Université Abou Bekr Belkaid جامعة أبي بكر بلقايد
تلمسان الجزائر

République Algérienne Démocratique et Populaire
Université Abou Bakr Belkaid– Tlemcen
Faculté des Sciences
Département d'Informatique

Mémoire de fin d'études

pour l'obtention du diplôme de Master en Informatique

Option: *Système d'Information et de Connaissances (S.I.C)*

Thème

Une implémentation de l'approche Aspect pour l'adaptation dynamique des applications à base de composants : Etude de faisabilité

Réalisé par :

- *M^lle Mohammedi Asmaa*

Présenté le 25 Septembre 2012 devant le jury composé de :

- *M^r A. Benamar* *(Président)*
- *M^r A. Belabed* *(Encadreur)*
- *M^r A. Bentaalah* *(Examinateur)*
- *M^r H. Matallah* *(Examinateur)*

Année universitaire : 2011-2012

SOMMAIRE

Table des figures

Listings

Liste des tableaux

INTRODUCTION GENERALE

INTRODUCTION GENERALE

Le génie logiciel à base de composants a pour objectif le développement de gros logiciels par l'intégration de composants existants.

La programmation par aspect combinée avec la programmation à base de composants pour faire l'adaptation dynamique représente un nouveau challenge pour le génie logiciel et l'informatique récente.

Le besoin d'adaptation des applications selon les changements de l'environnement et les besoins des utilisateurs rend indispensable de trouver un moyen pour supporter les mis à jour, surtouts pour les applications à haute disponibilité dont l'adaptation doit intervenir dynamiquement sans avoir besoin d'arrêter puis redémarrer tout le projet en exécution.

Le travail présenté dans ce mémoire concerne l'adaptation dynamique dans le cadre des applications à base de composants.

L'approche utilisée dans notre travail est une approche par aspect. Notre mémoire est organisé comme suit : Le chapitre I introduit la notion des composants et la programmation à base de composants. Le chapitre II présente la programmation par aspect et ses plateformes. Le chapitre III survole la notion d'adaptation dynamique et ses approches et mécanismes. Le chapitre IV concerne la partie pratique de notre travail qui présente une étude de faisabilité sur l'adaptation à base de tissage dynamique d'aspects ainsi qu'un exemple d'implémentation. Et enfin une conclusion générale qui résume la totalité de notre travail et introduit quelques perspectives..

Chapitre 1
Programmation à base de composants

I.1 Introduction

La programmation orientée composant (POC) est pratique car elle permet la décomposition d'une application en un ensemble d'entités collaborant entre eux pour former un système qui entend des requêtes envoyées par des utilisateurs avec la possibilité de développer chaque entité séparément en favorisant aussi la réutilisation des composants grâce à l'indépendance des briques logicielles.

Nous allons, pour commencer, présenter ce qu'est un composant logiciel. Puis nous allons exposer quelques modèles de composants.

I.2 Architecture logicielle et composants

L'architecture logicielle est une discipline récente du génie logiciel focalisant sur la structure, le comportement et les propriétés globales d'un système et s'adresse plus particulièrement à la conception des logiciels complexes et de grande taille.

L'objectif de la description des architectures et d'avoir l'abstraction nécessaire pour Modéliser les systèmes logiciels complexes durant leur développement, déploiement et évolution.

L'architecture logicielle, comme le montre la figure I.1, se définit comme une spécification abstraite d'un système en termes de composants logiciels ou modules qui le constituent, des interactions entre ces composants (connecteurs) et d'un ensemble de règles qui gouvernent cette interaction [Boa95, CLB+01, Gar00, IBRZ00, LVM95]. Les composants encapsulent typiquement l'information ou la fonctionnalité. Tandis que les connecteurs assurent la communication entre les composants.

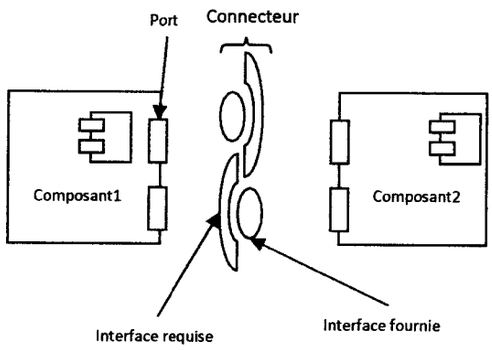

Figure I.1 – Les éléments d'une architecture logicielle

Le contenu d'un programme restera toujours le même. Néanmoins, sa phase de conception évolue.

I.2.1 Définition de composant

Lorsque l'on parle de composants, il s'agit de simples fichiers, contenant généralement du code compilé. Sous les systèmes de type Unix, par exemple, les composants se présentent sous la forme de fichiers portant l'extension .so (shared object). Sous les

systèmes de type Microsoft Windows, il s'agit des fameuses dll (dynamic library link). On parle également de modules, de bibliothèques, ou de librairies. Il est possible de créer des composants avec la grande majorité des langages. Toutefois, dans certains cas, notamment pour les langages interprétés ou semi-compilés il n'est pas possible de créer des composants « classiques ». Par exemple, en java, il est possible de créer des bibliothèques de classes (.jar). Ainsi, seul un programme écrit en java pourra utiliser comme composant un fichier .jar.

Un composant regroupe un certain nombre de fonctionnalités qui peuvent être appelées depuis un programme externe, ou client. Comme le composant ne contient que du code compilé, il n'est a priori pas possible de savoir de quelle manière elles sont implémentées, à moins de disposer du code source. De plus, pour pouvoir être utilisé, le composant doit fournir une interface, c'est-à-dire un ensemble de fonctions lui permettant de communiquer avec le programme client. Dans le cas où le code source n'est pas disponible, les spécifications détaillées de ces fonctions doivent être fournies avec la documentation.

Un composant est censé fournir un service bien précis. Les fonctionnalités qu'il encapsule doivent être en rapport et cohérentes entre elles.

Enfin, un composant doit être réutilisable, c'est-à-dire qu'il ne doit pas simplement servir dans le cadre du projet durant lequel il a été développé. Cet aspect n'est possible qu'à la condition qu'il possède un comportement suffisamment général.

En voici quelques définitions selon plusieurs auteurs:

- « A software component is a static abstraction with plugs » Nierstraz et Tsichritzis(95).
- « A component is a piece of software small enough to create and maintain, big enough to deploy and support, and with standard interfaces for interoperability » Harris (95).
- « Software component are defined as prefabricated, pretested, self-contained, reusable software modules that perform specific functions » MetaGroup (94).
- « A software component is a unit of composition with contractually specified interfaces and explicit context dependencies. Software component can be

deployed independently and is subject to composition by 3rd parties » Participant 1er Workshop Component Oriented Programming (97).

La définition d'un composant se rapproche beaucoup de celle de l'objet, en effet, ce qui les distingue, c'est d'une part la granularité d'un composant, mais aussi ses contraintes et ses fonctionnalités. On peut représenter un composant comme un type de boîte noire (la connaissance de son implémentation n'est pas nécessaire) constitué d'un ou plusieurs objets, où il suffit de connaître les services rendus et les quelques règles d'interconnexion. En effet, comme nous l'illustre la figure ci-dessous, un composant exporte les interfaces qu'il fournit et les interfaces qu'il requiert

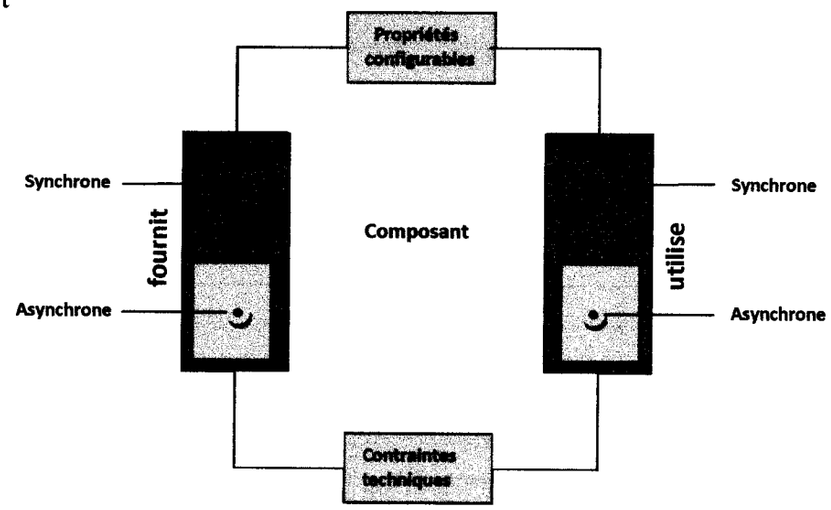

Figure I.2 – Architecture d'un composant

Ces interfaces sont aussi appelées ports. Un exemple de port est celui à la base de l'introspection. Celui-ci permet au composant de s'analyser lui-même (méthodes/propriétés via un mécanisme de réflexion) dans le but d'afficher une page des propriétés au sein de l'IDE sous lequel on développe.

De plus, le composant est interconnectable avec d'autres composants d'origines diverses, configurable, auto descriptif (introspection), mais aussi, il est diffusable de manière unitaire et prêt à l'emploi. La plupart de ces fonctionnalités sont dues à ses interfaces.

En ce qui concerne la communication vis-à-vis des composants (application/composant ou composant/composant), on entend parler (comme l'illustre le schéma précédent) de communications synchrones, asynchrones voire de diffusions en continu (flots de données). La communication synchrone est en fait un simple appel de méthode comme il est possible de faire sur un objet, et le mode asynchrone reflète un mécanisme d'événements (abonnement/notification comme en Java).

En effet, il n'y a pas de consensus qui ait donné une définition commune aux composants. Selon les modèles (Corba, Java, Microsoft), les composants sont différents, ce qui rend difficile la tâche de les décrire.

I.2.2 Types de composants

Un type de composant est caractérisé par trois éléments : ses interfaces, ses modes de coopération et ses propriétés configurables [1] :

• **Mode de coopération :** Pour chacune des interfaces d'un composant, il est nécessaire de spécifier le mode de coopération. Il existe trois modes de coopération : le mode synchrone (par exemple l'invocation de méthode), le mode asynchrone (par exemple l'envoi d'un message) et le mode diffusion en continu (par exemple la communication par flots de données).

• **Propriétés configurables :** Ces propriétés permettent d'adapter une instance de composant en configurant son comportement. Le code du composant n'est pas modifié en fonction du besoin, mais paramétré, augmentant ainsi la réutilisabilité du composant.

A l'image des composants Java, il existe deux catégories de composants : les composants client/IHM (généraux) (Java Beans par exemple) et les composants serveur/métier (EJBs), dont le mode de fonctionnement diffère légèrement et qui répond à d'autres besoins.

a. **Composants client/IHM :** On appelle un composant « composant IHM » car il est surtout dédié, à l'image des beans aux interfaces.
b. **Composants serveur/métier :** Il existe un autre type de composant, appelé composant métier, qui est beaucoup plus spécifique. Il est aussi appelé composant serveur car il ne fonctionne que sur ce type de poste. En général on entend par composant ce composant-là, catégorie dans laquelle rentrent

les EJBs de Java par exemple (et non les Java Beans qui peuvent s'exécuter chez le client). La présence d'un conteneur pour encapsuler le composant et d'un serveur d'applications les distingue des autres. C'est d'ailleurs eux qui répondent à d'autres besoins que les composants standards, pas les composants métier en eux-mêmes. Ces besoins expriment la volonté de ne pas se soucier des services non fonctionnels du composant, juste de sa partie métier.

Par exemple, quand on programme une application, il est nécessaire d'une part de se soucier de ce que fait l'application en elle-même, mais aussi d'autres caractéristiques plus proches du système et récurrentes à d'autres programmes, et qui ne sont pas incluses sous forme de services de manière standardisée. Voici quelques services principaux masqués par les conteneurs : Nommage du composant ; Gestion de la sécurité ; Sûreté de fonctionnement (gestion de concurrence ou pannes par exemple) ; Persistance pour sauvegarder l'état d'un composant via une sérialisation ; Prise en charge partielle des connecteurs inter composants.

En ce qui concerne le serveur d'applications, c'est sur lui que repose le conteneur. On parle aussi de « structures d'accueil ». Il joue le rôle d'interface entre le système et le conteneur et constitue un espace d'exécution pour le conteneur et ses composants. Voici un schéma illustrant ceci :

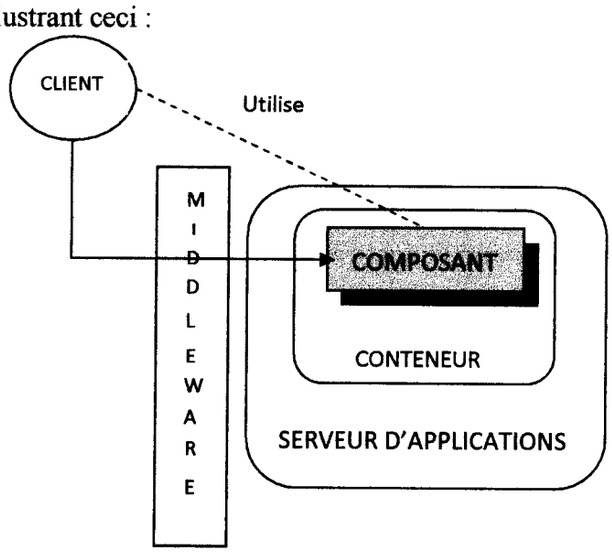

Figure I.3 – Architecture Client - Serveur d'un composant

Un exemple de serveur d'applications est Web Sphere d'IBM, qui suit la spécification J2EE. Le Middleware qui est décrit ici est un terme générique pour parler de l'interface entre l'infrastructure réseau et la structure d'accueil. Cela peut être PHP, Java... etc.

I.2.3 Port

Un port permet de spécifier les points d'interactions d'un composant avec son environnement. Il permet de regrouper les interfaces relatives à une préoccupation donnée (par exemple, les interfaces d'administration, configuration, etc.). Les ports (et par conséquent les interfaces) peuvent être fournis ou requis, ils peuvent même posséder à la fois des interfaces requises et fournies. Le comportement interne d'un composant n'est visible et accessible qu'à travers ses ports.

I.2.4 Interface

C'est le point de communication qui permet d'interagir avec l'environnement [Oli05].

Une interface de composant est une spécification de ses points d'accès qui permettent aux clients d'y accéder aux services fournis par les composants.

Une interface ne fournit aucune implémentation, seulement une description des protocoles d'utilisation des services et leurs noms.

Pour un composant, il existe deux types d'interfaces :

– Les interfaces fournies décrivent les services proposés par le composant : interface métier, interface de configuration, de maintenance, ...

– Les interfaces requises décrivent les services que les autres composants doivent fournir pour le bon fonctionnement du composant.

Ces interfaces sont exprimées par l'intermédiaire des ports.

I.2.5 Connecteur

Le connecteur, appelé aussi connexion, correspond à un élément d'architecture logicielle qui représente un moyen d'interaction (transfert de contrôle et de données)

entre les composants [Car03, Oli05]. Il permet également d'assembler des composants en utilisant leurs interfaces fournies et requises.

Les connecteurs sont classifiés [2] selon les services qu'ils offrent, donc on trouve les services de communication, de coordination, de conversion et de facilitation. Par ailleurs d'autres services peuvent être offerts par les connecteurs, par exemple, la notion de MVC (multi-versioning connector) [3], qui permet l'utilisation simultanée de plusieurs versions d'un même composant, ce connecteur redirige les appels aux services vers la version du composant la plus adéquate. Les connecteurs peuvent aussi être utilisés comme moyen de détection et de réparation (self-healing mechanism) des anomalies au niveau des objets et des interactions des tâches dans les composants [4].

Afin de fixer les conditions d'utilisation des connecteurs, des contraintes sont également déclarées.

I.2.6 Contraintes

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.

I.2.7 Configuration

Une configuration décrit l'ensemble des composants logiciels nécessaires pour le fonctionnement d'une application, ainsi que leurs connexions. Elle décrit la structure architecturale caractéristique du style c'est à dire la manière dont doivent interagir les composants et les connecteurs. Elle est définie par un schéma décrivant le comportement général d'une spécification, la composition des composants et des connecteurs [RL00].

Une configuration représente en fait une instance possible d'un style architectural. Ce dernier nous le détaillerons dans la suite de chapitre.

Plus précisément, une configuration décrit les instances de composants intervenants et les connexions qu'elles entretiennent entre elles [MDK94].

La conception d'une architecture logicielle, a une importance capitale pour la réussite d'un projet informatique. Elle est souvent liée au savoir-faire de l'architecte. Une architecture logicielle doit tenir compte des contraintes suivantes :

– **La réutilisabilité :** est la capacité à rendre générique des composants et à concevoir et construire des boites noires susceptibles de fonctionner avec des langages et des environnements variés.

– **La maintenabilité :** est la capacité de modifier et d'adapter une application afin de la maintenir sur une période de vie assez longue. Une architecture bien spécifiée doit être maintenue tout au long de son cycle de vie. La prévision de l'intégration des extensions à l'architecture et la correction des erreurs sont nécessaires dès la phase de conception.

– **La performance :** c'est l'optimisation du temps mis par une application pour répondre à une requête donnée. La performance d'une application dépend de l'architecture logicielle choisie, de son environnement d'exécution, de son implémentation et de la puissance des infra-structures utilisées (e.g. débit du réseau utilisé).

I.3 Programmation a base des composants

La programmation par composant est donc basée sur le fait d'intégrer, d'emboîter des composants entre eux pour former notre programme. A titre de comparaison, on pourrait prendre l'exemple des sociétés d'assembleur de Micro Informatique. Lors de la conception d'un PC, ils emboîtent des composants dont ils connaissent le principe de fonctionnement et la manière dont chacun d'eux peuvent communiquer avec son environnement.

Cette approche se caractérise par une séparation entre l'étape de développement et celle d'assemblage des composants. L'assemblage est réalisé à partir de composants qui peuvent être obtenus de divers constructeurs. L'assemblage doit satisfaire un ensemble de contraintes liées au bon fonctionnement de chaque composant. Ces contraintes sont appelées contrat.

I.3.1 langages de description d'architecture

Pour mettre en évidence la structure d'une application, une architecture peut être décrite à l'aide de langages de description d'architecture ADL (Architecture Description Language). Les ADLs focalisent sur la structure de l'application globale à un haut niveau d'abstraction plutôt que sur les détails d'implémentation [5]. Pour cela ils fournissent une syntaxe concrète et un cadre conceptuel pour la modélisation d'une architecture conceptuelle du système logiciel. Celle-ci repose généralement sur les modèles de base suivants :

- Les composants représentant des unités de calcul ou de stockage
- les connecteurs qui sont des composants particuliers employés pour modéliser les interactions entre les composants ainsi que les règles qui régissent ces interactions ;
- les configurations architecturales La configuration permet de décrire la structure globale d'un logiciel, c'est-à-dire l'assemblage des composants de ce logiciel sous forme de graphes de connexion formés de composants et de connecteurs.

Les ADLs plus connus sont : Wright [6], ACME [7], Rapide [8], UniCon [9], C2 [10] et Darwin [11].

I.3.2 modèle de composants et environnement d'exécution

Un modèle à composants définit la structure des composants et de leurs assemblages et permet de réaliser le développement suivant le cycle de développement propre à cette approche [20], c'est une spécification d'une infrastructure de support aux composants, telles que EJB, CCM, Fractal, .Net.

Un modèle à composants est accompagné par un environnement d'exécution qui fournit du support aux applications pendant l'exécution. L'environnement d'exécution est parfois comparé à un mini système d'exploitation car il est chargé de gérer des aspects divers tels que le cycle de vie des instances des composants ou bien les propriétés non fonctionnelles [16].

I.4 Etude De Divers Modèles A Composants

Plusieurs modèles de composants on été définies, des standards industriels tel que EJB de Sun MicroSystems [12], CCM de l'OMG [13] et .NET de Microsoft [14] d'autre en voix de standardisation tel que Fractal [15]. Nous consacrons cette partie, pour présenter une brève description des architectures de ces modèles. En général on distingue deux différents modèles : Une connexion entre composants basant sur une interface réceptacle appelés généralement « connecteur » comme dans le modèle CCM (Corba Component Model) de Corba. Dans la deuxième, les attributs du composant sont « créés » grâce à une interface donnant une spécification des noms des méthodes assesseurs et d'affectation, qui sont à la base de sa configuration, comme pour le Java bean.

I.4.1 le modèle Entreprise Java Beans (EJB)

I.4.1.1 présentation

Le modèle de composants Entreprise Java Beans (EJB), dont la première spécification est apparue en 1998 par Sun Microsystems, propose un modèle de composants pour la construction d'applications réparties côté serveur dans le cadre des architectures n-tiers. Une telle application est constituée de composants appelés Beans implémentés en Java.

Une entreprise bean est un composant côté serveur qui encapsule la logique métier d'une application. La logique métier est le code qui accomplit le but de l'application [17].

I.4.1.2 Architecture

Les EJB sont déployés dans des environnements trois tiers ou plus, donc on trouve : un client, un serveur (EJB), et des sources de données.

Les EJB s'exécutent dans un environnement particulier : le serveur d'EJB. Celui ci fournit un ensemble de fonctionnalités utilisées par un ou plusieurs conteneurs d'EJB qui constituent le serveur d'EJB. En réalité, c'est dans un conteneur que s'exécute un EJB et il lui est impossible de s'exécuter en dehors (voire figure I.4).

• L'Entreprise Bean (EB) : Un entreprise Java Bean est un composant écrit en Java. Il s'exécute au sein d'un conteneur qui lui fournit divers services systèmes : sécurité, gestion des transactions etc.

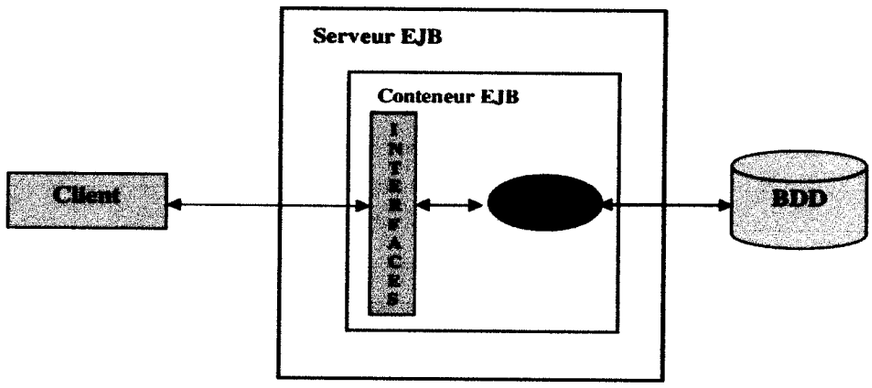

Figure I.4 – Architecture EJB [22]

• Serveur EJB : Le serveur EJB fournit aux entreprises beans des services système et gère les conteneurs dans lesquels s'exécutent les beans. Il doit offrir un service de nommage et un service de transaction.

• Le conteneur d'EJB : Un conteneur est un système d'exécution pour un ou plusieurs entreprises beans. Exemples : Weblogic, Websphere, BES, Oracle Orion Server, JRun, JBoss…

Les fonctionnalités fournies par un conteneur d'EJB sont :

- ✓ La connectivité entre les clients et les EJB ;
- ✓ La gestion de la persistance ;
- ✓ La gestion des transactions ;
- ✓ La gestion de la sécurité ;
- ✓ La gestion de la concurrence ;
- ✓ La gestion du cycle de vie des composants : c'est le conteneur EJB qui gère la vie des instances du composant.
- ✓ La gestion du déploiement : on appelle déploiement, la phase d'installation du composant dans le serveur d'application. Lors du déploiement, le serveur vérifie la cohérence des différents éléments du composant (interfaces, classe d'implémentation, fichiers de déploiement). Si jamais le composant ne passe pas cette vérification, il n'est pas déployé.

• Client EJB : un client peut être : Une servlet, Une applet, Une application classique ou bien un autre Bean.

• Interfaces : un client communique avec les EJB, à travers les interfaces implémentées au niveau de conteneur EJB, Ces interfaces définissent la vue qu'a le client sur le bean, on trouve principalement deux interfaces :

- ✓ L'interface Home : L'interface home définit les méthodes utilisées par le client pour créer, localiser et détruire les instances d'une entreprise bean.
- ✓ L'interface Remote : L'interface remote définit les méthodes métier implémentées dans le bean. Un client accède à ces méthodes via cette interface.

Trois types de composants peuvent être définis :

• Les EJB session : représentent la logique applicative, c'est à dire l'ensemble des fonctionnalités fournies aux clients. Ils peuvent être avec ou sans état.

- ✓ Sans état : utilisés pour traiter les requêtes de plusieurs clients.
- ✓ Avec état : conservent un état entre les appels. Un EJB de ce type est associé à un seul client et ne peut être partagé.

• Les EJB entité : représentent les objets persistants de l'entreprise, stockés dans une base de données. Chaque enregistrement de la base de données, est chargé dans un EJB entité pour qu'il puisse être manipulé.

• Les EJB orientés message : introduits dans la spécification 2.0, ils permettent de traiter les requêtes de façon asynchrone.

Les modules EJB sont livrés sous forme de fichiers contenant, en plus des composants, un ensemble de descripteurs. Ces descripteurs contiennent certaines informations relatives aux composants et décrivent la façon de les gérer à l'exécution. Le déploiement d'un composant EJB consiste à injecter le composant effectif, contenu dans l'unité de déploiement, dans l'environnement d'exécution.

I.4.2 le modèle CORBA Component Model (CCM)

I.4.2.1 présentation

Spécifié par l'Object Management Group (OMG) en 2002, le modèle CCM est basé sur CORBA qui permet l'utilisation d'environnements d'exécution hétérogènes [18]. Le

CCM se décompose en deux niveaux : un niveau basique, qui permet essentiellement de « componentiser » (rendre des composants) des objets CORBA, et un niveau étendu, beaucoup plus riche et intéressant. Tout comme les EJB de Sun, auxquels correspond la version basique du CCM en Java, la technologie CCM repose sur l'utilisation de conteneurs pour héberger les instances de composants et faciliter leur déploiement [19].

CCM propose toute une structure pour définir un composant, son comportement, son intégration, et son déploiement dans l'environnement distribué CORBA [21].

CCM définit les types de composants, ainsi que leurs implantations, à l'aide de langages déclaratifs. Les types de composants sont décrits en utilisant une extension de l'IDL de CORBA qui permet de programmer les composants dans des langages différents et de les exécuter dans des environnements hétérogènes.

Les implantations des composants sont décrites à l'aide du langage de description CIDL [20].

I.4.2.2 Architecture

La spécification CCM [18], est découpée en quatre modèles couvrant le cycle de développement et de déploiement des applications à base de composants CORBA [22] :

1. Le modèle abstrait de composant : qui spécifie en IDL (Interface Definition Language) le composant en termes de ports et d'interfaces. Le composant CORBA est constitué de [21] :

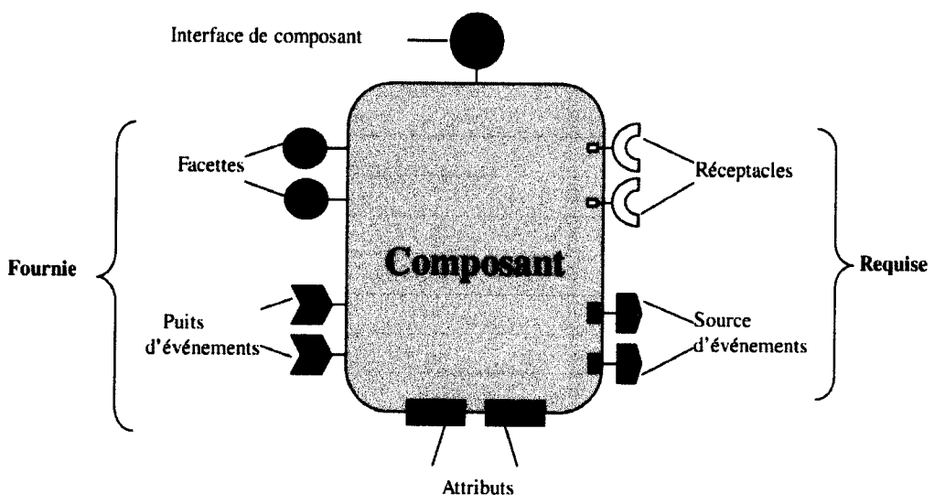

Figure I.5 – modèle de composant CCM

- d'une référence,
- de quatre types de ports:
  - Facette : c'est une interface fournie par le composant, elle donne une vue sur le composant Il peut y avoir plusieurs facettes.
  - Réceptacles : c'est une interface requise qui permet au composant de recevoir et d'exploiter des données fournies par d'autres composants via un lien dynamique.
  - La source d'événements : permet de diffuser un type d'événement vers des Puits d'événements.
  - Le Puits d'événements : permet de recevoir des événements.
- des attributs,
- des fabriques qui contrôlent le comportement du composant en fonction de son cycle de vie et qui gère les instances du composant (création, gestion de références, recherche, exploration,...)

Dans les facettes du composant, une interface particulière nommée Equivalent Interface, permet la navigation entre les différentes facettes du composant.

2. Le modèle abstrait de programmation : définit la façon d'implanter un composant à l'aide du langage CIDL (Component Implementation Description Language) et du framework CIF (Component Implementation Framework) ;

3. Le modèle de dépoilement : définit comment le composant sera distribué, assemblé et déployé dans une architecture CCM ;

4. Le modèle d'exécution : définit l'environnement d'exécution des instances de composants (voire figure I.6). Le rôle principal des conteneurs est de masquer et prendre en charge les aspects non fonctionnels des composants qu'il n'est alors plus nécessaire de programmer.

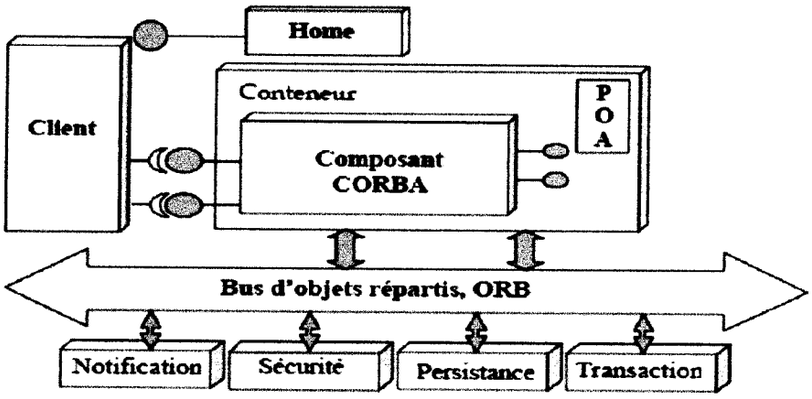

Figure I.6 – environnement d'exécution CCM

Le schéma définit des Conteneurs CCM, des composants CORBA qui s'exécutent sur ces Conteneurs, des Clients, l'adaptateur POA (Portable Object Adapter), le bus ORB (Object Request Broker) et des Services CORBA tels que : les transactions, sécurité, persistance, etc. le composant interagit avec l'adaptateur POA pour les transactions, la sécurité, la persistance et la notification des services via les interfaces du conteneur.

I.4.3 Le modèle de composants .NET

I.4.3.1 présentation

Lancée le 22 juin 2000 lors d'une conférence donnée par Bill Gates et Steve Ballmer à l'occasion du Forum 2000, ".NET" ("dot Net" en anglais) est une architecture logicielle destinée à faciliter la création, le déploiement des applications en général, mais plus spécifiquement des applications Web ou des services Web.

Cette architecture logicielle concerne aussi bien les clients que les serveurs qui vont dialoguer entre eux à l'aide d' XML [23].

Le modèle de composants .NET a pour domaine d'application les applications largement réparties réalisées à partir de différents langages de programmation.

Avec .NET, Microsoft présente une nouvelle plateforme dont le but est de développer simplement des applications Web inter opérables, reposant sur une architecture totalement nouvelle [24].

I.4.3.2 Architecture

Le modèle de composants est étroitement lié au framework .NET (figure I.7) et se base sur le principe d'une machine virtuelle appelée Common Language Runtime (CLR) et d'un langage intermédiaire le Microsoft Intermediaire Language (MSIL) comparable au bytecode Java. Chaque composant .NET est programmé dans un langage de programmation donné (C#, VB .NET, ASP, ADO .NET, etc.), puis est compilé une première fois en MSIL qui lui-même peut être interprété par la CLR. [1].

Les composants .NET sont appelés assemblage. D'un point de vue externe, un assemblage est caractérisé par un nom, un numéro de version et un ensemble de ressources et de méthodes exportées, Il peut s'agir d'un exécutable (.exe) ou d'une bibliothèque (.dll).

D'un point de vue interne, un assemblage comporte les 4 éléments suivants [22]:

- les méta-données qui décrivent l'assemblage,
- les méta-données qui décrivent chacun des types contenus dans l'assemblage,
- le code des types sous la forme de MSIL,
- les ressources : pouvant être des pages HTML, des images, des sons ou des Vidéos.

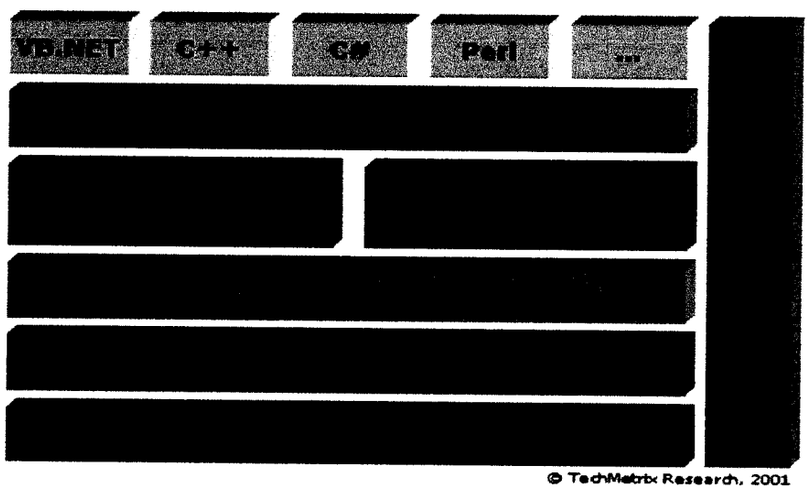

Figure I.7 – L'architecture générale de .NET

Le fichier de description, appelé Manifest, fournit les informations telles que le numéro de version de l'assemblage, les méthodes importées et exportées ainsi que les dépendances de modules.

.NET propose une évolution de COM+ « .NET remoting » qui permet la distribution de composants. COM+ est une version avancée de COM qui regroupe l'ensemble des services dédiés aux applications réparties de Windows, à savoir :

DCOM, MTS et MSMQ.

- DCOM est l'équivalent au niveau de la couche transport du protocole IIOP de CORBA mais pour la plate-forme Windows.

- MTS (Microsoft Transaction Server) permet le développement, le déploiement et l'exécution de composants répartis fonctionnant sous le contrôle d'un conteneur. MTS offre également des services pour la prise en charge des transactions.

- MSMQ (Microsoft Message Queue Server) est un middleware de type

Message Oriented Middleware qui prend en charge la communication asynchrone point à point entre applications. MSMQ permet la création et l'utilisation d'entités appelées files (queues) par lesquelles les messages sont échangés. MSMQ peut aussi être utilisé dans le cadre de transactions réparties.

.NET supporte trois modèles de composants :

1. « Single Call » : Une instance du composant ne sert qu'une seule requête. Ni l'état du composant ni le contexte des clients ne sont gérés.

2. « Singleton Objects » : Une instance du composant peut servir plusieurs clients simultanément. L'état du composant est partagé par les clients.

3. « Client Activated Objects » : Une instance du composant est activée pour chaque client (mode d'activation très proche de celui d'un composant COM+).

Le contexte du client est géré entre ses différents appels.

Le déploiement d'une application peut se limiter à la copie de fichiers dans une arborescence de répertoires. Les informations de configuration sont stockées dans des fichiers XML qui peuvent être modifiés avec un quelconque éditeur de texte.

I.5 Cycle de développement d'une application à base de composants

Le cycle de développement d'une application à base de composants passe par trois étapes principales (figure I.3) [16] [25] :

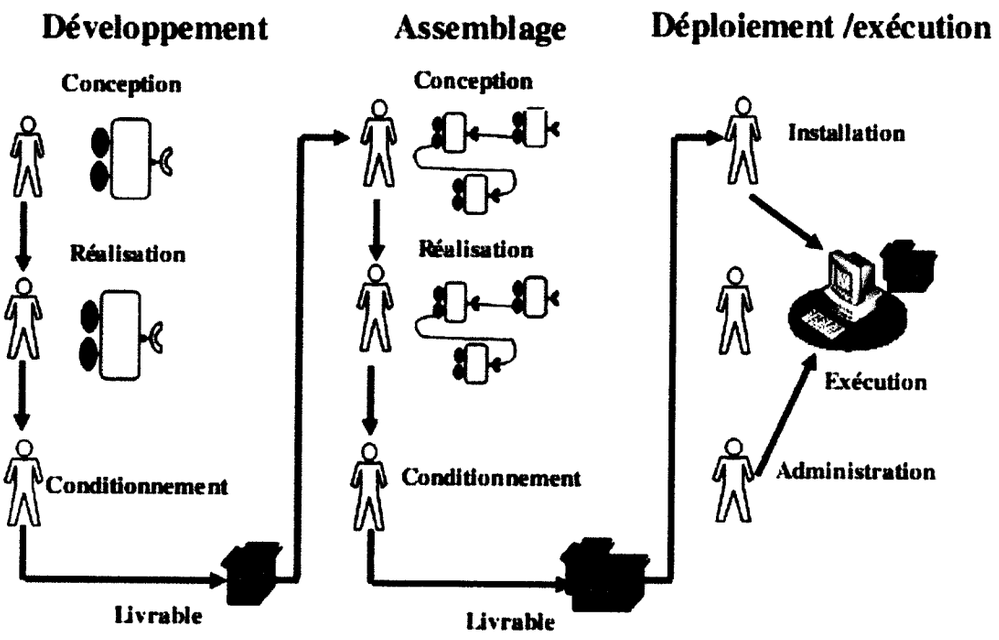

Figure I.8 – Cycle de vie de développement simplifié

1- le développement des composants : Le développement de composants inclut les activités de conception, réalisation et conditionnement des composants destinés à être assemblés dans différentes applications. La conception inclut la spécification de la vue externe (interfaces et propriété de configuration) d'un composant ainsi que de son comportement. La réalisation, c'est l'implémentation effective des interfaces du composant (les fonctionnalités), un composant peut avoir plusieurs implémentations pour une même vue externe. Le conditionnement consiste à introduire le composant dans un package livrable, ce package permet le dépoilement du composant dans une application d'une manière indépendante (autonomie).

2- L'Assemblage (composition) : L'étape d'assemblage suppose l'existence de composants développés auparavant qui sont utilisés par les acteurs de cette étape. L'assemblage (ou composition) de composants inclut les activités de conception,

réalisation et de conditionnement. La conception d'un assemblage est réalisée en étudiant la manière de composer un ensemble de composants préexistants pour créer une composition représentant soit une partie ou bien la totalité d'une application, c'est-à-dire une architecture. La réalisation d'un assemblage consiste à l'écriture du code permettant de réaliser la composition des instances de composants. Lors du conditionnement, un assemblage est introduit dans un paquetage d'assemblage qui peut inclure les composants employés dans l'assemblage ainsi que des ressources propres à l'assemblage telles que des fichiers de configuration ou des fichiers binaires (librairies, images, sons, etc.).

3- Déploiement, Exécution et Administration : le déploiement c'est l'installation des paquetages contenant des compositions et des composants, dans cette étape des activités de configuration peuvent être réalisées. L'exécution est la mise en marche de l'application à ce moment là, l'environnement d'exécution est actif. L'administration inclut l'application de mises à jour, d'adaptation et de maintenance, ainsi que la désinstallation d'une application et de ses composants.

I.6 Style architectural

Dans n'importe quelle activité relative au domaine du génie logiciel, une question qui revient souvent est: comment bénéficier des expériences antérieures pour produire des systèmes plus performants ? Dans le domaine des architectures logicielles, une des manières, selon [NR99], est de classer les architectures par catégories et de définir leurs caractéristiques communes. En effet, un style architectural définit une famille d'architectures logicielles qui sont caractérisées par des propriétés structurelles et sémantiques communes [MKMG97].

Un style architectural est une technique générique facilitant l'expression des solutions structurelles des systèmes. Il comprend un vocabulaire d'éléments conceptuels (les composants et les connecteurs), impose des règles de configuration (ensemble de contraintes) et véhicule une sémantique qui donne un sens (non ambigüe) à la description structurelle [SDK+95, wCGS+02, LPR03, MOO07].

Un style n'est pas une architecture mais une aide générique à l'élaboration de structures architecturales [CN01]. Un style architectural inclut une spécification statique et une spécification dynamique. La partie statique englobe l'ensemble des éléments

(composants et connecteurs) et des contraintes sur ces éléments. La partie dynamique décrit l'évolution possible d'une architecture en réaction à des changements prévus ou imprévus de l'environnement. Un style architectural est précisément défini par un ensemble de caractéristiques telles que : le vocabulaire utilisé (les types de composants et de connexions), les contraintes de configuration (contraintes topologiques appelées aussi patrons structurels), les invariants du style, les exemples communs d'utilisation, les avantages et inconvénients d'utilisation de ce style et les spécialisations communes du style. Un style architectural spécifie aussi les types d'analyse que l'on peut faire sur ses instances (systèmes construits conformément à ce style) [GAO94].

Parmi les principaux styles architecturaux, nous citons le style "client-serveur", le style "publier-souscrire" et le style "pipe and filtre".

I.7 Les avantages à utiliser la POC

Les avantages à utiliser une approche POC pour conduire un projet sont multiples :

- **spécialisation :** L'équipe de développement peut-être divisée en sous-groupes, chacun se spécialisant dans le développement d'un composant
- **sous traitance :** Le développement d'un composant peut-être externalisé, à condition d'en avoir bien réalisé les spécifications au préalable
- **facilité de mise à jour :** La modification d'un composant ne nécessite pas la recompilation du projet complet
- **facilité de livraison/déploiement :** Dans le cas d'une mise à jour, d'un correctif de sécurité, ... alors que le logiciel à déjà été livré au client, la livraison en est facilitée, puisqu'il n'y a pas besoin de re-livrer l'intégralité du projet, mais seulement le composant modifié
- **choix des langages de développement :** Il est possible, dans la plupart des cas, de développer les différents composants du logiciel dans des langages de programmation différents. Ainsi, un composant nécessitant une fonctionnalité particulière pourra profiter de la puissance d'un langage dans un domaine particulier, sans que cela n'influe le développement de l'ensemble du projet

- **productivité :** La réutilisabilité d'un composant permet un gain de productivité non négligeable car elle diminue le temps de développement, d'autant plus que le composant est réutilisé souvent

I.8 Les inconvénients

Bien que l'utilisation de la POC soit réellement appréciable dans la conduite d'un projet de développement, elle n'est pas sans imposer quelques contraintes importantes.

Tout d'abord, la POC est une méthode dont le bénéfice se voit surtout sur le long terme. En effet, lorsque l'on parle de réutilisation, de facilité de déploiement, c'est que le développement est sinon achevé, du moins bien entamé. Mais factoriser un logiciel en composants nécessite un important travail d'analyse. La rédaction des signatures des méthodes devra être particulièrement soignée, car modifier une signature nécessitera de retravailler toutes les portions de codes du projet qui font appel au composant, et l'on perdrait alors les bénéfices de l'indépendance des briques logicielles.

En un mot, si la POC industrialise le développement, la phase de conception du logiciel prendra un rôle encore plus important.

Le fait de ne pas connaître l'implémentation d'un composant (à moins d'avoir accès aux sources), peut également gêner certains chefs de projets qui veulent garder un contrôle total sur leur logiciel.

I.9 Conclusion

Dans ce chapitre, nous avons expliqué ce qui est un composant et à quoi sert une telle programmation basant sur ces concepts architecturaux. Ces architectures à base de composants peuvent changer de structure suite à la variation des exigences des utilisateurs ou à la variation du contexte de l'application.

Dans ce travail, nous nous concentrons sur la modélisation de l'adaptabilité dynamique des composants logiciels au niveau « run-time » (à l'exécution). Nous remarquons dans ce contexte que l'architecture est à la base de la structure et de l'évolution dynamique des systèmes logiciels. Le développement de ces systèmes exige des approches bien établies garantissant la robustesse de la structure de l'application. En plus, l'adaptation de l'architecture doit être contrôlée selon son style architectural afin de préserver des propriétés architecturales du système pendant son évolution et pour ne pas aboutir à des configurations qui risquent de nuire au bon fonctionnement du système.

Chapitre 2
Programmation par aspect

II.1 Introduction

La Programmation Orientée Aspect ("*AOP, Aspect-Oriented Programming*") est apparue pour résoudre le problème d'enchevêtrement des fonctionnalités désirées dans une application, ce qui permet d'éviter l'entrelacement des sous-problèmes techniques ou fonctionnels.

Le but de cette section est d'introduire la programmation orientée aspect telle qu'elle a été définie par Kiczales [Kiczales et al. 01]. Pour ce faire, nous commençons par une présentation générale de cette approche, puis nous proposons un survol de quelques langages et platforms à base d'aspect.

II.2 Définition de l'AOP

La programmation orientée aspect est un paradigme de programmation qui concentre sur des constructs appelés *l'aspect* qui traite les préoccupations des objets, classes, ou méthodes [27].

Cette nouvelle unité de modularité appelée aspect a pour but de modulariser les préoccupations transverses des systèmes complexes au moyen de trois concepts-clés : le point de coupe, le greffon et le tisseur.

Plutôt que d'attaquer le développement d'une application de front, une approche orienté aspect suggère de concevoir des composants logiciels indépendants et fournit des moyens pour les assembler.
Même si cette approche est complémentaire, les techniques de développement par aspects ne doivent pas être confondues avec les techniques d'assemblage par composants. En effet, les techniques de développement par composants se focalisent sur les règles de composition d'un composant particulier par rapport aux autres, alors que les techniques de développement par aspects se focalisent sur la composition d'un ensemble de composants particuliers et correspondant à une préoccupation (comme la sécurité) par rapport à un autre ensemble de composants (le reste de l'application).
En d'autres termes, la POA se focalise sur l'intégration de composants. [26]

II.3 Le but d'une AOP

Dans la programmation structurelle, procédurale et orienté-objet, l'implémentation des besoins non-fonctionnels tels que la sécurité ou la gestion des transactions est dispersée sur plusieurs modules. Ce qui conduit à dupliquer des parties du code dans les différents modules fonctionnels du système. Autrement dit, les besoins non-fonctionnels ne bénéficient pas d'une encapsulation adéquate ni au niveau des modèles de conception ni au niveau des langages de programmation. Cette situation est désignée par l'entre coupage des préoccupations «*crosscutting concern* » ce qui rend difficile la lecture du code source ainsi que sa maintenance. La programmation orienté aspect tente de résoudre les problèmes des approches actuelles en séparant les préoccupations d'une application moyennant des concepts spécifiques. [28]

Donc, le but d'une AOP est de séparer le code de programme lié aux buts principaux (la préoccupation de noyau) du code lié aux buts secondaires (la préoccupation qui se recoupe). [27]

Cette séparation des préoccupations ("*separation of concerns*") techniques des descriptions métier dans une application a permis l'utilisation de l'aspect dans les travaux sur l'adaptation logicielle en considérant l'aspect comme une unité d'adaptation. Ceci a permis de rendre les adaptations indépendantes de l'application elle-même. C'est la propriété qui permet aux programmeurs de prévoir des adaptations sans connaître de manière spécifique les applications qui seront déployées. [29]

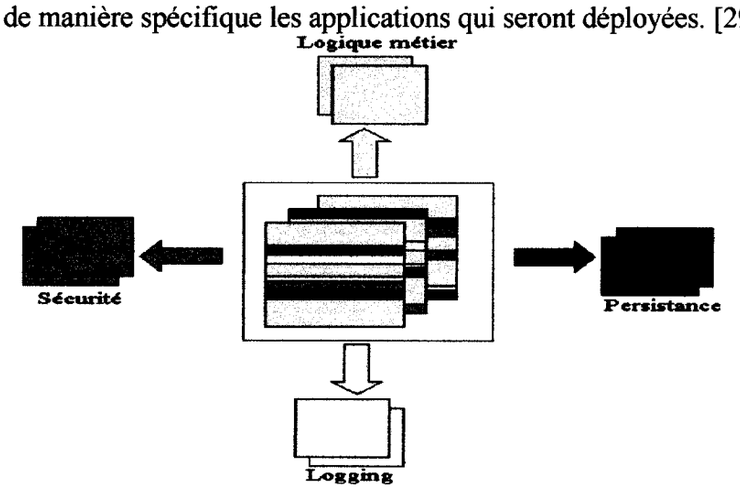

Figure II.1 – Un système vu comme un ensemble de préoccupations

II.4 Avantages par rapport à la programmation objet

La POA diffère grandement de la POO dans la manière dont elle gère les problèmes de recoupements.

La Programmation Orientée Objet : effectue un découpage du projet selon les Objets (implémentation de Classes).

- **Caractéristiques:** emmêlement et éparpillement du code.
- **Conséquences:** à long terme mauvaise traçabilité, faible productivité, faible réutilisation et pauvre qualité du code, évolution complexe.

La Programmation Orientée Aspect : modularise l'implémentation des problématiques entrelacées selon trois étapes:

- **Décomposition par aspects:** les aspects dits "techniques" ou "non-métiers" sont isolés du coeur de l'application,
- **Implémentation des besoins:** les besoins purement métiers sont implémentés (éventuellement, selon une POO)
- **Recomposition en fonction des aspects (tissage ou *wrapping*):** Les aspects précédemment isolés sont, *au runtime*, rattachés au programme (implémentant les besoins). Ces aspects sont alors aisément configurables et évolutifs.
- **Interactions minimisées**: Les aspects sont développés individuellement et ne sont pas nécessaires au fonctionnement du programme métier.
- **Évolutivité**: On peut donc aisément ajouter ou configurer des aspects "montés à chaud" sur le programme en cours d'exécution.
  Si un aspect est déprécié on peut le remplacer par une nouvelle version, *sans rien changer au programme.*
- **Réutilisation du code**: On peut par exemple tisser des aspects (POA) sur un programme conçu en POO.
  Le code purement métier est allégé (les aspects sont codés séparément).

Avec la POA, chaque unité implémentée (Aspect) reste inconsciente du fait que les autres unités peuvent l'instancier en tant qu'aspect.
Par exemple, un module de gestion de carte bancaire ne sait pas que d'autres unités journalisent et authentifient ses opérations.
Cela représente un très fort changement de concept par rapport à la POO.

Une implémentation POA peut employer une autre méthodologie de programmation en tant que méthodologie de base, ce qui permet de cumuler les bénéfices des deux approches. Par exemple, une implémentation POA peut choisir la POO comme système de base pour profiter des bénéfices d'une meilleure implémentation des besoins communs au travers de la POO. Avec une telle implémentation, les besoins individuels peuvent employer les techniques de POO pour chaque besoin identifié. C'est analogue au fait d'utiliser un langage procédural comme base pour un langage orienté objet (C et C++ par exemple).

Finalement, on peut voir la POA comme une alternative, plus poussée, à l'utilisation de *design patterns*, ou autres "astuces", en POO. [26]

POO	POA
Besoins communs spécifiques métier implémentés par modules	Modules séparés pour les problématiques métiers et les aspects techniques
Modules communs implémentés nommés: « Classes »	Modules techniques implémentés nommés: « Aspects »
	Applications
	De nombreux projets de recherche visant à implanter les paradigmes de la POA sont désormais bien avancés et proposent des outils puissants permettant leur mise en œuvre comme dans le domaine des langages ouverts et réflexifs : OpenC++, OpenJava, Javassist... Ou le domaine des ORB, AspectIX... De plus, les dernières générations des outils modernes tendent désormais à intégrer de plus en plus de possibilités pour le support des aspects en standard, et qui ne demandent qu'à être exploitées dans le cadre d'une méthodologie et d'un environnement de développement cohérents.

Tableau II.1 – Évolution logique de la POO, la POA

II.5 Mise en œuvre de la programmation par aspects

Les principaux problèmes posés par la mise en œuvre de la programmation par aspects sont la représentation et la composition des aspects. Nous exposons dans cette partie les différentes approches qui tentent de répondre à ces questions.

Dans un premier temps, nous introduisons le critère de classification des différentes approches étudiées. Nous présentons ensuite les différentes approches pour programmer avec des aspects. Un même exemple (celui de synchronisation des instances de la classe Ouvrage extrait de l'exemple de la librairie électronique) est utilisé pour les illustrer. [30]

II.6 Concepts généraux

Les concepts de la programmation orientée aspect ont été formulés initialement par Kiczales et son équipe en 1997. L'AOP est une méthode de programmation qui permet

de séparer l'implémentation de toutes les exigences, fonctionnelles ou non, d'un logiciel [Kiczales et al. 01]. Le principe est donc de programmer chaque problématique séparément et de définir leurs règles d'intégration en vue de former le système final. Cette disposition favorise la réutilisation et permet ainsi la réduction des délais de développement et de maintenance.

Cependant, la réutilisation n'est pas toujours aisée [Bouraquadi et al. 01]. C'est notamment le cas pour les applications dont la construction ne se limite pas à la simple définition d'un ensemble de services donnés, mais nécessite également la prise en compte de différentes propriétés non-fonctionnelles telles que la distribution, la persistance, etc. Dans de telles applications, l'implémentation des services réalisés et celle des différentes propriétés non- fonctionnelles se trouvent intimement enchevêtrées (Figure II.2 (a)). De ce fait, la réutilisation se trouve compromise.

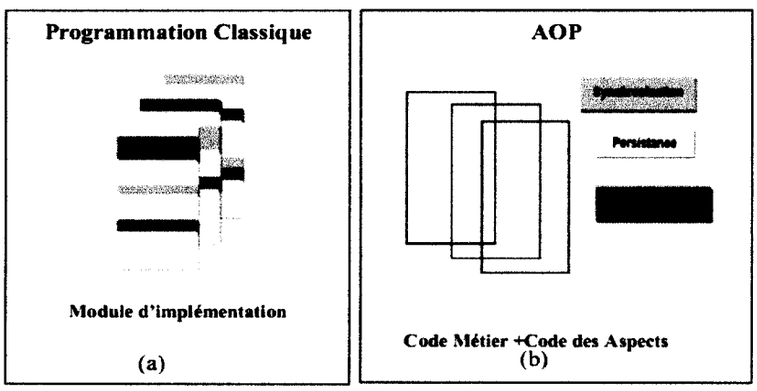

Figure II.2 – programmation classique vis AOP

Dans le but de permettre une meilleure réutilisation et de minimiser la dispersion du code, le paradigme de l'AOP propose de structurer les applications sur la base du concept d'aspect (Figure II.2 (b)).

Un aspect est une abstraction qui définit une structure et un comportement qui se superposent à l'application. Les propriétés non fonctionnelles (sécurité, persistance,...) peuvent être définies par des aspects dont la structure est présentée par la figure II.3.

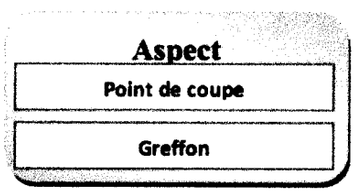

Figure II.3 – Modèle général d'un aspect

II.6.1 Point de coupe (angl Pointcut)

Désigne un ensemble de points de jonction du programme de base à adapter où l'aspect va intervenir.

II.6.2 Point de jonction (angl Join Points)

La composition des aspects avec le programme de base se traduit par l'établissement d'une jonction entre ces unités de modularisation [Hachni 02]. Un point de jonction est donc un point du flot d'exécution de l'application qui peut être contrôlé et où les aspects peuvent intervenir (invocation de méthode, envoi de message, etc.). À un point de coupe correspond un ou plusieurs points de jonction.

II.6.3 Greffon (angl. advice)

Permet de décrire les actions à effectuer lorsqu'un point de coupe est identifié. Le greffon peut être exécuté avant (*before*), après (*after*), ou à la place (*instead*) du point de jonction sélectionné par le point de coupe.

La programmation orientée aspect consiste donc en un ensemble de points de coupe (qui produisent des points de jonction), de greffons et un mécanisme de tissage dont le rôle est de calculer les modifications à effectuer et ceci en injectant le code du greffon dans les points de jonction identifiés. Un greffon a également accès au contexte d'exécution d'un point de jonction donné. Ainsi, en plus d'identifier un point de l'exécution du programme, un point de coupe déclare également les parties du contexte du point de jonction disponible au code de l'aspect.

II.6.4 Aspect

Un aspect consiste en général à plusieurs déclarations de point de coupe et d'un greffon qui leur est associé. De plus, il peut définir son propre état ainsi que ses propres méthodes qui peuvent être utilisées à leur tour dans le code du greffon.

II.6.5 Tissage

L'intégration des aspects dans l'exécution de la fonctionnalité de base est appelée tissage.

Ayant présenté en détail la programmation par aspects, nous présentons dans ce qui suit, AspectJ qui est un exemple de langage de programmation par aspects. [29]

II.7 Les plateforms AOP

Ils existent plusieurs plateforms ou frameworks d'implémentation pour la POA, en Java notamment. JBossAOP, SpringAOP, AspectJAOP, AspectWerkz, JAC ... il existe également des outils pour les langages C (nommé AspectC), C++ (nommé AspectC++), C# ou Smalltalk.

II.7.1 AspectWerkz

AspectWerkz est un projet source ouvert. Il peut tisser par le prétraitement à temps de compilation ou l'emploi de la manipulation du bytecode d'exécution. La langue d'aspect est Java.

AspectWerkz offre à la fois puissance et la simplicité et vous aidera à s'intégrer facilement dans les projets de l'AOP à la fois nouveaux et existants.

AspectWerkz utilise la modification du bytecode à tisser vos classes à projet accumulation de temps, le temps de chargement de la classe ou de l'exécution. Il se branche dans l'utilisation standardisée niveau de la JVM API. Il dispose d'un modèle de jointure riche et très orthogonale point. Aspects, des conseils et des introductions sont rédigées dans un langage simple et vos classes Java cibles peuvent être POJO réguliers. Vous avez la possibilité d'ajouter, de supprimer et de re-structure de conseils ainsi que l'échange de la mise en œuvre de vos présentations lors de l'exécution. Vos points peuvent être définis en utilisant soit des annotations Java 5, Java 1.3/1.4 doclets personnalisés ou un fichier de définition XML simple.

AspectWerkz fournit une API pour utiliser les mêmes aspects de procurations, offrant ainsi une expérience transparente, ce qui permet une transition en douceur pour les utilisateurs familiers avec les mandataires.

II.7.2 JBossAOP

JBoss-POA est l'architecture de Java POA utilisée pour le serveur d'application de JBoss. Il emploie le tissage d'exécution, le Java et le XML comme langues d'aspect.

JBoss AOP est un 100% pur Java aspecté cadre orienté exploitable dans n'importe quel environnement de programmation ou étroitement intégré à notre serveur d'applications. Aspects permettent de plus facilement modulariser votre code de base lors de la programmation orientée objet ordinaire n'a tout simplement pas l'affaire. Il peut fournir un produit de nettoyage de séparation de la logique applicative et le code système. Il fournit un excellent moyen d'exposer des points d'intégration dans votre logiciel. Combiné avec annotations JDK 1.5, il est aussi un excellent moyen d'étendre le langage Java d'une manière propre enfichable plutôt que d'utiliser les annotations uniquement pour la génération de code.

JBoss AOP n'est pas seulement un cadre, mais aussi un ensemble préemballé des aspects qui sont appliquées par l'intermédiaire des annotations, des expressions de point d'action, ou dynamiquement lors de l'exécution. Il s'agit notamment de la mise en cache, la communication asynchrone, les transactions, la sécurité, l'accès distant, et bien d'autres.

II.7.3 AspectJ

Contrairement à ses concurrents Jboss AOP et JAC, AspectJ n'est pas un framework de travail mais il est basé sur des extensions au langage java.

Dans AspectJ, l'unité n'est pas la classe, mais une préoccupation (aspect), qui se partage entre plusieurs classes. Les aspects peuvent être des propriétés, des zones d'intérêt, d'un système et la POA décrit leurs relations, les compose ensemble dans un programme. Les aspects encapsulent un comportement commun à plusieurs classes. Les aspects d'un système peuvent être insérés, changés, supprimés pendant la compilation. [31] [28]

AspectJ est une implémentation open-source pour Java du Xerox PARC.
AspectJ fournit les outils de compilation, débugging et de documentation du code.
Le compilateur d'AspectJ construit une classe compatible avec les spécifications du bytecode Java, permettant à une JVM (Java Virtual Machine) d'interpréter cette classe. [26]

AspectJ est aujourd'hui une référence pour la programmation par aspects telle que définie par Gregor Kiczales. C'est un langage qui étend Java et permet la définition des aspects. Un tisseur est chargé de « mélanger » le code du programme de base écrit en Java avec celui des aspects afin de générer une application complète [Pawlak et al. 05]. Ce langage représente l'application des aspects dans le paradigme de l'Orienté Objet. [3]

Tout ça nous mène à se demander : Pourquoi AspectJ est le plus favorisé?

AspectJ présente divers avantages :

- L'utilisation d'AspectJ peut réduire considérablement la taille d'un programme Java sans perte de performance, et simplifie la conception d'autant.
- Améliore la modularité et la réutilisabilité du code.
- Particulièrement utile pour déboguer de grands projets. [26]

II.7.3.1 Modèle d'Aspect

La définition d'un nouvel aspect en AspectJ est très semblable à la création d'une nouvelle classe [Pawlak et al. 05]. Le mot-clef utilisé est *aspect*. Dans un aspect nous définissons les points de coupes et les codes du greffon (advice) qui induiront les fonctionnalités liées à cet aspect. La représentation d'un aspect est donnée par le listing II.1. La gestion des aspects (instanciation et destruction) se fait automatiquement selon le besoin de l'application (le programmeur n'a pas le droit d'instancier un aspect).

```
public aspect Nom_Aspect {
  pointcut Nom_Method() : // code
// Greffon
before() : Nom_Method() {
  // code
  } }
```

Listing II.1 – Modèle général d'un aspect en AspectJ

Le mot clé *pointcut* permet de définir une coupe qui regroupe un ou plusieurs points de jonction. *AspectJ* supporte différents types de points de jonction qui sont [Pawlak et al. 05] :

- Appel d'une méthode, exécution d'une méthode
- Appel d'un constructeur, exécution d'un constructeur
- Exécution d'un greffon
- Lecture ou écriture d'un attribut
- Exécution d'un bloc de codes statiques particulier
- Initialisation, exception

Chaque action définie par le greffon doit être précédée par la définition du moment de son exécution : avant l'événement (*before*), après (*after*) ou autour de l'événement (*around*). Le mot-clef *around* permet soit de remplacer l'instruction en elle-même, soit d'exécuter du code avant et après le point de jonction.

II.7.3.2 Mécanisme de tissage. Le tisseur *AspectJ* prend en entrée des aspects, l'application et les directives de tissage et produit dans la sortie l'application augmentée des aspects [Levy 06].

Le mécanisme de tissage en *AspectJ* est statique. Les aspects sont tissés au moment de la compilation directement dans le code de l'exécutable ce qui génère soit un fichier source .java ou directement un fichier *bytecode* .class. Les tisseurs statiques ont moins de contraintes de temps d'exécution que les tisseurs dynamiques [Levy 06], mais il y a un risque d'allonger la phase de compilation. L'inconvénient majeur du tissage statique est qu'une fois le tissage terminé, nous retrouvons la même application que dans l'approche OO classique (dispersion du code). Par conséquence, l'apport de l'utilisation d'aspect est perdu au moment de l'exécution.

Kiczales démontre qu'il est possible de minimiser la dispersion du code en regroupant les préoccupations transverses (sécurité, persistance, etc.) dans des entités réutilisables appelées Aspect. Ces dernières sont intégrées dans l'application par le mécanisme du tissage. Plus tard, Kiczales a développé une implémentation des concepts nécessaires pour la programmation orientée aspect qui est le langage AspectJ. L'inconvénient dans cette implémentation est que le mécanisme de tissage est statique, nous perdons alors au moment de l'exécution la séparation des préoccupations de l'application de base. Une autre limite de ce modèle est le manque de mécanisme de traitement des interactions entre les aspects. Il n'y a aucun mécanisme pour la composition des aspects. Les aspects sont appliqués d'une manière séquentielle. De ce faite, l'application tissée peut

avoir des comportements non souhaités dans le cas où nous avons plusieurs aspects qu'ils s'appliquent au même endroit. [29]

II.7.4 JAC (Java Aspect Components)

JAC est un outil de RAD (Développement Applicatif Rapide) open-source.
C'est un cadre logiciel pour la programmation orientée aspect en Java.
Contrairement aux langages tels AspectJ qui adoptent une approche essentiellement basée sur les classes, JAC adopte une granularité objet et ne requiert aucune extension syntaxique du langage Java.

Un programme orienté aspect avec JAC est un ensemble d'objets d'aspects qui peuvent être dynamiquement déployés et retirés sur des objets applicatifs en cours d'exécution. Les objets d'aspects peuvent définir trois types de méthodes d'aspects: des méthodes encapsulantes (qui encapsulent des méthodes applicatives et permettent d'exécuter du code avant et après la méthode encapsulée), des méthodes de rôle (qui ajoutent de nouvelles fonctionnalités aux objets applicatifs) et des gestionnaires d'exceptions.
Le problème de la composition d'aspects est traité à l'aide d'un contrôleur d'encapsulation bien défini qui spécifie pour chaque objet encapsulé, au moment de l'encapsulation, à l'exécution ou dans les deux cas, l'ordre d'exécution des objets d'aspects.

"JAC (Java Aspect Components) est un serveur d'applications qui offre une alternative open-source (licence LGPL) aux environnements J2EE (Java 2 Enterprise Edition) pour la réalisation en Java d'applications distribuées. JAC est également un environnement de développement qui intègre un atelier UML (Unified Modelling Language). Cet atelier permet au développeur de modéliser en UML le cœur métier de son application et d'en auto-générer le code. Ces classes Java, exécutées au sein du conteneur JAC, peuvent bénéficier indépendamment de services techniques tels que la persistance des données, l'authentification, le déploiement, le load-balancing ou la gestion de sessions. Basé sur la technologie de la programmation orientée aspect, JAC permet de parfaitement séparer ces préoccupations techniques de la logique métier de l'application." Renaud Pawlak, AOPSYS. [26]

II.7.5 Spring AOP

Spring est considéré comme un conteneur dit « léger ». La raison de ce nommage est très bien expliquée par Erik Gollot dans l'introduction du document Introduction au framework Spring

« SPRING est effectivement un conteneur dit " léger ", c'est-à-dire une infrastructure similaire à un serveur d'application J2EE. Il prend donc en charge la création d'objets et la mise en relation d'objets par l'intermédiaire d'un fichier de configuration qui décrit les objets à fabriquer et les relations de dépendances entre ces objets. Le gros avantage par rapport aux serveurs d'application est qu'avec SPRING, les classes n'ont pas besoin d'implémenter une quelconque interface pour être prises en charge par le framework (au contraire des serveurs d'application J2EE et des EJBs). C'est en ce sens que SPRING est qualifié de conteneur " léger ". »

Spring s'appuie principalement sur l'intégration de trois concepts clés :

- l'inversion de contrôle est assurée de deux façons différentes : la recherche de dépendances et l'injection de dépendances
- la programmation orientée aspect
- une couche d'abstraction.

La couche d'abstraction permet d'intégrer d'autres frameworks et bibliothèques avec une plus grande facilité. Cela se fait par l'apport ou non de couches d'abstraction spécifiques à des frameworks particuliers. Il est ainsi possible d'intégrer un module d'envoi de mails en toute facilité.

L'inversion de contrôle :

1) La recherche de dépendance : consiste pour un objet à interroger le conteneur, afin de trouver ses dépendances avec les autres objets. C'est un cas de fonctionnement similaire aux EJBs.
2) L'injection de dépendances : cette injection peut être effectuée de trois manières possibles :

- L'injection de dépendance via le constructeur.
- L'injection de dépendance via les modificateurs (setters).
- L'injection de dépendance via une interface.

Les deux premières sont les plus utilisées par Spring.

Nous utilisons Spring dans notre dernier chapitre pour la modélisation de notre exemple d'application.

II.8 Modèle d'adaptabilité

Les motivations et la mise en œuvre de la notion d'aspect a évolué depuis sa création en 1997. L'AOP avait pour but original de minimiser la dispersion du code en réduisant le nombre d'instructions à écrire pour réaliser un programme complexe. En analysant de plus près les préoccupations visées, on a constaté que ces préoccupations étaient souvent de nature non fonctionnelle. Mais, cela dépendait du domaine d'application que l'on considère. Ainsi, on peut également considérer l'aspect comme une simple modularité transverse. En tant que telle, un aspect peut être déclenché non seulement par le programme, mais également par des événements exogènes (sollicitation de l'environnement physique extérieur au programme). Enfin, nous pouvons considérer l'aspect comme un paradigme de programmation à partir duquel on peut construire entièrement une application.

En particulier, les motivations des aspects ont changé. En effet, de l'analyse d'un problème de complexité d'un système d'information, nous nous intéressons à présent à la complexité de l'adaptation dynamique logicielle en informatique ambiante.

De manière générale, nous constatons que l'utilisation des aspects sert à modifier à priori le comportement d'une application existante. Dans notre travail on dispose d'un ensemble d'aspects d'assemblage prédéfinis dont les points de coupe servent de conditions de déclenchement de l'adaptation et donc, de mécanisme d'adaptabilité. [32]

II.9 Conclusion

Nous avons présenté dans ce chapitre le paradigme de programmation orienté aspects et nous avons passé en revue ses principes, ses concepts et ses avantages comparés au paradigme orienté objets. Il ressort de cette présentation que ce paradigme, relativement nouveau, représente une évolution certaine dans le domaine de la modélisation et de la séparation avancée des préoccupations en général.

En étendant la programmation orientée objet () avec la notion d'aspect, la POA permet d'atteindre une parfaite modularité des programmes informatiques complexes c'est pour cela qu'elle intéresse de plus en plus de services informatiques de grands groupes industriels et reçoit une grande importance dans les travaux de recherche dans le monde de développement logiciel de nos jours.

Chapitre 3
Adaptation dynamique

II.1 Introduction

La Programmation Orientée Aspect ("*AOP, Aspect-Oriented Programming*") est apparue pour résoudre le problème d'enchevêtrement des fonctionnalités désirées dans une application, ce qui permet d'éviter l'entrelacement des sous-problèmes techniques ou fonctionnels.

Le but de cette section est d'introduire la programmation orientée aspect telle qu'elle a été définie par Kiczales [Kiczales et al. 01]. Pour ce faire, nous commençons par une présentation générale de cette approche, puis nous proposons un survol de quelques langages et platforms à base d'aspect.

II.2 Définition de l'AOP

La programmation orientée aspect est un paradigme de programmation qui concentre sur des constructs appelés *l'aspect* qui traite les préoccupations des objets, classes, ou méthodes [27].

Cette nouvelle unité de modularité appelée aspect a pour but de modulariser les préoccupations transverses des systèmes complexes au moyen de trois concepts-clés : le point de coupe, le greffon et le tisseur.

Plutôt que d'attaquer le développement d'une application de front, une approche orienté aspect suggère de concevoir des composants logiciels indépendants et fournit des moyens pour les assembler.
Même si cette approche est complémentaire, les techniques de développement par aspects ne doivent pas être confondues avec les techniques d'assemblage par composants. En effet, les techniques de développement par composants se focalisent sur les règles de composition d'un composant particulier par rapport aux autres, alors que les techniques de développement par aspects se focalisent sur la composition d'un ensemble de composants particuliers et correspondant à une préoccupation (comme la sécurité) par rapport à un autre ensemble de composants (le reste de l'application).
En d'autres termes, la POA se focalise sur l'intégration de composants. [26]

II.3 Le but d'une AOP

Dans la programmation structurelle, procédurale et orienté-objet, l'implémentation des besoins non-fonctionnels tels que la sécurité ou la gestion des transactions est dispersée sur plusieurs modules. Ce qui conduit à dupliquer des parties du code dans les différents modules fonctionnels du système. Autrement dit, les besoins non-fonctionnels ne bénéficient pas d'une encapsulation adéquate ni au niveau des modèles de conception ni au niveau des langages de programmation. Cette situation est désignée par l'entre coupage des préoccupations «*crosscutting concern* » ce qui rend difficile la lecture du code source ainsi que sa maintenance. La programmation orienté aspect tente de résoudre les problèmes des approches actuelles en séparant les préoccupations d'une application moyennant des concepts spécifiques. [28]

Donc, le but d'une AOP est de séparer le code de programme lié aux buts principaux (la préoccupation de noyau) du code lié aux buts secondaires (la préoccupation qui se recoupe). [27]

Cette séparation des préoccupations ("*separation of concerns*") techniques des descriptions métier dans une application a permis l'utilisation de l'aspect dans les travaux sur l'adaptation logicielle en considérant l'aspect comme une unité d'adaptation. Ceci a permis de rendre les adaptations indépendantes de l'application elle-même. C'est la propriété qui permet aux programmeurs de prévoir des adaptations sans connaître de manière spécifique les applications qui seront déployées. [29]

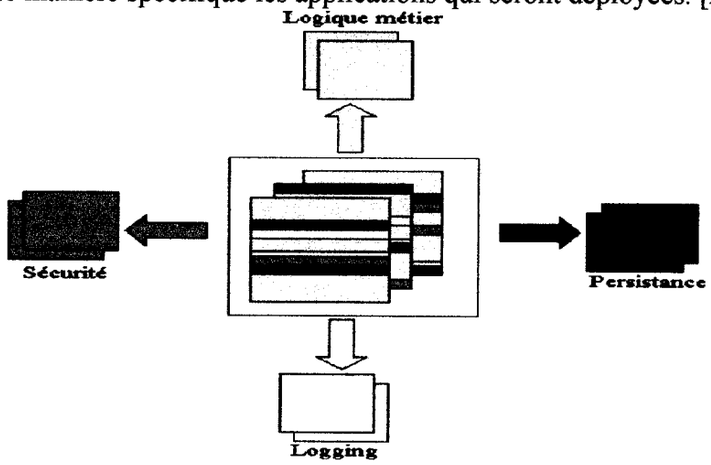

Figure II.1 – Un système vu comme un ensemble de préoccupations

II.4 Avantages par rapport à la programmation objet

La POA diffère grandement de la POO dans la manière dont elle gère les problèmes de recoupements.

La Programmation Orientée Objet : effectue un découpage du projet selon les Objets (implémentation de Classes).

- **Caractéristiques:** emmêlement et éparpillement du code.
- **Conséquences:** à long terme mauvaise traçabilité, faible productivité, faible réutilisation et pauvre qualité du code, évolution complexe.

La Programmation Orientée Aspect : modularise l'implémentation des problématiques entrelacées selon trois étapes:

- **Décomposition par aspects:** les aspects dits "techniques" ou "non-métiers" sont isolés du coeur de l'application,
- **Implémentation des besoins:** les besoins purement métiers sont implémentés (éventuellement, selon une POO)
- **Recomposition en fonction des aspects (tissage ou *wrapping*):** Les aspects précédemment isolés sont, *au runtime*, rattachés au programme (implémentant les besoins). Ces aspects sont alors aisément configurables et évolutifs.
- **Interactions minimisées**: Les aspects sont développés individuellement et ne sont pas nécessaires au fonctionnement du programme métier.
- **Évolutivité**: On peut donc aisément ajouter ou configurer des aspects "montés à chaud" sur le programme en cours d'exécution.
  Si un aspect est déprécié on peut le remplacer par une nouvelle version, *sans rien changer au programme.*
- **Réutilisation du code**: On peut par exemple tisser des aspects (POA) sur un programme conçu en POO.
  Le code purement métier est allégé (les aspects sont codés séparément).

Avec la POA, chaque unité implémentée (Aspect) reste inconsciente du fait que les autres unités peuvent l'instancier en tant qu'aspect.
Par exemple, un module de gestion de carte bancaire ne sait pas que d'autres unités journalisent et authentifient ses opérations.
Cela représente un très fort changement de concept par rapport à la POO.

Une implémentation POA peut employer une autre méthodologie de programmation en tant que méthodologie de base, ce qui permet de cumuler les bénéfices des deux approches. Par exemple, une implémentation POA peut choisir la POO comme système de base pour profiter des bénéfices d'une meilleure implémentation des besoins communs au travers de la POO. Avec une telle implémentation, les besoins individuels peuvent employer les techniques de POO pour chaque besoin identifié. C'est analogue au fait d'utiliser un langage procédural comme base pour un langage orienté objet (C et C++ par exemple).

Finalement, on peut voir la POA comme une alternative, plus poussée, à l'utilisation de *design patterns*, ou autres "astuces", en POO. [26]

POO	POA
Besoins communs spécifiques métier implémentés par modules	Modules séparés pour les problématiques métiers et les aspects techniques
Modules communs implémentés nommés: « Classes »	Modules techniques implémentés nommés: « Aspects »
	Applications
	De nombreux projets de recherche visant à implanter les paradigmes de la POA sont désormais bien avancés et proposent des outils puissants permettant leur mise en œuvre comme dans le domaine des langages ouverts et réflexifs : OpenC++, OpenJava, Javassist... Ou le domaine des ORB, AspectIX... De plus, les dernières générations des outils modernes tendent désormais à intégrer de plus en plus de possibilités pour le support des aspects en standard, et qui ne demandent qu'à être exploitées dans le cadre d'une méthodologie et d'un environnement de développement cohérents.

Tableau II.1 – Évolution logique de la POO, la POA

II.5 Mise en œuvre de la programmation par aspects

Les principaux problèmes posés par la mise en œuvre de la programmation par aspects sont la représentation et la composition des aspects. Nous exposons dans cette partie les différentes approches qui tentent de répondre à ces questions.

Dans un premier temps, nous introduisons le critère de classification des différentes approches étudiées. Nous présentons ensuite les différentes approches pour programmer avec des aspects. Un même exemple (celui de synchronisation des instances de la classe Ouvrage extrait de l'exemple de la librairie électronique) est utilisé pour les illustrer. [30]

II.6 Concepts généraux

Les concepts de la programmation orientée aspect ont été formulés initialement par Kiczales et son équipe en 1997. L'AOP est une méthode de programmation qui permet

de séparer l'implémentation de toutes les exigences, fonctionnelles ou non, d'un logiciel [Kiczales et al. 01]. Le principe est donc de programmer chaque problématique séparément et de définir leurs règles d'intégration en vue de former le système final. Cette disposition favorise la réutilisation et permet ainsi la réduction des délais de développement et de maintenance.

Cependant, la réutilisation n'est pas toujours aisée [Bouraquadi et al. 01]. C'est notamment le cas pour les applications dont la construction ne se limite pas à la simple définition d'un ensemble de services donnés, mais nécessite également la prise en compte de différentes propriétés non-fonctionnelles telles que la distribution, la persistance, etc. Dans de telles applications, l'implémentation des services réalisés et celle des différentes propriétés non- fonctionnelles se trouvent intimement enchevêtrées (Figure II.2 (a)). De ce fait, la réutilisation se trouve compromise.

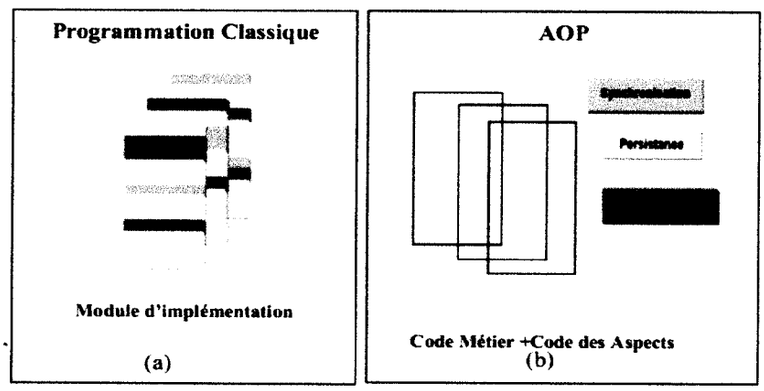

Figure II.2 – programmation classique vis AOP

Dans le but de permettre une meilleure réutilisation et de minimiser la dispersion du code, le paradigme de l'AOP propose de structurer les applications sur la base du concept d'aspect (Figure II.2 (b)).

Un aspect est une abstraction qui définit une structure et un comportement qui se superposent à l'application. Les propriétés non fonctionnelles (sécurité, persistance,...) peuvent être définies par des aspects dont la structure est présentée par la figure II.3.

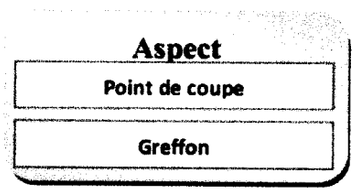

Figure II.3 – Modèle général d'un aspect

II.6.1 **Point de coupe** (angl Pointcut)

Désigne un ensemble de points de jonction du programme de base à adapter où l'aspect va intervenir.

II.6.2 **Point de jonction** (angl Join Points)

La composition des aspects avec le programme de base se traduit par l'établissement d'une jonction entre ces unités de modularisation [Hachni 02]. Un point de jonction est donc un point du flot d'exécution de l'application qui peut être contrôlé et où les aspects peuvent intervenir (invocation de méthode, envoi de message, etc.). À un point de coupe correspond un ou plusieurs points de jonction.

II.6.3 **Greffon** (angl. advice)

Permet de décrire les actions à effectuer lorsqu'un point de coupe est identifié. Le greffon peut être exécuté avant (*before*), après (*after*), ou à la place (*instead*) du point de jonction sélectionné par le point de coupe.

La programmation orientée aspect consiste donc en un ensemble de points de coupe (qui produisent des points de jonction), de greffons et un mécanisme de tissage dont le rôle est de calculer les modifications à effectuer et ceci en injectant le code du greffon dans les points de jonction identifiés. Un greffon a également accès au contexte d'exécution d'un point de jonction donné. Ainsi, en plus d'identifier un point de l'exécution du programme, un point de coupe déclare également les parties du contexte du point de jonction disponible au code de l'aspect.

II.6.4 **Aspect**

Un aspect consiste en général à plusieurs déclarations de point de coupe et d'un greffon qui leur est associé. De plus, il peut définir son propre état ainsi que ses propres méthodes qui peuvent être utilisées à leur tour dans le code du greffon.

II.6.5 Tissage

L'intégration des aspects dans l'exécution de la fonctionnalité de base est appelée tissage.

Ayant présenté en détail la programmation par aspects, nous présentons dans ce qui suit, AspectJ qui est un exemple de langage de programmation par aspects. [29]

II.7 Les plateforms AOP

Ils existent plusieurs plateforms ou frameworks d'implémentation pour la POA, en Java notamment. JBossAOP, SpringAOP, AspectJAOP, AspectWerkz, JAC ... il existe également des outils pour les langages C (nommé AspectC), C++ (nommé AspectC++), C# ou Smalltalk.

II.7.1 AspectWerkz

AspectWerkz est un projet source ouvert. Il peut tisser par le prétraitement à temps de compilation ou l'emploi de la manipulation du bytecode d'exécution. La langue d'aspect est Java.

AspectWerkz offre à la fois puissance et la simplicité et vous aidera à s'intégrer facilement dans les projets de l'AOP à la fois nouveaux et existants.

AspectWerkz utilise la modification du bytecode à tisser vos classes à projet accumulation de temps, le temps de chargement de la classe ou de l'exécution. Il se branche dans l'utilisation standardisée niveau de la JVM API. Il dispose d'un modèle de jointure riche et très orthogonale point. Aspects, des conseils et des introductions sont rédigées dans un langage simple et vos classes Java cibles peuvent être POJO réguliers. Vous avez la possibilité d'ajouter, de supprimer et de re-structure de conseils ainsi que l'échange de la mise en œuvre de vos présentations lors de l'exécution. Vos points peuvent être définis en utilisant soit des annotations Java 5, Java 1.3/1.4 doclets personnalisés ou un fichier de définition XML simple.

AspectWerkz fournit une API pour utiliser les mêmes aspects de procurations, offrant ainsi une expérience transparente, ce qui permet une transition en douceur pour les utilisateurs familiers avec les mandataires.

II.7.2 JBossAOP

JBoss-POA est l'architecture de Java POA utilisée pour le serveur d'application de JBoss. Il emploie le tissage d'exécution, le Java et le XML comme langues d'aspect.

JBoss AOP est un 100% pur Java aspecté cadre orienté exploitable dans n'importe quel environnement de programmation ou étroitement intégré à notre serveur d'applications. Aspects permettent de plus facilement modulariser votre code de base lors de la programmation orientée objet ordinaire n'a tout simplement pas l'affaire. Il peut fournir un produit de nettoyage de séparation de la logique applicative et le code système. Il fournit un excellent moyen d'exposer des points d'intégration dans votre logiciel. Combiné avec annotations JDK 1.5, il est aussi un excellent moyen d'étendre le langage Java d'une manière propre enfichable plutôt que d'utiliser les annotations uniquement pour la génération de code.

JBoss AOP n'est pas seulement un cadre, mais aussi un ensemble préemballé des aspects qui sont appliquées par l'intermédiaire des annotations, des expressions de point d'action, ou dynamiquement lors de l'exécution. Il s'agit notamment de la mise en cache, la communication asynchrone, les transactions, la sécurité, l'accès distant, et bien d'autres.

II.7.3 AspectJ

Contrairement à ses concurrents Jboss AOP et JAC, AspectJ n'est pas un framework de travail mais il est basé sur des extensions au langage java.

Dans AspectJ, l'unité n'est pas la classe, mais une préoccupation (aspect), qui se partage entre plusieurs classes. Les aspects peuvent être des propriétés, des zones d'intérêt, d'un système et la POA décrit leurs relations, les compose ensemble dans un programme. Les aspects encapsulent un comportement commun à plusieurs classes. Les aspects d'un système peuvent être insérés, changés, supprimés pendant la compilation. [31] [28]

AspectJ est une implémentation open-source pour Java du Xerox PARC.
AspectJ fournit les outils de compilation, débugging et de documentation du code.
Le compilateur d'AspectJ construit une classe compatible avec les spécifications du bytecode Java, permettant à une JVM (Java Virtual Machine) d'interpréter cette classe. [26]

AspectJ est aujourd'hui une référence pour la programmation par aspects telle que définie par Gregor Kiczales. C'est un langage qui étend Java et permet la définition des aspects. Un tisseur est chargé de « mélanger » le code du programme de base écrit en Java avec celui des aspects afin de générer une application complète [Pawlak et al. 05]. Ce langage représente l'application des aspects dans le paradigme de l'Orienté Objet. [3]

Tout ça nous mène à se demander : Pourquoi AspectJ est le plus favorisé?

AspectJ présente divers avantages :

- L'utilisation d'AspectJ peut réduire considérablement la taille d'un programme Java sans perte de performance, et simplifie la conception d'autant.
- Améliore la modularité et la réutilisabilité du code.
- Particulièrement utile pour déboguer de grands projets. [26]

II.7.3.1 Modèle d'Aspect

La définition d'un nouvel aspect en AspectJ est très semblable à la création d'une nouvelle classe [Pawlak et al. 05]. Le mot-clef utilisé est *aspect*. Dans un aspect nous définissons les points de coupes et les codes du greffon (advice) qui induiront les fonctionnalités liées à cet aspect. La représentation d'un aspect est donnée par le listing II.1. La gestion des aspects (instanciation et destruction) se fait automatiquement selon le besoin de l'application (le programmeur n'a pas le droit d'instancier un aspect).

```
public aspect Nom_Aspect {
  pointcut Nom_Method() : // code
// Greffon
before() : Nom_Method() {
  // code
  } }
```

Listing II.1 – Modèle général d'un aspect en AspectJ

Le mot clé *pointcut* permet de définir une coupe qui regroupe un ou plusieurs points de jonction. *AspectJ* supporte différents types de points de jonction qui sont [Pawlak et al. 05] :

- Appel d'une méthode, exécution d'une méthode
- Appel d'un constructeur, exécution d'un constructeur
- Exécution d'un greffon
- Lecture ou écriture d'un attribut
- Exécution d'un bloc de codes statiques particulier
- Initialisation, exception

Chaque action définie par le greffon doit être précédée par la définition du moment de son exécution : avant l'événement (*before*), après (*after*) ou autour de l'événement (*around*). Le mot-clef *around* permet soit de remplacer l'instruction en elle-même, soit d'exécuter du code avant et après le point de jonction.

II.7.3.2 Mécanisme de tissage. Le tisseur *AspectJ* prend en entrée des aspects, l'application et les directives de tissage et produit dans la sortie l'application augmentée des aspects [Levy 06].

Le mécanisme de tissage en *AspectJ* est statique. Les aspects sont tissés au moment de la compilation directement dans le code de l'exécutable ce qui génère soit un fichier source .java ou directement un fichier *bytecode* .class. Les tisseurs statiques ont moins de contraintes de temps d'exécution que les tisseurs dynamiques [Levy 06], mais il y a un risque d'allonger la phase de compilation. L'inconvénient majeur du tissage statique est qu'une fois le tissage terminé, nous retrouvons la même application que dans l'approche OO classique (dispersion du code). Par conséquence, l'apport de l'utilisation d'aspect est perdu au moment de l'exécution.

Kiczales démontre qu'il est possible de minimiser la dispersion du code en regroupant les préoccupations transverses (sécurité, persistance, etc.) dans des entités réutilisables appelées Aspect. Ces dernières sont intégrées dans l'application par le mécanisme du tissage. Plus tard, Kiczales a développé une implémentation des concepts nécessaires pour la programmation orientée aspect qui est le langage AspectJ. L'inconvénient dans cette implémentation est que le mécanisme de tissage est statique, nous perdons alors au moment de l'exécution la séparation des préoccupations de l'application de base. Une autre limite de ce modèle est le manque de mécanisme de traitement des interactions entre les aspects. Il n'y a aucun mécanisme pour la composition des aspects. Les aspects sont appliqués d'une manière séquentielle. De ce faite, l'application tissée peut

avoir des comportements non souhaités dans le cas où nous avons plusieurs aspects qu'ils s'appliquent au même endroit. [29]

II.7.4 JAC (Java Aspect Components)

JAC est un outil de RAD (Développement Applicatif Rapide) open-source.
C'est un cadre logiciel pour la programmation orientée aspect en Java.
Contrairement aux langages tels AspectJ qui adoptent une approche essentiellement basée sur les classes, JAC adopte une granularité objet et ne requiert aucune extension syntaxique du langage Java.

Un programme orienté aspect avec JAC est un ensemble d'objets d'aspects qui peuvent être dynamiquement déployés et retirés sur des objets applicatifs en cours d'exécution. Les objets d'aspects peuvent définir trois types de méthodes d'aspects: des méthodes encapsulantes (qui encapsulent des méthodes applicatives et permettent d'exécuter du code avant et après la méthode encapsulée), des méthodes de rôle (qui ajoutent de nouvelles fonctionnalités aux objets applicatifs) et des gestionnaires d'exceptions.
Le problème de la composition d'aspects est traité à l'aide d'un contrôleur d'encapsulation bien défini qui spécifie pour chaque objet encapsulé, au moment de l'encapsulation, à l'exécution ou dans les deux cas, l'ordre d'exécution des objets d'aspects.

"JAC (Java Aspect Components) est un serveur d'applications qui offre une alternative open-source (licence LGPL) aux environnements J2EE (Java 2 Enterprise Edition) pour la réalisation en Java d'applications distribuées. JAC est également un environnement de développement qui intègre un atelier UML (Unified Modelling Language). Cet atelier permet au développeur de modéliser en UML le cœur métier de son application et d'en auto-générer le code. Ces classes Java, exécutées au sein du conteneur JAC, peuvent bénéficier indépendamment de services techniques tels que la persistance des données, l'authentification, le déploiement, le load-balancing ou la gestion de sessions. Basé sur la technologie de la programmation orientée aspect, JAC permet de parfaitement séparer ces préoccupations techniques de la logique métier de l'application." Renaud Pawlak, AOPSYS. [26]

II.7.5 Spring AOP

Spring est considéré comme un conteneur dit « léger ». La raison de ce nommage est très bien expliquée par Erik Gollot dans l'introduction du document Introduction au framework Spring

« SPRING est effectivement un conteneur dit " léger ", c'est-à-dire une infrastructure similaire à un serveur d'application J2EE. Il prend donc en charge la création d'objets et la mise en relation d'objets par l'intermédiaire d'un fichier de configuration qui décrit les objets à fabriquer et les relations de dépendances entre ces objets. Le gros avantage par rapport aux serveurs d'application est qu'avec SPRING, les classes n'ont pas besoin d'implémenter une quelconque interface pour être prises en charge par le framework (au contraire des serveurs d'application J2EE et des EJBs). C'est en ce sens que SPRING est qualifié de conteneur " léger ". »

Spring s'appuie principalement sur l'intégration de trois concepts clés :

- l'inversion de contrôle est assurée de deux façons différentes : la recherche de dépendances et l'injection de dépendances
- la programmation orientée aspect
- une couche d'abstraction.

La couche d'abstraction permet d'intégrer d'autres frameworks et bibliothèques avec une plus grande facilité. Cela se fait par l'apport ou non de couches d'abstraction spécifiques à des frameworks particuliers. Il est ainsi possible d'intégrer un module d'envoi de mails en toute facilité.

L'inversion de contrôle :

1) La recherche de dépendance : consiste pour un objet à interroger le conteneur, afin de trouver ses dépendances avec les autres objets. C'est un cas de fonctionnement similaire aux EJBs.
2) L'injection de dépendances : cette injection peut être effectuée de trois manières possibles :

- L'injection de dépendance via le constructeur.
- L'injection de dépendance via les modificateurs (setters).
- L'injection de dépendance via une interface.

Les deux premières sont les plus utilisées par Spring.

Nous utilisons Spring dans notre dernier chapitre pour la modélisation de notre exemple d'application.

II.8 Modèle d'adaptabilité

Les motivations et la mise en œuvre de la notion d'aspect a évolué depuis sa création en 1997. L'AOP avait pour but original de minimiser la dispersion du code en réduisant le nombre d'instructions à écrire pour réaliser un programme complexe. En analysant de plus près les préoccupations visées, on a constaté que ces préoccupations étaient souvent de nature non fonctionnelle. Mais, cela dépendait du domaine d'application que l'on considère. Ainsi, on peut également considérer l'aspect comme une simple modularité transverse. En tant que telle, un aspect peut être déclenché non seulement par le programme, mais également par des événements exogènes (sollicitation de l'environnement physique extérieur au programme). Enfin, nous pouvons considérer l'aspect comme un paradigme de programmation à partir duquel on peut construire entièrement une application.

En particulier, les motivations des aspects ont changé. En effet, de l'analyse d'un problème de complexité d'un système d'information, nous nous intéressons à présent à la complexité de l'adaptation dynamique logicielle en informatique ambiante.

De manière générale, nous constatons que l'utilisation des aspects sert à modifier à priori le comportement d'une application existante. Dans notre travail on dispose d'un ensemble d'aspects d'assemblage prédéfinis dont les points de coupe servent de conditions de déclenchement de l'adaptation et donc, de mécanisme d'adaptabilité. [32]

II.9 Conclusion

Nous avons présenté dans ce chapitre le paradigme de programmation orienté aspects et nous avons passé en revue ses principes, ses concepts et ses avantages comparés au paradigme orienté objets. Il ressort de cette présentation que ce paradigme, relativement nouveau, représente une évolution certaine dans le domaine de la modélisation et de la séparation avancée des préoccupations en général.

En étendant la programmation orientée objet () avec la notion d'aspect, la POA permet d'atteindre une parfaite modularité des programmes informatiques complexes c'est pour cela qu'elle intéresse de plus en plus de services informatiques de grands groupes industriels et reçoit une grande importance dans les travaux de recherche dans le monde de développement logiciel de nos jours.

Chapitre 4
Implémentation

IV.1 Introduction

Dans ce chapitre nous allons donner un exemple de déroulement d'une adaptation dynamique d'une application à base de composants en introduisant la notion de tissage d'aspect dynamique pour réaliser cette adaptation, les composants utilisés sont implémentés sur le framework Spring.

Nous allons illustrer le fonctionnement de tissage d'aspect pour notre application et les méthodes d'adaptations utilisées dans notre exemple d'adaptation.

Nous allons aussi comparer le temps de réponses entre deux programmes, l'un qui utilise une adaptation dynamique et l'autre sans le mécanisme d'adaptation pour en conclure la performance de cette approche.

IV.2 L'architecture globale du système

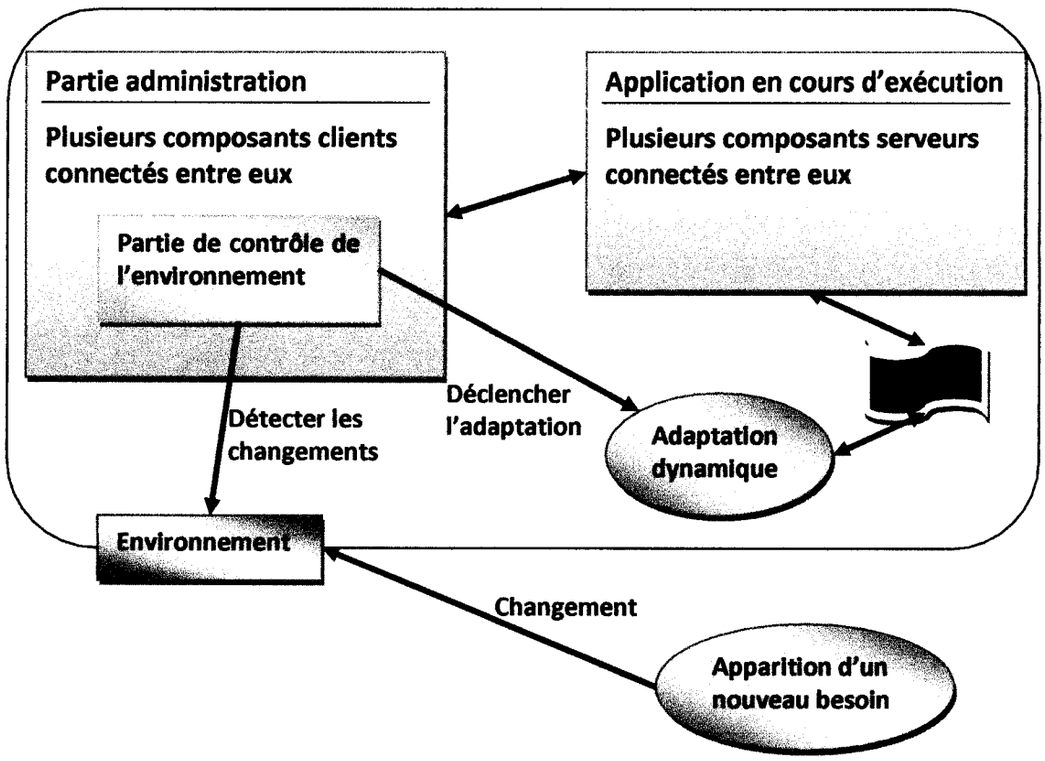

Figure IV.1 – Architecture générale du système

Notre architecture à base de composants est découpée en deux grandes parties : La partie administration qui exprime les besoins des clients et l'autre partie c'est l'application qui fournie les résultats pour les requêtes renvoyées.

Chaque composant fourni et/ou requiert une ou plusieurs interfaces fournies et requises, cela permet de séparer proprement le composant du reste de la conception et de réutiliser le composant dans d'autres contextes en associant ses interfaces aux différents composants.

Le mécanisme de tissage engage une stratégie de détection d'un point de coupure dans le programme de notre application pour déclencher l'adaptation lors de l'apparition d'un changement dans l'environnement pris en compte par un mécanisme de contrôle d'environnement.

IV.3 Exemple d'implémentation

L'adaptation consiste à remplacer le composant Composant2 par le composant Composant3. Le composant Composant2 (Serveur) est un simple composant fournissant un service de calcul (somme de deux nombres entiers) au composant Composant1 (Client). (Voir Figure IV.2)

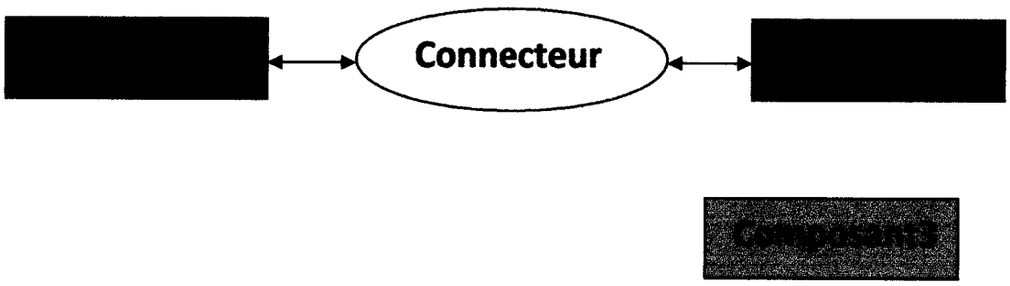

Figure IV.2 – exemple d'application

Le composant client requiert l'interface suivante :

public interface Result {

public double somme(double val1, double val2);}

Le composant serveur fournit l'interface suivante :

public interface calcul {

public double ajout(double val1, double val2);}

Le client fait donc appel à la méthode « somme » de l'interface « Result », qui n'est pas fournie par le serveur. Le rôle du connecteur est de résoudre l'incompatibilité entre le client (la méthode somme) et le serveur (la méthode ajout).

Le connecteur implémente l'interface « Result », le code de la méthode « somme », fait appel donc à la méthode « ajout » de l'interface « calcul ». Cet appel se fait à travers un script, ce qui rend le corps de la méthode « somme », plus flexible et apte d'être modifié d'une manière dynamique.

Le composant serveur « Compsant2 » fournit le service de calcul de somme suivant :

```
public class calcul1 {

    public int somme(int intVal, int intVal2) {

        System.out.println(" appel composant 1 ... calcul de la somme des deux
    valeurs:" + intVal + "," + intVal2);

        return (intVal+intVal2); }}
```

Listing IV.1 – Le code de la méthode implémentée par le Composant1

Un autre client mais toujours en utilisant la même interface requière par le composant « Composant1 » à travers le connecteur, a besoin d'un autre service de calcul mais pas la somme des deux entiers, il veut le produit de ces deux entiers donc il fait appel à cet instant là à une autre interface fournie par composant serveur « Composant3 ».

Le composant serveur « Composant3 » fournit le service de calcul de produit suivant :

```
public class calcul2 {

   public int produit(int val, int valu){

        System.out.println("appel composant 2... calcul de produit des deux
    valeurs: "+ val + ","+ valu);

        return val*valu; }}
```

Listing IV.2 – code de la méthode implémentée par le Composant2

IV.4 Le principe d'adaptation

L'adaptation dans notre application est faite dynamiquement par tissage d'aspect.

Le tissage est introduit après la redirection d'appel vers le composant serveur « Composant3 » en définissant les points de jonctions et les points de coupes qui auront lieu dans le code du connecteur, car c'est ce dernier qui va appeler les méthodes du nouveau composant serveur et supprimer la première liaison avec le premier serveur, cette redirection d'appel est déclenché automatiquement quand l'utilisateur saisie sa

nouvelle demande dans un fichier à part(fichier texte dans notre exemple) et dynamiquement car c'est manipulé pendant l'exécution du programme.

Le code de l'adaptation dynamique est le suivant :

```
public class test extends DynamicMethodMatcherPointcut {
   public boolean matches(Method method, Class clazz, Object[] objects){
          String filePath = "fichier.txt";
          int testVal=0;
               try{
                 Scanner scanner=new Scanner(new File(filePath));
                 while (scanner.hasNextLine()) {
                    String line = scanner.nextLine();
                    testVal=Integer.parseInt(line); }
              scanner.close();}
                 catch(Exception e){}
          return (testVal > 50); }}

public class adv implements MethodInterceptor {
   public Object invoke(MethodInvocation methodInvocation) throws Throwable
{
          calcul2 c2 = new calcul2();
          Object [] arg=methodInvocation.getArguments();
          System.out.println(c2.produit((Integer)arg[0],(Integer)arg[1]));
          return null; }}
```

Listing IV.3 – code de l'interface d'adaptation

La méthode « matches » qui a comme paramètres : Method, Class et Object[], est invoquée seulement si les deux arguments retournent « vrai » pour la méthode condidate et la classe cible et si le booléen retourné appelé le « isRuntime() » retourne vrai.

Dans notre exemple, la valeur à tester est lise d'un fichier texte, si la compatibilité entre les deux versions fournie et désiré alors la méthode « matches » est invoquée, et le nouveau code est exécuté.

IV.5 Interface graphique de l'application

Nous avons développé une interface graphique, pour simplifier l'administration de l'application. Cette interface permet à l'utilisateur de l'application de :

1. visualiser l'architecture de l'application pour illustrer l'adaptation.
2. contrôler l'adaptation du composant.
4. modifier la valeur du test.

La Figure IV.3 illustre cette interface.

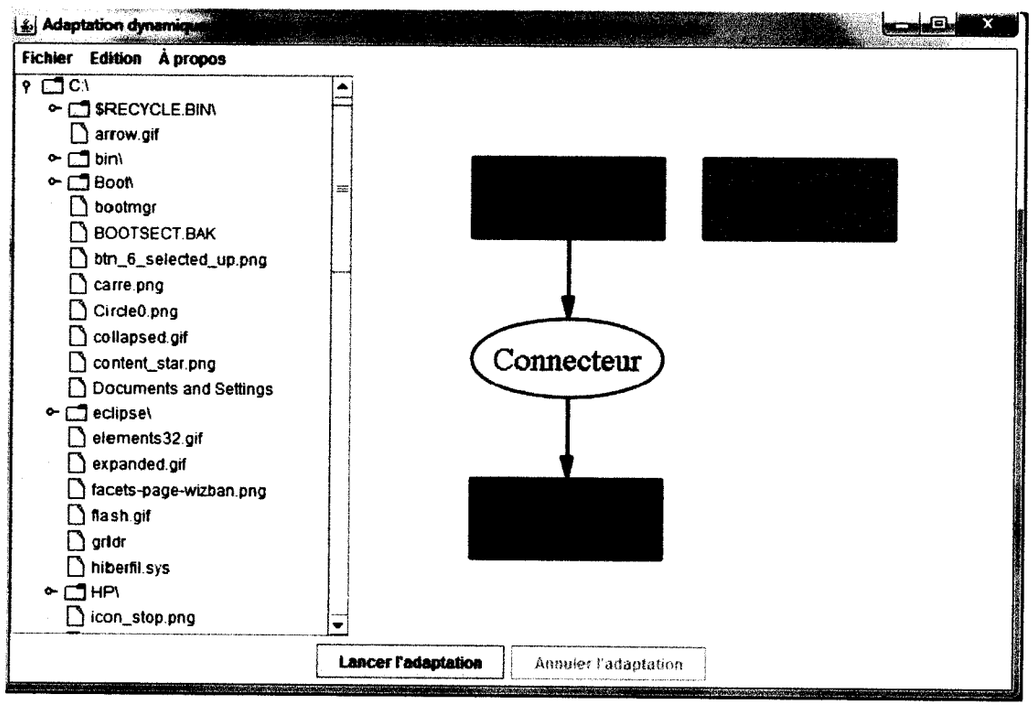

Figure IV.3 – Interface de l'application

Les graphes de cette interface sont faits à l'aide de l'API "Grappa" (A Java Graph Package) [54]. Cette API facilite la création et la gestion de graphes dans les applications Java.

IV.6 Evaluation d'un exemple d'application

Nous présentons, dans cette section, une évaluation de notre approche d'adaptation.

Le test de performances dans notre cas est fait en comparant deux versions de la même application, une implémente le mécanisme d'adaptation, et l'autre sans ce mécanisme.

L'exemple d'application est celui utilisé par SpringAOP : un simple client qui envoie une entier et un serveur qui la renvoie en détectant le nom de la méthode et affichant son nom plus la valeur renvoyée par le client.

Les tests d'évaluation sont effectués sur un micro-ordinateur d'un processeur Dual-Core de 2.10 GHz et 2,00 Go de RAM.

Notre expérimentation montre l'influence d'une adaptation sur l'exécution de l'application.

L'adaptation consiste à remplacer le composant serveur par une autre version, cette version renvoie la valeur entière renvoyée par le client multipliée par soi-même. Cette expérience permet d'évaluer la durée d'exécution d'une adaptation.

Le temps d'adaptation moyen est la différence entre le temps de repense moyen avec adaptation et le temps de réponse moyen sans adaptation. D'une manière formelle :

Tadapt_moy = Trpm_Aa - Trpm_Sa ; d'où :

Tadapt_moy : le temps d'adaptation moyen.

Trpm_Aa : Temps de réponse avec adaptation.

Trpm_Sa : Temps de réponse sans adaptation.

Le tableau suivant résume les résultats de l'expérimentation :

Nombre de requêtes	**Temps de réponse sans adaptation**	**Temps de réponse avec adaptation**	**Temps d'adaptation**
1000	42 ms	97 ms	55 ms
9000	224 ms	571 ms	347 ms
20000	531 ms	1178 ms	647 ms
100000	2399 ms	6519 ms	4120 ms
500000	17055 ms	30584 ms	13529 ms
1000000	65755 ms	71888 ms	6133 ms

Tableau IV.1 – Calcul de temps d'exécution

Temps moyen d'adaptation
4138,5 ms ≈ 4 secondes

Tableau IV.2 – Temps moyen d'adaptation

On remarque dans les deux tableaux en dessus que le temps moyen d'adaptation est égale à peu prés à quatre secondes. Ce temps ne dépend pas du nombre de requêtes exécutées, on voit par exemple les deux dernières lignes dans le tableau : le temps de réponse pour 500000 requêtes et supérieur à celui de 1000000 requêtes.

On peut conclure que le temps de réponse n'est pas le seul critère pour déterminer une adaptation, ils existent d'autres critères comme le taux de message altérés qui a une relation avec le mécanisme de préservation des canaux de communication pendant une

adaptation, et le temps de réponse en fonctions de nombres de requêtes en calculant le taux d'augmentation …

Nous n'avons pas discuté toutes les expérimentations possibles pour évaluer une adaptation dynamique à force du temps, nous souhaitions le faire pour une meilleure évaluation.

IV.6 Conclusion

Nous avons montré qu'une adaptation dynamique utilisant la notion d'aspect nous a permis de séparer le code et de faciliter l'utilisation d'une application en générale grâce à la notion de tissage du code.

On peut aussi dire qu'une adaptation dynamique permet de généraliser un code d'un programme en spécifiant chaque tâche souhaitée par un client séparément et de la tisser dynamiquement selon les changements des besoins des clients.

CONCLUSION GENERALE

CONCLUSION GENERALE

Dans notre mémoire nous avons abordé le domaine d'adaptation dynamique des composants en se basant sur la programmation par aspect, nous avons illustré notre idée par un exemple simple de calcul puis nous faisions un tableau de comparaison des temps d'exécutions pour voir l'utilité de cette démarche.

Notre travail s'est basé sur le contexte de programmation à base de composants, nous avons donné une vue générale sur ce domaine en introduisant les notions de la programmation par aspect pour se charger de séparer le code métier de notre application de celui de l'adaptation.

Notre exemple d'application se base sur des notions génériques de ce domaine et a pour but de ne pas avoir besoin d'arrêter l'application pendant son exécution.

Nous avons choisi ce contexte pour notre mémoire car il présente un sujet important pour les informaticiens, il offre une souplesse de travail pour la programmation et évite d'écrire plusieurs programmes pour des différents besoins, donc il permet de généraliser un programme d'exécution, mais en lui adaptant pendant l'exécution selon les besoins et les changements de son environnement.

Des perspectives telles que : trouver les paramètres pour automatiser les adaptations via des expressions exprimés par les utilisateurs; Généraliser le code Aspect pour qu'il maintienne plus des applications diverties. Ces perspectives peuvent mener à des travaux futurs pour bien explorer notre travail d'étude dans ce domaine très intéressant.

Bibliographie

[1] : J. Tournier. QINNA : une architecture à base de composants pour la gestion de la qualité de services dans les systèmes embarqués mobiles : thèse de doctorat de l'Institut Nationale des Science Appliqué de Lyon, France. juillet 2005.

[2] : N. Mehta, N. Medvidovic, S. Phadke. Towards a Taxonomy of Software Connectors. ICSE 2000, Limerick, Ireland. ACM 23000 1-58113-206-9/00/06..

[3] : M. Rakic, N. Medvidovic. Increasing the Confidence in Off-the-Shelf Components: A Software Connector-Based Approach. SSR'01, May 18-20, 2001, Toronto, Ontario, Canada. 2001 ACM 1-58113-358-8 /01/0005.

[4] : M. Shin, D. Cooke. Connector-Based Self-Healing Mechanism for Components of a Reliable System. DEAS 2005, May 21, 2005, St. Louis, Missouri, USA. 2005 ACM 1-59593-039-6/05/0005.

[5] : D. Garlan, « Software Architecture: a Road Map ». in The Future of Software Engineering, A. Finkekstein (Ed), ACM Press, 2000.

[6] : Allen R.J, « A Formal Approach to Software Architecture », PhD Thesis, CMU-CS-97-144, 1997. School of Computer Science Carnegie Mellon University Pittsburgh.

[7] Garlan D., Monroe R.T., Wile D., « Acme: Architectural Description of Component-Based Systems », Foundations of Component-Based Systems, Leavens G.T. and Sitaraman M. (eds), Cambridge University Press, 2000.

[8] Luckham D.C., Kenney J.J., Augustin L.M., Vera J., Bryan D., Mann W., « Specification and Analysis of System Architecture Using Rapide », IEEE Transactions on Software Engineering, Vol.21, No.4, April 1995.

[9] Shaw M., DeLine R., Klein D.V., Ross T.L., Young D.M., and Zelesnik G., « Abstractions for Software Architecture and Tools to Support Them », IEEE Trans. Software Eng., vol.21, no.4, p.314-335, April 1995.

[10] Taylor R.N., Medvidovic N., Anderson K.M., Whitehead Jr.E.J., Robbins J.E., Nies K.A., Oreizy P., Dubrow D.L., « A component- and message-based architectural

style for GUI software », IEEE Transactions on Software Engineering, Vol.22, NO.6, JUNE 1996.

[11] Magee J., Dulay N., Eisenbach S., Kramer J., « Specifying Distributed Software Architectures », the 5th European Software Engineering Conference (ESEC'95), Spain, 26 September 1995.

[12] labo sun: http://www.labo-sun.com/resource-fr-essentiels-835-6-java-j2ee-ejb-2-les-entreprise-java-bean-javabeans-.htm, 2006.

[13] CORBA Component Model Specification OMG Available Specification Version 4.0 formal/06-04-01, April 2006.

[14] Microsoft. .Net. Technical report, Microsoft, http ://www.microsoft.com/net/, 09 avril 2006.

[15] E. Bruneton, T. Coupaye, J.B. Stefani : The Fractal Component Model. The ObjectWeb Consortium Specification, Draft 5, 2004, version 2.0-3.

[16] H.CERVANTES, vers un modèle à composants orienté services pour supporter la disponibilité dynamique : thèse de doctorat de l'Université Joseph Fourier, Mars 2004.

[17] labo sun: http://www.labo-sun.com/resource-fr-essentiels-835-6-java-j2ee-ejb-2-les-

entreprise-java-bean-javabeans-.htm, 2006.

[18] CORBA Component Model Specification OMG Available Specification Version 4.0 formal/06-04-01, April 2006.

[19] Projet ACCORD .Assemblage de composants par contrat en environnement ouvert et réparti : Le modèle de composants CORBA, Livrable 1.1-4, mai 2002.

[20] : O.NANO. Un Modèle De Réécriture Pour L'intégration De Services : thèse de doctorat de l'Université De Nice - Sophia Antipolis, décembre 2004.

[21] : K. Macedo De Amorim. Modélisation d'aspects qualité de service en UML : application aux composants logiciels : thèse de doctorat de l'université de Renne 1, mai 2004.

[22] Projet ACCORD .Assemblage de composants par contrat en environnement ouvert et réparti. Livrable 1.1 – 1 juin 2002.

[23] Intergiciel et Construction d'Applications Réparties : 19 janvier 2007,p155-203, http://creativecommons.org/licenses/by-nc-nd/2.0/fr/deed.fr.

[24] Christophe Lauer "Introduction à Microsoft .NET", http://www.dotnet-fr.org..

[25] Kung-Kiu Lau : Software Component Models ; ICSE'06, May 20–28, Shanghai, China. ACM 1-59593-085-X/06/0005, 2006.

[26] : http://www-igm.univ-mlv.fr/~dr/XPOSE2002/JAC/html/POA3.htm

[27] : NGUYEN Manh Tien : Programmation Orientée Aspect ; Juillet 2005, Hanoï, Institut de la Francophonie pour l'Informatique

[28] : BOUTAGHANE Rafika : Test Automatique Des Préoccupations Dans Les Langages A Aspects, 2010, Sekikda

[29] : Sana FATHALLAH : Tissages Multiples d'Aspects d'Assemblage, Application à l'adaptation logicielle pour l'Informatique Ambiante, 16/O7/2009, Tunisie

[30] : Noury M. N. Bouraqadi-Saâdani—Thomas Ledoux : Le point sur la programmation par aspects, 1999, www.easycomp.org;

[31] : http //www.scriptol.org/fr-aspectj.html

[32] : Daniel CHEUNG-FOO-WO : ADAPTATION DYNAMIQUE PAR TISSAGE D'ASPECTS D'ASSEMBLAGE, 5 mars 2009, France

[33] : Jérémy Buisson — Françoise André — Jean-Louis Pazat : Adaptation dynamique de codes parallèles, publié dans "Journées Composants, Lille : France (2004) ;

[34] : Sana FATHALLAH, Stéphane LAVIROTTE, Jean-Yves TIGLI, Kamel HAMROUNI ; Equipe RAINBOW : Tissage multiples d'aspects d'assemblage ; application à l'adaptation logicielle pour l'informatique ambiante, Novembre 2009 , Laboratoire d'informatique de signaux et systèmes de Sophia Antipolis- UNSA-CNRS .

[35] R.S. Fabry : "How to design a system in which modules can be changed on the fly", Proc. 2nd lnt. Conf. on Soft. Eng., pp. 470-476 (1976).

[36] : Abdelmadjid Ketfi, Noureddine Belkhatir, Pierre-Yves Cunin : Adaptation Dynamique Concepts et Expérimentations, 2002 , Grenoble Cedex 9 France

[37] A.Senart. canevas logiciel pour la construction d'infrastructures logicielles dynamiquement Adaptable : thèse de doctorat de l'Institut Nationale polytechnique de Grenoble, France. Novembre 2003.

[38] Aksit.M, Choukair.Z, «Dynamic, Adaptive and Reconfigurable Systems Overview and Prospective Vision», Proceedings of the 23 rd International Conference on Distributed Computing Systems Workshops (ICDCSW'03) 0-7695-1921-0/03 IEEE computer society 2003.

[39] Belabed Amine: Adaptabilité dynamique des Applications à base de composants Etude de cas : la solution proposée ;Oran , Algérie

[40] Tarak CHAARI : Adaptation d'applications pervasives dans des environnements multi-contextes, 28/09/2007 Laboratoire d'Informatique en Image et Systèmes d'information (LIRIS) , Lyon, France.

[41] Guillaume Grondin, Noury Bouraqadi et Laurent Vercouter : Assemblage automatique et adaptation d'applications à base de composants, Département IA, École des Mines de Douai 941 rue Charles Bourseul – B.P. 10838, 59508 Douai Cedex, France {grondin,bouraqadi}@ensm-douai.fr, http://csl.ensm-douai.fr,Centre G2I, École des Mines de Saint-Étienne,158 cours Fauriel, 42023 Saint-Étienne Cedex 02, France, vercouter@emse.fr, http://www.emse.fr/~vercouter

[42] Imen Tounsi : Une Approche pour la Modélisation et la Vérification des Politiques d'Adaptation pour le Style P/S, 15 mai 2010, Université de Sfax, Tunisie.

[43] Chouarfia Abdellah1, Belabed Amine : Une approche orientée connecteurs pour l'adaptation dynamique des applications à base de composants, Université Mohamed Boudiaf Département d'informatique Oran, Algérie ;

[44] Aksit, M., Choukair, Z.: Dynamic, Adaptive and Reconfigurable Systems Overview and Prospective Vision. ICDCSW'03 0-7695-1921-0/03 IEEE computer society (2003).

[45] Occello, A., Dery-Pinna, AM.: Safe Runtime Adaptations of Components: a UML Metamodel with OCL Constraints. Electronic Notes in Theoretical Computer Science FUSE (2004).

[46] Chouarfia, A., Yahlali, M.: Quality of software components assembly, i9th IBIMA conference, Marrakech, Maroc 04-06 Jan (2008).

[47] Oreizy, P., Medvidovic, Taylor, N.R.N.: Architecture-Based Runtime Software Evolution. ICSE '98. Kyoto, Japan, April 19-25, (1998).

[48] Yang, Q., Yang, X., Xu, M.: A Mobile Agent Approach to Dynamic Architecture-based Software Adaptation.ACM SIGSOFT Software Engineering Notes, V.31 Number 3, May (2006).

[49] Ketfi, A., Belkhatir, N., Cunin, P. : Adaptation Dynamique Concepts et Expérimentations. ICSSEA (2002).

[50] Plasil, F., Balek, D., Janecek, R.: DCUP: Dynamic Component Updating in Java/CORBA Environment. Tech. Report No. 97/10, Dep. Of SW Engineering, Charles University, (1997).

[51] Chefrour, D., André, F.: ACEEL: modèle de composants auto adaptatifs. Journées sur les systèmes à composants adaptables et extensibles, (2001).

[52] Aubert, O., Beugnard, A.: Adaptive Strategy Design Pattern. Laboratoire d'Informatique des Télécommunications, ENST Bretagne, France June 25, (2001).

[53] Ayed, D., Berbers, Y.: Dynamic Adaptation of CORBA Component Based Applications. SAC '07, March 1115, Seoul, Korea (2007).

[54] Oriol, M., Serugendo, G.: disconnected service architecture for unanticipated runtime evolution of code. IEEE Proceedings, Special Issue on Unanticipated Software Evolution, (2004).

[55] Oriol, M.: Primitives for the Dynamic Evolution of Component Based Applications. SAC'07 March 1115, Seoul, Korea, (2007).

[56] Kiczales, G., des Rivieres, J. and Bobrow, G.: The Art of the Meta-Object Protocol. MIT Press, Cambridge (MA), USA, (1991).

[57] PROJET RNTL ARCAD D1.1 - État de l'art sur l'adaptabilité, décembre (2001).

[58] Pinto, M., Fuentes, L., Troy, J.M.: DAOP-ADL: An Architecture Description Language for Dynamic Component and Aspect Based Development. GPCE 2003, LNCS 2830. (C) Springer-Verlag Berlin Heidelberg (2003).