'Component'*** s'il est déployable comme faisant partie intégrante du processus représentant l'architecture en cours de réalisation. Nous disons qu'il fait partie du ***MainThread,***
  - La classe ***'TComponent'*** si le composant représente une tache (thread) intégrante du processus représentant l'architecture en cours de réalisation. Dans ce contexte l'architecture est multithread (multitache)

- La classe ***PComponent*** si le composant représente processus totalement indépendant du processus représentant l'architecture en cours de réalisation. Dans ce contexte l'architecture est dite distribuée ou multiprocessus.

- Les interfaces/ports fournis d'un composant sont définis à l'aide des interfaces java implanté dans le code du composant.
- Le service requis d'un composant à l'aide d'une annotation **@Requires**.
- Pour la connexion entre les composants, AOCF fournit deux méthodes:
  - **connect**(Object CmpServer, String port, String method);
  - **connect**(Object CmpServer, String port)**:** dans ce cas tous les methods seront connectés entre elles.
- Pour la déconnection d'un client à un serveur, AOCF fournit trois types de méthodes:
  - **disconnect**(Object ConnectedCmp)**:** pour se déconnecter d'un composant.
  - **disconnect**(Object ConnectedCmp, String portName)**:** pour se déconnecter d'une interface.
  - **disconnect**(Object ConnectedCmp, String portName, String methodName): pour se déconnecter d'une méthode.
- Pour la connexion entre deux composants situés sur deux JVM différentes les méthodes de connexion deviendront prend comme paramètres au lieu la référence de l'objet le nom de la machine distante ou bien l'adresse IP et un numéro de port TCP par défaut comme suit :
  - **connect**(String Remote_Machine_Name, int PortTCP, String interface, String method);
  - **connect**(String Remote_Machine_Name, int PortTCP, String interface).
- Aussi pour la déconnection d'un client à un serveur distant, on a les trois méthodes suivantes:
  - **disconnect**(String Remote_Machine).
  - **disconnect**(String Remote_Machine, String portName.
  - **disconnect**(String Remote_Machine, String portName, String methodName).

- Le tissage des aspects en AOCF se fait par la méthode *'weave'* sur le composite qui prends comme paramètre: l'expression du point de coupure, le référence du composant aspect, le nom de l'advice et le type de l'advice (before, after ou around):

```
weave (String pointCut, Component aspect, String advice, String advice_type);
```

Figure V.1: la méthode weave de l'AOCF

- Le détissage se faite par la méthode *'Unweave'* qui prends comme paramètre la référence du composant aspect et l'expression du point de coupure et le type:

```
unweave (String pointCut, Component aspect, String advice_type);
```

Figure V.2: La méthode Unweave de l'AOCF

V.4. Les Tests:

Dans cette section nous essayons de présenter quelques exemples d'implémentation des architectures logicielle à base des composants par l'utilisation de notre framework, nous prendrons comme exemple illustrative l'architecture de l'application qui permet de calculer l'équation mathématique: s=cos(x) + sin(y) + tan(z). Quatre composants nécessaire pour réaliser le composite qui représente cet application (un composant qui fait le calcule du cosinus, un composant qui fait le calcule du sinus, un composant qui fait le calcule du tangente et un composant qui fait la somme) :

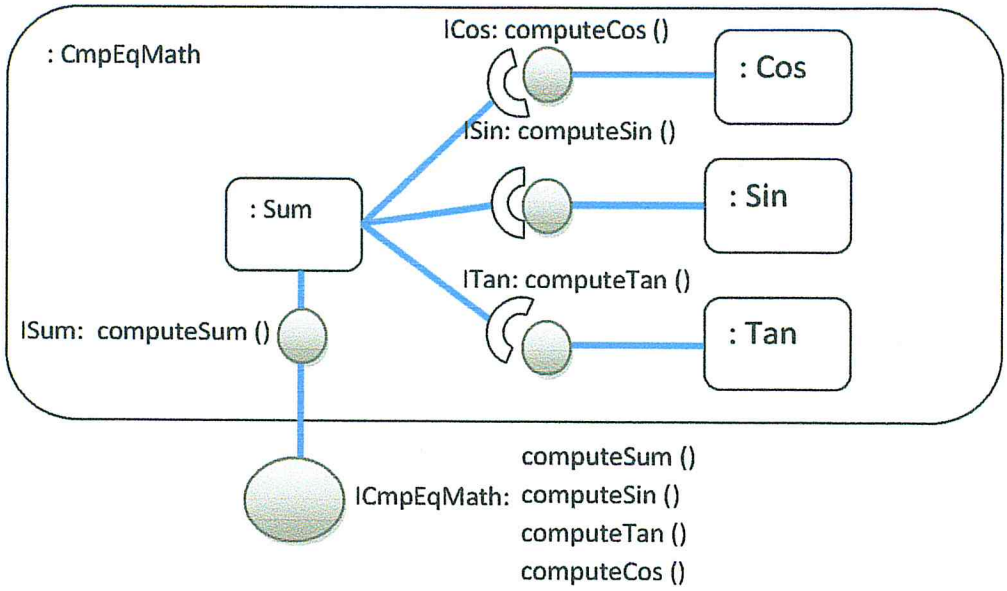


Figure V.3: architecture de composite équation mathématique

Par la suite, nous considérons 3 cas d'études:

- Dans le premier cas d'étude, tous les composants (:cos, :sin, :tan et :sum) sont déployés comme partie intégrante du **mainThread.** Dans ce cas, ces composants doivent être des ***Componen***t (ils étendent tous la classe '***Component***'). Le type de connecteur utilisé dans ce cas est représenté par un appel simple des méthodes.
- Dans le deuxième cas d'étude, nous considérons que chaque composant est déployé comme une tache (Thread). Dans ce cas les composant sont des '***TComponent***'. Ils étendent la classe ***TComponent***. Le type de connecteur utilisé dans ce cas est représenté par un appel de méthode pouvant être synchronisé dans le cas où le port ou un de ses points d'accès (les méthodes) sont déclarés comme ressource critique.
- Dans le dernier cas d'étude, nous considérons que les composants sont distribués sur des machines distinctes. Dans ce cas les composant sont dit des ***PComponent*** car ils étendent la classe '***PComponent***'. Le type de connecteur utilisé dans ce cas est représenté par un appel distant de méthode.

V.4.1. Le cas de déploiement 'MAINTHREAD' :

Tous les composants définis dans cette application soit les primitives soit les composites sont représentés par une classe JAVA étends de la classe **"Component"** définis par le framework AOCF;

```
import aocf.component.Component;

public class Sum extends Component implements ISum {

	ICos cos;  ISin sin;  ITan tan;

		// constructors
	// provides methods
	@Override
	public double Sum(double c, double s, double t) {
		double s1=(double) this.getConnectedComponent("ISin","computeSin")
					.call("computeSin", c);
		double c1=(double)this.getConnectedComponent("ICos","computeCos")
					.call("computeCos", s);
		double t1=(double) this.getConnectedComponent("ITan","computeTan")
					.call("computeTan", t);
		return (s1+c1+t1);
	}	}
	@Requires
	public double computeSin(double deg) {
		return 0;
	}
	@Requires
	public double computeCos(double deg) {
		return 0;
	}
	@Requires
	public double computeTan(double deg) {
		return 0;
	}
```

Listing V.4: exemple d'implémentation d'un composant possède des services requis à l'aide de l'AOCF

La figure (V.5) représente un code java pour le composant composite **":CmpEqMath"** par l'utilisation de l'AOCF le composant doit être étends de la classe 'Component' définis par notre framework, puisque **":CmpEqMath"** est un composite, il doit faire instancié tous ces inter-composants et de les connecter par la méthode **"connect (Object cmp, String port, String method)"** :

```
import aocf.component.Component;

public class CmpEqMath extends Component implements ICmpEqMath {

	Cos cos;  Sin sin;  Tan tan; Sum sum;
	public CmpEqMath () {
		super();
		// TODO Auto-generated constructor stub
		cos=new Cos(this,"cos1");
		sin=new Sin(this,"sin1");
		tan=new Tan(this,"tan1");
		sum=new Sum(this,"sum1");
		sum.Connect("ISin","computeSin", sin);
		sum.Connect("ICos","computeCos", cos);
		sum.Connect("ITan","computeTan", tan);
		double s=sum.ComputeSum(30, 30, 30);
		System.out.println("sum of cos 30 sin 30 tan 30= "+s);
	}
	public MathEqComposite(Component c, String name) {
		super(c, name);
		// TODO Auto-generated constructor stub
	}
	@Override
	public double Sum(double c, double s, double t) {
		return this.sum.ComputeSum(c, s, t);
	}
	@Override
	public double computeSin(double deg) {
		return this.sin.computeSin(deg);
	}
	@Override
	public double computeCos(double deg) {
		return this.cos.computeCos(deg);
	}
	@Override
	public double computeTan(double deg) {
		return this.tan.computeTan(deg);
	}
```

Figure V.5: implémentation du composite 'CmpEqMath' à l'aide de l'AOCF

Le résultat de l'exécution donne:

```
calling method ComputeSum
calling method ComputeSin
calling method ComputeCos
calling method ComputeTan
sum of cos 30 sin 30 tan 30= 1.9433756729740643
```

Chaque composant pourrai être réutilisé dans une autre application, et aussi pourrai être hérité d'un autre composant ou bien une classe objet normal. Les sous classes du

composant hérite les méthodes, les interfaces et peuvent aussi définit des nouvelle méthodes et interfaces, le concept d'héritage augmente encore plus la réutilisabilité.

V.4.1.1. Exemple d'ajout d'un aspect:

Quand nous voudrons ajouter un composants spécial (aspect) à une application à composant, AOCF fournit des services pour tisser/détisser des aspect dynamiquement (au moment de l'exécution), le tissage se fait à l'aide de la méthode **"weave"** qui prend en paramètre: l'advice, type de l'advice (before, after ou around) et l'expression de point de coupure:

L'expression de point de coupure doit être conforme à notre modèle définis par AOCF (voir l'exemple (figure V.6) de pointCut en AOCF):

```
"Cos:.*/ICos/ComputeCos(double):double"
```

FigureV.6: exemple d'un point de coupure conforme au modèle de pointCut en AOCF

Notre modèle du point de coupure est défini par trois expressions régulières séparés par slache '/', la première expression pour la partie composant (contient le nom de la classe et le nom de l'objet), dans l'exemple (V.6) cette expression contient la méthode `"computeCos"` sur `"ICos"` de tous les objets du composant `"Cos"`.

Le tisseur d'aspects définis par notre AOCF suit les étapes suivantes:

a) Extraire à partir du point de coupure la liste des composants qui possède les interfaces fournies les services (méthodes) définis dans le point de coupure.
b) Extraire la liste des composants à aspectiser (ce sont celles qui requises les services définies dans le point de coupure).
c) Pour chaque composant 'c' du (a), le tisseur d'AOCF va créer un composant proxy dynamiquement qui fournis les mêmes services du composant 'C', et au niveau de ce proxy l'injection des greffons sera faite.
d) Pour chaque appel d'un service à partir d'un composant du (b), sera redirigé vers le représentant (proxy) créé dans l'étape (c) du composant qui fournis le service demandé.

Nous prenons un exemple d'un composant aspect (*Logger*) qui permet de faire le trace d'exécution d'une méthode selon, le point de coupure définis dans (V.8). Puisque notre approche est symétrique les deux types de composant soit régulier soit aspect étends de la classe **"Component"** (symétrie d'élément), l'expression de point de coupure est séparé de code advice, il est défini au niveau de connecteur transverse (la méthode ***weave)***.

L'architecture de la figure (V.3) devient comme indiquer dans la figure (V.7):

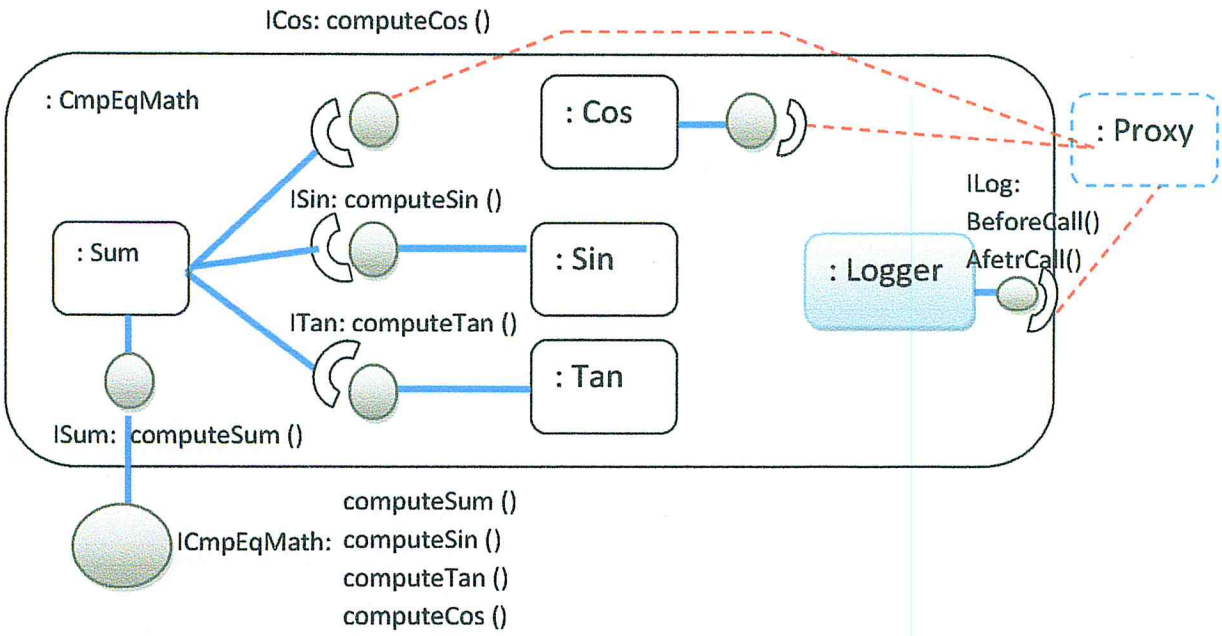


Figure V.7: architecture de composite équation mathématique après le tissage d'un aspect

Le composant 'proxy' est une classe transparente qui n'est pas instancié au niveau du composite, il est créé dynamiquement après l'appel de la méthode ***weave*** contenant l'expression du point de coupure de la figure (V.6).

Le code du composite de la figure (V.7) devient comme suit:

```
import aocf.component.Component;
public class CmpEqMath extends Component implements ICmpEqMath {

	Cos cos;  Sin sin;  Tan tan; Sum sum;
	public CmpEqMath () {
		super();
		cos=new Cos(this,"cos1");
		sin=new Sin(this,"sin1");
		tan=new Tan(this,"tan1");
		sum=new Sum(this,"sum1");

		Logger log=new Logger(this,"log");
		String pointcut="Cos:.*/ICos/ComputeCos(double):double"

		sum.Connect("ISin","computeSin", sin);
		sum.Connect("ICos","computeCos", cos);
		sum.Connect("ITan","computeTan", tan);

		this.weave(pointcut,"before",log,{"BeforeCall"});
		this.weave(pointcut,"after",log,{"AfterCall"});

		double s=sum.ComputeSum(30, 30, 30);
		System.out.println("sum of cos 30 sin 30 tan 30= "+s);
	}
	.....
}
```

Figure V.8: le code du composite 'CmpEqMath' avec l'ajout d'un aspect

Le résultat d'exécution devient comme suit:

```
calling method ComputeSum
calling method ComputeSin
before calling Time= 30362605416132
calling method ComputeCos Time= 30362605518480
after calling Time= 30362605613130
calling method ComputeTan
sum of cos 30 sin 30 tan 30= 1.9433756729740643
```

Pour le cas du greffons de type 'Around', il est nécessaire que la méthode weave d'avoir deux codes un pour avant l'appel et un autre pour après l'appel **exemple:**

```
this.weave(pointcut,"around",log,{"BeforeCall","AfterCall"});
```

V.4.2. Le cas de déploiement "THREAD" : Exemple sur une application multitâche :

On prend le même exemple de la figure (V.3) où chaque composant sera maintenant déployé comme faisant une tache, dans ce cas tous les composants seront étends de la classe '*TComponent*' du framework.

```
import aocf.component.TComponent;

public class Cos extends TComponent implements ICos{


	public Cos(double deg) {
		super();
		// TODO Auto-generated constructor stub
		this.deg=deg;
	}
	@Override
	public double computeCos(double deg) {
		double Deg2Rad = Math.toRadians(deg);
		return Math.cos(Deg2Rad);
	}
	@Override
	public void Main(){
		double res=this.computeCos(deg);
		System.out.println("compute cos by "+
this.thread.getName()+"=>Cos("+deg+")="+res);

	}

}
```

Figure V.9: le code du composant 'Cos' déployé comme Thread

Au niveau de la méthode "Main" défini dans la classe 'TComponent', le travail du composant thread sera définis, dés que la méthode 'start' est appelé sur ce dernier, l'appel sera rediriger vers la méthode 'Main' et le thread (la tache) commence son travail.

Un TComponent client doit être attends son serveur j'usqu'il termine son travail comme le cas de composant définis dans la figure (V.10):

(a)

```
import aocf.annotation.Requires;
import aocf.component.TComponent;

public class Sum extends TComponent implements ISum{

    double sum; double deg1;double deg2;double deg3;
      public Sum(double deg1,double deg2,double deg3) {
            this.deg1=deg1;
            this.deg2=deg2;
            this.deg3=deg3;
      }
      @Override
      public double ComputeSum(double c,double s,double t) {

            Sin sin=(Sin)this.getConnectedComponent("ISin","computeSin");
            Cos cos=(Cos)this.getConnectedComponent("ICos","computeCos");
            Tan tan=(Tan)this.getConnectedComponent("ITan","computeTan");
            double sum=sin.res+cos.res+tan.res;
      return sum;
      }
       @Override
       public void Main(){
          this.sum=this.ComputeSum(deg1, deg2, deg3);
                                          System.out.println("compute Sum by
                                  "+
                                  this.thread.getName()+"=>Cos("+deg1+")+"+"Sin(
                                  "+deg2+")+"+"Tan("+deg3+")="+sum);

       }
}
}
```

(b)

```
import aocf.component.TComponent;


public class EqMath extends TComponent{

  public EqMath () {
   super();
   double deg1 = 20,deg2=30,deg3=60;
   Cos cos=new Cos(deg1);
   Sin sin=new Sin (deg2);
   Tan tan=new Tan(deg3);
   Sum sm=new Sum(deg1, deg2, deg3);
   cos.start();
   sin.start();
   tan.start();
   cos.join();
   sin.join();
   tan.join();
   sm.Connect("ISin","computeSin", sin);
   sm.Connect("ICos","computeCos", cos);
   sm.Connect("ITan","computeTan", tan);
   sm.start();
```

Figure V.10: (a) le code du composant "Sum" (b) le code du composite déployés comme Threads

V.4.3. Le cas de déploiement PROCESS:

On prend l'exemple illustré dans la figure (V.11):

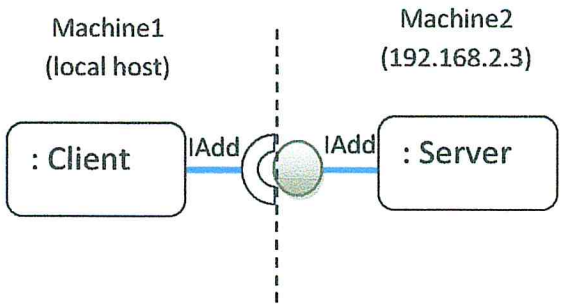


Figure V.11: Architecture de deux composants distants

Le composant 'Server' fournis un service 'Add' sur son interface fournis 'IAdd', la figure (V.12) représente l'implémentation de cette architecture à l'aide de notre framework:

```
public class Server extends PComponent implements IAdd{
      public Server(int port) {
            super(port);
            // TODO Auto-generated constructor stub
      }

      public void add(int a, int b) {
            System.out.println("addition of "+a+"and"+b+"is"+(a*b));

      }

public static void main(String[] args){
      Server server=new Server1(8080);
    }

}
```

```
public class Client extends PComponent{


      public Client1() {

      }

      public static void main(String [] args){
            Client cl=new Client();
            cl.RMIconnect("192.168.2.3", 8080,"IAdd");
            IAdd addimpl=(IAdd) cl.getConnectedServer("192.168.2.3");
            addimpl.add(20, 20);

      }
}
```

Figure V.12: Implémentation deux composants distants grâce à l'AOCF

V.5. Les exceptions gérées par AOCF:

Notre framework indique des exceptions sur les faute de conception, comme l'appel des services vers un composant n'étais pas déjà connecté avec l'appelant; la connexion avec un composant qui ne possède pas les services requis du composant appelant, la non compatibilité entre les services fournis et les services requises et la connexion entre deux composants qui ne sont pas dans le même composite. On prend les exemples suivants:

- Si le composant 'Sum' dans la figure (V.4) appel le service 'ComputeSin' sur l'interface 'ISin' avant qu'il est connecté avec le composant 'Sin' qui fournis ce service l'exception suivante sera généré:

```
aocf.exceptions.PortNotConnectedException: port not connected: ISin
        at aocf.component.Component.getConnectedComponent(Component.java:211)
        at aocf.tests.MainThreadComponents.Sum.ComputeSum(Sum.java:22)
        at…
```

- Si le composant 'Sum' demande un service ex. 'ComputeExpenential' au composant 'Sin' qui ne dispose pas se service, l'exception suivante sera généré:

```
aocf.exceptions.ConnectionException: Connection refused: method
computeExpenential not available in aocf.tests.MainThreadComponents.Sin
        at aocf.component.Component.Connect(Component.java:159)
        ataocf.tests.MainThreadComponents.MathEqComposite.<init>(MathEqCompos
ite.java:27)
```

- Si le concepteur oublié de mettre les annotations (@Requires) sur les services requises d'un composant, l'exception suivante sera généré:

```
aocf.exceptions.ConnectionException: Connection refused: requires services
of aocf.tests.MainThreadComponents.Sum must be indicated by annotation
        at aocf.component.Component.Connect(Component.java:159)
        ataocf.tests.MainThreadComponents.MathEqComposite.<init>(MathEqCompos
ite.java:28)
```

- Si un déconnection sur un composant qui était déjà déconnecté, l'exception suivante sera générée:

```
aocf.exceptions.DisconnectException: Component is already disconnected:
aocf.tests.MainThreadComponents.Sin not connected
        at aocf.component.Component.Disconnect(Component.java:189)
```

Chapitre VI: Conclusion et Perspectives

IV.1. Conclusion générale :

Notre travail s'inscrit dans le domaine de l'Architecture logicielle, plus exactement dans le sous domaine de la programmation orientée composant (POC). Notre objectif est la réalisation d'un Framework pour la programmation orienté composant et aspect (AOCF: Aspect Oriented Component Framework).

Afin d'aboutir à notre objectif, nous avons à travers tout d'abord réalisé une première activité de recherche dans le but de maitriser les notions fondamentales de l'architecture logicielle, telles que les ADLs (Architecture Description Language), les composants, les connecteurs, les configurations et les styles architecturaux. Par la suite nous avons réalisé une recherche en vue d'établir un état de l'art sur le sous domaine de la programmation orientée composant et aspect. qui est le domaine dans lequel s'inscrit de manière plus précise notre travail. Nous avons ainsi pu étudié les approches à composants les plus connu au niveau implémentation et nous avons réalisé une évaluation selon des critères qui nous semble être intéressant s'ils sont présent dans un approche à composant et qui ouvrent la voie vers l'injection des aspects. Après une synthèse sur cette étude, nous avons pu déterminer les limites des approches actuelles.

Afin de surmonter les limites des approches actuelles, nous avons proposé une nouvelle approche basée sur le concept de framework. Le framework AOCF (Aspect Oriented Component Framework) proposé unifie les approches à composant et l'orienté aspect. AOCF supporte directement les concepts fondamentaux de l'architecture logicielle tels que les concepts de composants, ports, connexion et composant composite. Un tel framework permettrait la programmation orienté composant et aspect et pourrait représenter une cible efficace dans un processus de transformation d'une spécification en ADL en une spécification de niveau implémentation.

Notre approche offre une bonne souplesse d'utilisation. Elle pourrait être utilisable facilement même par des personnes non expert en java et qui ne sont pas familier avec ce langage.

VI.2. Perspectives:

Le travail de recherche que nous venons de réaliser ouvre un nombre important de perspectives, notamment dans le domaine de la transformation d'une spécification ADL en une spécification orienté composant de niveau implémentation. Dores et déjà un nombre important de perspectives peuvent être listés. Nous reportons dans ce qui suit quelques unes

- L'extension de notre approche pour supporter d'autres cas de déploiement tels que le déploiement d'un composant en composant EJB ou en une applet....
- Le support d'autre types de connecteurs pour permettre d'offrir au concepteur divers choix concernant la technique de liaison entre composants et celle qui répond le mieux aux besoins et contraintes exprimés.
- Il est fort probable que les composants utilisés dans une conception proviennent de sources différentes. Dans ce cas, il n'est pas évident que les ports interconnectés soient totalement compatibles du moins au niveau des noms. Une couche d'adaptation d'interfaces (ports liées aux composants connectés) est ainsi nécessaire. Cette couche pourrait être localisé au niveau du connecteur liant les interfaces à accorder.
- Transformer AOCF en une approche qui supporte l'AOP distribuée et dans laquelle nous retrouvons de nouveaux concepts tels que les coupes distribués.
- L'introduction d'un cadre formel pour la détection et la résolution des interférences d'aspects lorsque plusieurs aspects sont tissés à un système à composants.

A la fin nous espérons que notre travail sera concrétisé par une communication acceptée pour une présentation à une conférence.

Annexes

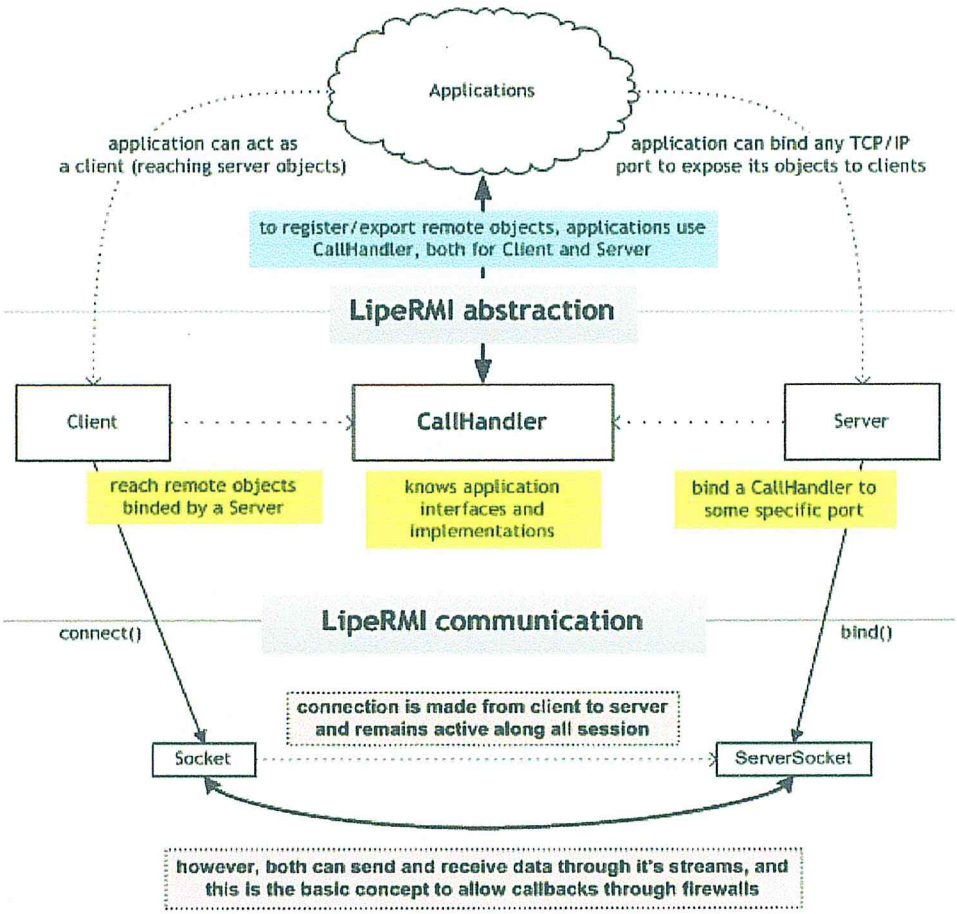
Annexe A

Lipe-RMI: "a light weight internet approach"

1. Présentation:

Lipe-RMI [2] c'est une nouvelle implémentation de java RMI (la première version en 2006 et la version disponible actuellement c'est 0.4.2), elle est totalement indépendamment de RMI permet les appels par internet, cette API a été conçu pour remédier aux problèmes vue dans RMI ordinaire, le programmeur n'a pas besoin d'un compilateur comme *rmic* le code stub est dynamiquement produit par Lipe-RMI, elle permet aussi de minimiser la bande passante.

Puisque Lipe-RMI a été conçu pour travailler dans un environnement internet, les services *Namming et Registry* n'ont pas beaucoup de sens et jusqu'à présent n'ont pas été mis en œuvre par Lipe-RMI, la seule façon d'atteindre un serveur est de connaitre leur adresse IP ou bien le nom de l'hôte.



2. Exemple:

- On prend l'interface suivant:

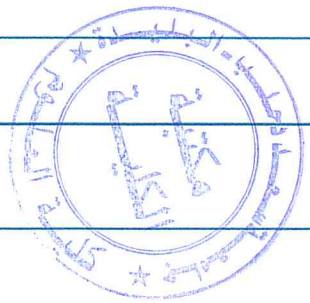
```
public interface ExampleService {
   void someMethod();
}
```

- Et une implémentation de ce service dans l'application du serveur:

```
public class ExampleServiceImpl {
    public void someMethod() {
        System.out.println("someMethod() called");
    }
}
```

- Maintenant le code de serveur en RMI native:

```
ExampleServiceImpl myService = new ExampleServiceImpl();
Naming.rebind("//localhost:4455/ExampleService", myService);
```



- En Lipe-RMI deviant:

```
ExampleServiceImpl myService = new ExampleServiceImpl();

// it always need a CallHandler!
CallHandler callHandler = new CallHandler();
callHandler.registerGlobal(ExampleService.class, myService);

server.bind(4455, callHandler);
```

La méthode 'registerGlobal' travail de la même manière pour le service binding/naming en Java RMI: la différence est ici: sans Registry.

Le code coté client en Java RMI:

```
ExampleService myServiceRemote;
myServiceRemote =
  (ExampleService) Naming.lookup("//localhost:4455/ExampleService");

myServiceRemote.someMethod();
```

- En Lipe-RMI devient:

```
// it always need a CallHandler!
CallHandler callHandler = new CallHandler();

// first we need to connect..
Client client = new Client("localhost", 4455, callHandler);

// then we can get the remote reference
ExampleService myServiceRemote;
  myServiceRemote = (ExampleService) client.getGlobal(TestService.class);

myServiceRemote.someMethod();
```

Annexe B

Javassist – Assisting Java Programming –

3. Présentation:

Javassist [3] est un système de réflexion au moment du chargement de Java. Il s'agit d'une bibliothèque de classes pour la manipulation du bytecode Java, il permet aux programmes Java de définir une nouvelle classe à l'exécution (at runtime) et de modifier un fichier de classe avant le JVM le charge (before load time). Contrairement à d'autres systèmes similaires, Javassist fournit abstraction au niveau source, les programmeurs peuvent modifier un fichier classe sans une connaissance approfondie du bytecode Java. Ils ne doivent pas même écrire une séquence de bytecode inséré; Javassist peut compiler un fragment de texte source en ligne (par exemple, une seule déclaration). Cette facilité d'utilisation est une caractéristique unique de Javassist contre d'autres outils.

4. Les cas d'utilisation de Javassist:

Javassist peut être utilisé dans les cas suivants:

- *La programmation orienté aspect*: javassist est un bon outil pour ajouter des nouvelle méthodes dans une classe java et injecter des greffons before, after et around.
- *La réflexion:* Javassist permet aux programmes Java d'utiliser un méta-objet qui contrôle les appels de méthodes sur les objets de niveau de base. Aucune compilateur spécialisé ou une machine virtuelle sont nécessaires.
- Remote Method Invocation.

5. Exemple de création d'une nouvelle classe java et méthode par javassist:

Nous prenons l'exemple de création d'une classe 'Task' étends de la classe "Compute" et implémente l'interface java 'java.lang.Runnable":

```
ClassPool cp=ClassPool.getDefault();
CtClass newClass=cp.makeClass("Task");
CtClass superClass;
superClass = ClassPool.getDefault().get("Compute");
CtClass myinterface = cp.get("java.lang.Runnable");
newClass.setInterfaces(new CtClass[] {myinterface});
newClass.setSuperClass(superClass);
newClass.toClass();
```

Bibliographies

Bibliographies

- **[ACCORD, 2002]** projet ACCORD "*Assemblage de composants par contrats en environnement ouvert et réparti*". 2002
- **[Adel, 2011]** Adel Alti: " *coexistence de la modélisation à base d'objets et de la modélisation à base de composants architecturaux pour la description de l'architecture logicielle*": thèse doctorat à l'UFAS (Algérie) 2011.
- **[Aldrich, 2005]** Aldrich, J. (2005). Open modules: Modular reasoning about advice. In *ECOOP 2005 - Object-Oriented Programming, 19th European Conference, Glasgow, UK, July 25-29, 2005, Proceedings*, volume 3586, pages 144–168. Springer.
- **[Allen, 1997a] Robert Allen:** A formal basis for architectural connection.: School of Computer Science Carnegie Mellon University Pittsburgh, September 1997.
- **[Bennouar, 2009]** Dr.Bennouar Djamel: 'une approche intégré pour l'architecture logicielle': thèse doctorat à l'ESI Avril 2009.
- **[Behrmann 2004]** Gerd Behrmann, Alexandre David and Kim G. Larsen. : A Tutorial on Uppaal. In SFM-RT: International School on Formal Methods for the Design of Computer, Communication, and Software Systems, numéro 3185 de LNCS, pages 200–236. Springer-Verlag, 2004. (Cited on pages 114 and 177.)
- **[ETMVJ, 2004]** E. Bruneton, T.Coupaye, M. Leclercq, V. Quéma, and J-B. Stefani, « An open component model and its support in java, In Ivica Crnkovic, Judith A. Stafford, Heinz W. Schmidt, and Kurt C. Wallnau, editors,CBSE,volume 3054 of Lecture Notes in Computer Science,pages7–22.Springer,2004. ISBN:3-540-21998-6.
- **[Garlan,1999] David Garlan and Mary Shaw:** an introduction to software architecture. January 1999.
- **[Hannousse, 2012]** Abdelhakim Hannousse: "Aspectualizing Component Models: Implementation and Interferences Analysis": thèse doctorat à l'*Université Nantes Angers Le Mans Janvier 2012.*
- **[Jboss, 2002]** JBoss team. JBoss Aspect Oriented Programming. *http ://www.jboss.org/*, 2002. developers/projects/jboss/aop.

- **[JCD, 2002a]** Jonathan Aldrich, Craig Chambers, et David Notkin, "Architectural reasoning in Archjava", In ECOOP '02: Proceedings of the 16th European Conference on Object-Oriented Programming, pages 334–367, London, UK, 2002. Springer-Verlag.
- **[JCD, 2002b]** Jonathan Aldrich, Craig Chambers, et David Notkin, "ArchJava: Connecting Software Architecture to Implementation", In ICSE, pages 187–197. ACM, 2002.
- **[Julien, 2012]** Julien Bigot "Du support générique d'opérateurs de composition dans les modèles de composants logiciels, application au calcul à haute performance" thèse doctorat à l'université Européenne de Bretagne. Octobre 2012.
- **[LEGOND, 2005]** LEGOND-AUBRY Fabrice, Un modèle d'assemblage de composants par Contrat et Programmation Orientée Aspect, Thèse De Doctorat Du Conservatoire National Des Arts Et Métiers, France, Juillet 2005, pp. 21-29.
- **[Larsen, 1997]** Kim G. Larsen, Paul Pettersson and Wang Yi. Uppaal in a nutshell. International Journal on Software Tools for Technology Transfer, vol. 1, pages 134–152, 1997. (Cited on pages 114, 119 and 177.).
- **[Medvidovic, 2000]**: Nenad Medvidovic and Nikunj Mehta. "Java Beans and Software Architecture. In Hossein Bidgoli, ed., *The Internet Encyclopedia*, John Wiley & Sons, Inc., vol. 2, pages 388-400, December 2003.
- **[M. Pinto, 2003]** M. Pinto, L. Fuentes, J. T. (2003). Daop-adl : An architecture description language for dynamic component and aspect-based development. Generative Programming and Component Engineering (GPCE).
- **[M. Pinto, 2005]** M. Pinto, L. Fuentes, J. T. (2005). A component and aspect dynamic platform. The Computer Journal.
- **[Nicolas, 2007]** Nicolas Pessemier, Unification des approches par aspects et à composants, Thèse de doctorat de l'universit´e des Sciences et Technologies de Lille, Juin 2007.
- **[OMG, 2002]** OMG (june 2002). CORBA Components, v3.0 (full specification), Document formal/02-06-65.
- **[Ongkingco et al., 2006]** Ongkingco, N., Avgustinov, P., Tibble, J., Hendren, L., de Moor, O., and Sittampalam,G. (2006). Adding OpenModules to

Aspectj. In *Proceedings of the 5nd International Conference on Aspect-Oriented Software Development (AOSD'03)*. ACM Press.

- **[PDFS02]** R. Pawlak, L. Duchien, G. Florin, and L. Seinturier. Jac : un framework pour la programmation orientée aspect en java. *L'objet*, 8(4) :145–168, 2002.
- **[Per et al, 2000]** Pernici B., Mecella M., Batini C., -Conceptual Modeling and Software Components Reuse: Towards the Unification-, In Solvberg A., Brinkkemper S., Lindencrona E. (eds.): Information Systems Engineering: State of the Art and Research Themes. Springer Verlag, 2000
- **[Reiko, 2004]** Reiko Heckel, Alexey Cherchago, and Marc Lohmann, "A formal approach to service specification and matching based on graph transformation", Electronic Notes in Theoretical Computer Science, 105 :37–49, 2004.
- **[RJ, 2006]** R. Johnson, J. Hoeller, e. a. (2006). Spring - java/j2ee application framework. http://static.springframework.org/spring/docs/2.0.x/spring-reference.pdf.
- **[Suvée et al., 2006]** Suvée, D., Fraine, B. D., and Vanderperren,W. (2006). A symmetric and unified approach towards combining aspect-oriented and component-based software development. In Component-Based Software Engineering, 9th International Symposium, CBSE 2006, Västerås, Sweden, June 29 - July 1, 2006, Proceedings, pages 114–122.
- **[Suvée, 2003]** Suvée, D., Wim V. and Viviane J., JAsCo: an Aspect-Oriented approach tailored for CBSD, Proc. of the second international conf. on aspect-oriented software development, Boston, march 2003.
- **[Sylvain, 2010]** Sylvain Chardigny, « Extraction d'une architecture logicielle à base de composants depuis un système orienté objet Une approche par exploration », Thèse de doctorat, université de Nantes, 2010
- **[SBF, 1996]** Steve Sparks, Kevin Benner, et Chris Faris. Managing object-oriented framework reuse. Computer, 29(9) :52.61, Septembre 1996.
- **[TALIGENT, 1995]** TALIGENT. Leveraging object-oriented frameworks, 1995. White paper.
- **[Villalobos, 2003]**Villalobos J., -Fédération de composants: une architecture logicielle pour la composition par coordination-. Thèse en Informatique à l'Université Joseph Fourier (Grenoble), juillet 2003

- **[Wilson, 1990]** D.A. Wilson, -Programming with MacApp-, Reading, Massachusetts, Addison-Wesley.

Webographies

- [1] http://caosd.lcc.uma.es/cam-daop/DAOP-ADL.htm
- [2] http://lipermi.sourceforge.net/
- [3] http://www.csg.ci.i.u-tokyo.ac.jp/~chiba/javassist/
- [4] http://www.eclipse.org/
- [5] http://commons.apache.org/proper/commons-bcel/
- [6] http://asm.ow2.org/