Université de Technologie de Troyes – UTT

DEA Réseaux Avancés de Connaissances et Organisations

Assemblage de Composants Logiciels : Les Composants-Composites

Mémoire de stage

Soutenu le 11 juillet 2003

Par

Salah HADJAB
émail : hajab@ensm-douai.fr

Pour l'obtention du

Diplôme d'Études Approfondies

Responsable du stage :
Djamel-Abdelhak SERIAI
émail : seriai@ensm-douai.fr

Équipe CSL

Ecole Des Mines De Douai

Responsable de la filière :
Dominique Gaiti
émail : Dominique.Gaiti@ut.fr

Équipe LM2S

Université de Technologie de Troyes

Remerciements

Il m'est difficile d'exprimer en peu de mots mon estime à l'égard de Monsieur Djamel-Abdelhak SERIAI. Je tiens à ce qu'il reçoive à travers ces mots ma gratitude et mes plus profonds remerciements pour son accueil, son encadrement et son soutien. Ses conseils et son expérience m'ont guidé tout au long de mon stage, et je lui en suis reconnaissant.

Je remercie Monsieur Noury BOURAQADI pour son accueil, ses remarques et son encouragement.

Je remercie Monsieur Philippe HASBROUCQ de m'avoir accepté au sein du département Génie Informatique et Productique de l'Ecole des Mines de Douai.

J'adresse mes remerciements à Monsieur Jean-Loup CORDONNIER, Madame Christine DELILLE ainsi qu'à tout le personnel du département GIP pour sa convivialité.

Je tiens à remercier l'équipe CSL toute entière pour son accueil chaleureux. Je remercie tous les membres pour leur disponibilité et leurs conseils.

Enfin, je remercie l'ensemble des enseignants que j'ai eux au cours de mon DEA : Marc LEMERCIER, Christophe DONIAT, Amalia TODIRASCU et surtout Mlle Dominique GAITI, mon responsable de filière, qui me fera l'honneur de juger le travail réalisé pendant mon stage.

Résumé

Le paradigme "*composant*" est apparu en réponse aux limites de l'approche de conception par objets. Il a introduit une nouvelle méthode pour la conception des applications logicielles. Cette méthode dénommée CBSE (Component-Based Software Engineering) est basée sur l'assemblage d'entités logicielles préfabriquées appelées composants. Elle a pour finalité de simplifier le travail des développeurs en favorisant la réutilisabilité. Mais cela n'est pas toujours évident, car les composants disponibles ne sont pas souvent compatibles. La technologie des connecteurs logiciels est ainsi proposée pour adapter les composants hétérogènes afin de faciliter leur assemblage. Néanmoins, certains types d'assemblage n'ont pas encore été étudiés. Parmi ces types, l'assemblage des composants via la relation de composition pour pouvoir concevoir des composants dits *composants-composites*.
Nous avons consacré le travail de ce stage à étudier et à introduire ce type d'assemblage dans les modèles de composants logiciels. Ainsi, après une étude comparative des trois principaux modèles de composants logiciels, à savoir EJB (Enterprise Java Beans), COM+(Commen Object Model) et CCM (CORBA Component Model), nous introduisons la notion de composition dans les modèles de composants logiciels. La réification de cette relation se fait par le nouveau type de connecteur que nous proposons, *connecteur de composition*. Ensuite, nous présentons la version enrichie du langage IDL3 (Interface Definition Language) proposé pour le modèle CCM, que nous baptisons IDL3+, associée à la déclaration des composants composites. Enfin, nous présentons l'implémentation réalisée en CCM permettant de valider le modèle proposé. Cette implémentation est proposée en deux versions : sans et avec notre proposition.

Mots-Clés : Composant logiciel, Connecteur, Assemblage, Composant-composite, Connecteur Composition, CCM.

Abstract

The "component" paradigm is appeared in response to the object-oriented approach limits. It introduced a new approach for the software applications design. This approach called CBSE (Component-Based Software Engineering) is based on the assembly of software entities called components. It has as a finality to simplify designer's work by supporting software reutilisability. Nevertheless, this is not always obvious, because available components are not often compatibles. Thus, the software connector technology has been proposed to adapt heterogeneous components in order to facilitate their assembly. However, certain types of assembly are not yet studied. Among these ones, we cited the components assembly using the composition link to design components known as composite-component.

We have consecrated this work to study and to introduce this particular type of software component assembly. Thus, after a comparative study of the three principal software component models, namely EJB (Enterprise Java Beans), COM+ (Commen Object Model) and CCM (CORBA Component Model), we introduce the composition properties to be integrated to software component models. Thus, the reification of this relation is done by a new type of connector, which we called composition connector. Then, we present the enriched version of IDL3 language (Interface Language Definition) suggested for CCM model. This version, witch we called IDL3+, is devoted to declaration of software composite-components. Lastly, we present the prototype, which we have developed with its two versions: with and without the composition connectors integrated CCM Component Model.

Keys-Words: Software Component, Connector, Assembly, Composite-component, Composition Connector, CCM.

Table des matières

Introduction

1. Contexte général

L'évolution des systèmes informatiques a été très rapide ces dernières années. Cette évolution est notamment perceptible au travers de l'évolution des méthodes de programmation. En effet, la programmation a évolué du mode *procédural* vers la programmation par *Objet*. Aujourd'hui la technologie des *composants logiciels* est de plus en plus répandue pour les applications logicielles qui deviennent de plus en plus complexes. À l'origine de la naissance de l'approche "*composant*" sont les limites de la technologie objet. En fait, même si l'approche Objet a marqué un changement important dans le monde de la programmation, ce paradigme a présenté des limites inhérentes telles que :

1. La structure de l'application est généralement peu visible : l'ensemble de fichiers de codes est nécessaire ;
2. La difficulté de la modification (ajout / suppression) de fonctionnalités si cela s'avère nécessaire ;
3. La construction de l'application est prise totalement en charge par le programmeur : construction des différents modules, définition des instances et interconnexions des modules, etc.

L'ingénierie de conception des applications logicielles à base de composants CBSE[1] [Belloir et al., 2001], est ainsi née du besoin de faciliter, le plus possible, **la réutilisation** du code, ce qui permet de concevoir des applications robustes et réduites très rapidement. En fait, pour créer une application, les développeurs interconnectent, d'une manière homogène et à l'aide d'une plate-forme, des entités logicielles (les composants) dont les sources (origines) sont éventuellement hétérogènes [Winston et al., 1987]. De cette façon, au lieu de réécrire systématiquement le même type de code en commençant de zéro (*start from scratch*), ils peuvent utiliser des composants logiciels standards, et ajuster simplement les éléments variant d'une application à l'autre. En résumé, la méthode CBSE vise principalement à :

1. **Réduire le temps** de développement des applications logicielles ;
2. **Diminuer le coût** de production et de maintenance des logiciels ;
3. **Simplifier le développement** des applications par l'utilisation des parties de code préexistantes ;
4. **Améliorer la qualité et la fiabilité des logiciels** qui résulte de l'utilisation des composants programmés par des spécialistes et testés par des experts ;
5. **Fournir des possibilités de décomposition**. Cela permet de modifier plus facilement les applications au fur et à mesure de l'évolution des besoins des utilisateurs.

2. Problématique et objectif

Aujourd'hui, il existe trois principaux modèles de composants logiciels, à savoir le modèle EJB (Enterprise Java Beans), COM+ (Commen Object Model) et CCM (CORBA[2] Component Model). Ces modèles permettent de concevoir des applications par l'assemblage de composants. Cependant, l'assemblage direct des composants n'est pas toujours évident, car on peut avoir des composants qui ne sont pas compatibles. La technologie des connecteurs logiciels est ainsi proposée pour adapter les composants hétérogènes afin de faciliter leur assemblage.

[1] CBSE: Component-Based Software Engineering.

[2] CORBA : Common Object Request Broker Archtecture. C'est une architecture basée sur un bus ORB (Object Request Broker). Ce dernier est chargé d'assurer les collaborations entre les applications.

Par ailleurs, la conception de certains types d'applications nécessite l'utilisation d'une relation particulière qui impose une sémantique bien définie pour l'assemblage de composants. Cette relation, qui est la relation de composition, sert à construire un nouveau type de composant logiciel, appelé **composant-composite**. En fait, selon cette façon contraignante d'assemblage, un composant logiciel composite sera formé par l'agrégation d'un ensemble de composants logiciels, appelés ses composants. La relation entre le composant composite et ses composants est la **relation de composition** qui assure une sémantique bien propre. Par exemple, si nous voulons concevoir un composant-composite voiture, il est nécessaire d'utiliser quatre roues. De la même manière, nous ne pouvons pas créer une voiture avec trois moteurs. Ainsi, il est nécessaire de définir un ensemble de règles afin de pouvoir assurer la cohérence de la sémantique du composant composite voiture.

Nous avons constaté que la relation de composition n'a pas été étudiée dans les modèles de composants logiciels existants[3]. Ainsi, l'objectif de ce stage de DEA est d'introduire la notion de composition pour ces modèles. Pour cela, les modèles en question sont étudiés pour montrer leurs avantages et leurs limites, en particulier, par rapport à l'assemblage par composition. Cette étude nous permet aussi de choisir le meilleur modèle à utiliser comme support pour la concrétisation de notre proposition.

3. Exemple support

Tout au long de ce rapport, nous allons utiliser comme exemple support celui de la conception d'un composant-composite "voiture". La figure ci-dessous illustre la relation de composition dans le modèle objet, présentée en langage UML [19]. Cet exemple concerne une voiture composée d'une carrosserie, d'un moteur et de quatre roues. La carrosserie, pour sa part, est composée de deux à sept sièges et de un à quatre airbags. Les liens entre la voiture et les autres entités sont des liens de composition, chacun présente une sémantique spécifique de la relation de composition. Par exemple, le moteur fait partie intégrante (dépendance forte) de la voiture, donc sa destruction implique la destruction de la voiture. Aussi, une voiture, a besoin pour rouler d'avoir un nombre bien défini de roues.

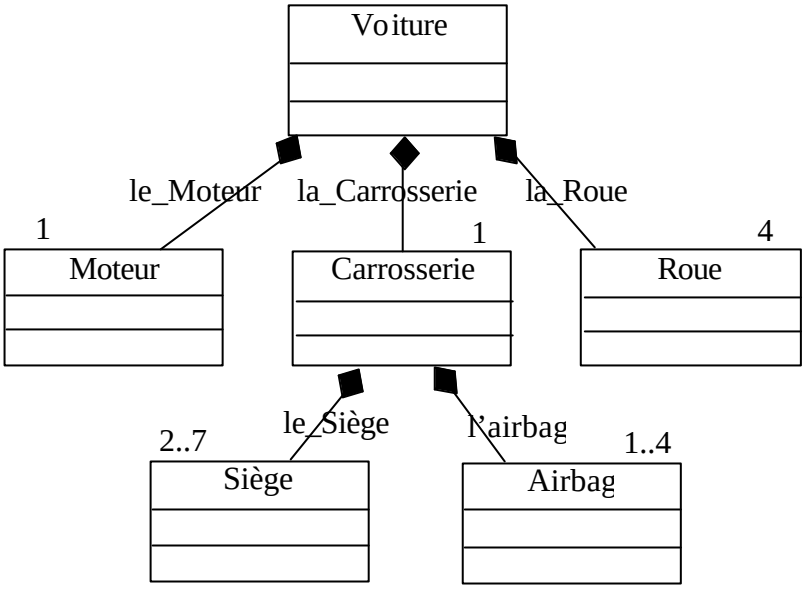

Figure 1 : La relation de composition de la voiture

[3] Certains travaux récents concernant le projet ACCORD[ACCORD 2003] commencent à étudier cette relation.

4. Organisation du rapport

Le reste de ce document est organisé de la manière suivante : le chapitre 1 présente en premier lieu les notions de base liées aux composants logiciels suivis par un état de l'art des modèles de composants, à savoir EJB, COM+ et CCM, en abordant certains points essentiels pour l'étude de notre problématique. Nous terminons ce chapitre par une comparaison entre ces modèles et le positionnement de notre travail par rapport aux travaux existants. Le chapitre 2 présente notre proposition : l'introduction de la notion de composition dans les modèles de composants logiciels. La mise en œuvre que nous proposons de la relation de composition pour le modèle CCM est présentée dans le chapitre 3. Enfin, nous concluons et nous citons les perspectives que nous envisageons comme suite à ce sujet de stage.

Enfin, nous adoptons dans la suite de ce document les conventions d'écriture suivantes :

- Les références [chiffre] indiquent des adresses Internet ;
- Les termes en Gras désignent des concepts ou des notions fondamentales ;
- Les définitions sont présentées en Italique.

Chapitre 1 : Les Composants logiciels

1.1 Introduction

Nous consacrons la première partie de ce chapitre à l'introduction des notions de base, à savoir les composants logiciels et leurs architectures, les langages de description d'architectures et les connecteurs logiciels. Dans la seconde partie nous présentons brièvement les trois principaux modèles de composants existants afin de pouvoir dégager les différences entre eux, puis d'établir une évaluation de ces modèles pour déterminer certaines de leurs limitations, de positionner notre travail et de choisir un modèle support pour notre implémentation.

1.2 Notions de base

1.2.1 Qu'est-ce qu'un composant logiciel?

En dépit de l'absence d'une définition consensuelle de ce qu'est un composant, nous proposons la traduction de la définition donnée par les participants à la première édition du *Workshop on Component Oriented Programing* [Szyperski et Pfister 1996][4] :

Un composant logiciel est une unité de composition spécifiant, par contrats, ses interfaces (fournies et requises) et ses dépendances explicites aux contextes. Un composant logiciel peut être déployé indépendamment et peut être sujet de composition par un tiers pour la conception d'applications logicielles.

Il en résulte de cette définition que :

- Un composant est une unité de composition spécifiant, par contrat, ses interfaces (fournies et requises) et ses dépendances explicites aux contextes ;
- Un composant logiciel peut être déployé indépendamment (l'installation sur différentes plates-formes) ;
- Un composant est capable de s'auto-décrire, ce qui permet aux constructeurs d'applications de l'utiliser facilement sans qu'ils aient besoin d'en connaître son fonctionnement.

1.2.2 Architecture d'un composant logiciel

L'architecture d'un composant logiciel (cf. figure 1.1) spécifie ses entrées et ses sorties afin de faciliter la description de son comportement (services offerts) quels que soient les langages de programmation utilisés.

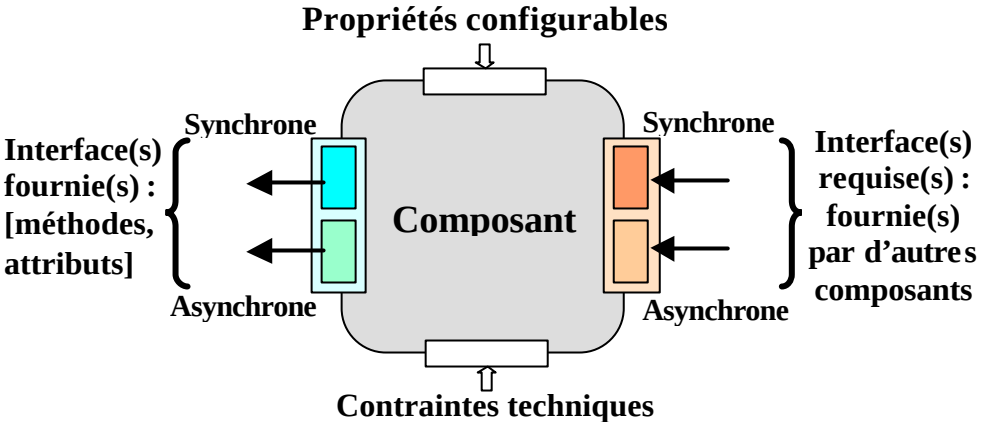

Figure 1.1 : Architecture d'un composant logiciel

[4] La version originale de la définition du composant : *"A Software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A Software component can be deployed independently and is subject to composition by third parties".*

Comme le présente la figure ci-dessus, un composant logiciel possède, principalement, les trois éléments suivants :

- Les interfaces fournies (*sorties*) et les interfaces requises (*entrées*), en mode synchrone ou asynchrone : ce qui définit ses moyens mis en œuvre pour coopérer. Ces moyens peuvent être des opérations (des fonctions promises aux clients) ou des propriétés ;
- Les propriétés configurables : généralement, ce sont des attributs. Elles permettent d'adapter et de personnaliser le composant dans des contextes d'exécutions spécifiques ;
- Les contraintes techniques (QoS : *Quality of Services*), qui peuvent être : la sécurité, la persistance, les transactions, etc.

1.3 Assemblage de composants logiciels

1.3.1 Les langages de description d'architecture : ADL

Dans l'approche "composant", une application logicielle est construite par une collection de composants logiciels interconnectés à l'aide de connecteurs. Pour mettre en évidence la structure d'une application, cet assemblage peut être décrit à l'aide des langages de description d'architectures, ADL (Architecture Description Language) [Sourcé et Duchien 2002].

Les ADL permettent de décrire une architecture logicielle en se basant sur la notion de configuration. 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. Elle permet aussi de décrire son comportement (création, utilisation et suppression des instances).

1.3.2 Les connecteurs logiciels

La notion d'assemblage des composants est définie soit directement pour les composants ayant des interfaces de même type, soit via des liens pour l'adaptation de la connexion des composants. Ces liens sont réifiés[5] par les connecteurs logiciels [Nikunj et al, 2000]. Ces derniers sont proposés principalement pour faciliter l'assemblage des composants en jouant le rôle d'intermédiaires entre les composants non-compatibles.

1.3.2.1 Définition et nécessité des connecteurs

En se basons sur toutes les définitions, proposées dans la littérature, pour les connecteurs nous proposons la définition ci-après qui combine les points essentiels de celles-ci.

Les connecteurs sont des entités architecturales de communication qui modélisent de manière explicite les interactions (transfert de contrôle et de données) entre les composants. Ils contiennent des informations concernant les règles d'interactions entre les composants.

Ainsi, l'objectif des connecteurs est d'atteindre une meilleure réutilisabilité lors de l'assemblage des composants. En effet, la raison de la proposition des connecteurs est de faciliter le développement d'applications à base de composants logiciels. Les composants s'occupent du calcul et de stockage et les connecteurs s'occupent de gérer les interactions (communication / coordination) entre les composants.

Ainsi, nous pouvons dire que les connecteurs sont des logiciels de communication capables d'adapter les besoins associés aux spécifications d'interfaces requises et fournies.

[5] La réification d'un concept consiste à créer les entités nécessaires pour l'implémentation de sa sémantique. L'autre terme utilisé comme équivalent au terme "réifier" est le terme "chosifier".

1.3.2.2 Classification des connecteurs

Les connecteurs peuvent décrire des interactions simples (appel de procédure) d'une manière directe entre des interfaces de même type ou des interactions compliquées en jouant le rôle d'adaptateurs d'interfaces.
Medidovic [Medidovic 2000] a classé les services d'interaction offertes par les connecteurs en quatre types[6]. Chaque type de connecteur offre un ou plusieurs service (s) d'interaction. Les services sont les suivants :

- L**e service de communication.** Un connecteur assure ce service s'il s'occupe des transmissions des données entre composants.
- **Le service de coordination** qui supporte le transfert de contrôle entre composants. Les appels des fonctions est un exemple de cette catégorie de connecteurs.
- **Le service de conversion** convertit les interactions inter-composants si nécessaire. Il permet aux composants hétérogènes d'interagir.
- **Le service de facilitation** négocie et améliore l'interaction entre composants. Ce service est assuré par le connecteur d'équilibrage de charge.

[6] Pour plus de détail voire annexe A.

1.4 Les modèles de composants

A l'heure actuelle, il existe principalement trois modèles de composants [Donsez 2001], encore appelés composants métiers : EJB de Sun MicroSystems, COM+ de Microsoft et CCM de l'OMG[7] [Riveill 2001]. Nous présentons ici les descriptions et les architectures de ces modèles. Pour plus de détail, une étude complète est proposée dans l'annexe B de ce rapport.

1.4.1 Le modèle des composants Enterprise Java Beans, EJB

1.4.1.1 Présentation

Avec le succès remarquable du langage Java, Sun MicroSystems a standardisé, au sein de J2EE[8], la technologie EJB en novembre 1997 [09]. La technologie EJB définit un modèle abstrait de composants Java côté serveur qui décrit la structure des composants EJB et des modèles de développement et de déploiement des applications à base de ces composants [Patzer 2000]. Dans le modèle EJB, les composants sont des composants Java non visuels, portables, distribués, réutilisables et déployables. Le modèle de composants EJB est une extension du composant visuel JavaBeans [13], proposé pour supporter les composants serveurs.

1.4.1.2 Architecture

Un EJB est nécessairement représenté par l'**interface distante** (*Remote interface*) qui définit une vue cliente de l'EJB (cf. figure 1.2). Cette interface définit toutes les méthodes fonctionnelles (business methods) qu'un client peut invoquer sur le composant. Elle hérite de plusieurs interfaces, soit prédéfinies, comme *EJBObject*, soit définies par l'utilisateur [Mougin et Barriolade 2001]. La fourniture d'opérations *métier* est l'objectif de la conception d'un EJB. L'implantation de ces opérations représente l'implantation du composant EJB [RNRT 2000]. En plus de cette interface métier, un EJB offre une autre interface pour accéder aux **méta-données** de l'instance du composant.

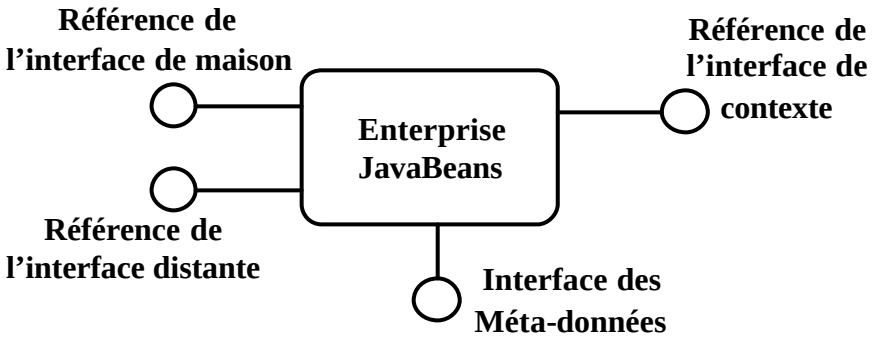

Figure 1.2 : Le modèle de composants EJB

En effet, la spécification 1.1 souligne l'importance de pouvoir modifier la logique métier d'un EJB sans avoir besoin d'accéder au code source de celui-ci. Dans ce sens, l'interface de méta-données permet de fixer le comportement d'un EJB au déploiement.
Un EJB offre une interface qui gère le cycle de vie d'un composant **: interface maison** (*home interface)*. Elle est accessible à travers le JNDI[9]. Cette interface définit les méthodes qu'un client peut invoquer pour créer, rechercher ou détruire une instance de composant [Edwards et Deshpande 1988]. Elle offre également une opération retournant une référence sur l'interface de méta-données.

[7] OMG: Object Management group.

[8] J2EE: Java 2 Enterprise Edition.

[9] JDNI: Java Naming and Directory Interface. Il permet à chaque objet de toute machine virtuelle, à partir d'une chaîne, d'identifier et de localiser tout objet construit et déployé sous forme d'EJB dans toute machine virtuelle [Patzer 2000].

La configuration des composants EJB se fait à l'aide des descripteurs de déploiement, à savoir des fichiers XML décrivant le composant EJB. Ces descripteurs regroupent, en plus des informations sur le fournisseur de composant, une description des besoins de l'EJB en terme de ressources (les propriétés non-fonctionnelles). Les descripteurs de déploiement permettent de modifier les propriétés et le comportement des EJBs sans qu'il soit nécessaire d'effectuer une re-compilation.

1.4.2 Le modèle des composants COM+

1.4.2.1 Présentation

Au début des années 90, le modèle COM (Component Object Model) s'est imposé comme une technologie Windows, importante et évolutive. Par définition, tout objet COM peut être distribué [14]. En d'autre terme, il peut être instancié à partir d'une autre machine via un simple appel réseau. On parle alors de DCOM (Distribueted COM), qui est l'extension de COM prenant en compte l'aspect distribué (il permet la communication entre objets situés sur des machines différentes) [15]. L'architecture Internet de Microsoft, pour développer et supporter les applications réparties, construite autour de DCOM s'appelle DNA (*Distribueted Network Application*) [Fabrice 2001].

Sous Windows 2000, DCOM change de nom pour devenir COM+. On peut dire alors que COM+ est l'évolution de COM. Dernièrement, Microsoft a développé un environnement de création d'application Web nommé « .NET » (prononcer dot-Net) [13].

1.4.2.2 Architecture

Le modèle COM+ inclut une série de fonctionnalités comme le "*data binding*" assurant la liaison entre certains objets et les champs d'une base de données, des fonctionnalités d'événements et des messages asynchrones (*COM+ Event Services* et *Queued Components*), et la gestion de la distribution, etc.

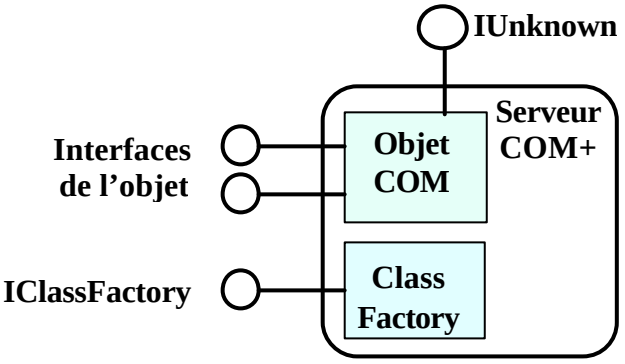

Figure 1 .3 : Le modèle de composants COM+

Un composant COM+ peut avoir un ou plusieurs **Interfaces de l'Objet** qui dérivent de l'interface Iunknown. Les interfaces sont fortement typées. L'interface **IUnknown** est l'interface de base que doit implémenter tout composant COM+. C'est par cette interface que l'on peut instancier un composant COM+ [Bellissard et al 1999]. Elle offre trois méthodes : *QueryInterface* qui permet aux clients de découvrir les interfaces fournies (supportées par le composant COM+), et ainsi, de naviguer entre elles.
La classe *Factory* joue le rôle de la classe fabrique. Elle fournit l'interface **IClassFactory** qui contient deux méthodes : la méthode *CreateInstance* a pour but d'instancier des Objets qu'ils soient distants ou locaux. La méthode *LockServer* permet de mettre l'Objet en mémoire afin de l'instancier au plus vite.

1.4.3 Le modèle des composants CCM

1.4.3.1 Présentation

La spécification CORBA 3.0 a été publiée par l'OMG (Object Management Group) en juillet 2002. Elle a introduit le modèle CCM[10], *CORBA Component Model*. Ce modèle propose toute une structure pour définir un composant CORBA, son comportement, son intégration dans un conteneur (ou application) et son déploiement dans l'environnement distribué CORBA [ACCORD 2002a]. CORBA supporte l'interaction, à travers le Web, entre des composants écrits en différents langages distribués et exécutés sur des ordinateurs avec différents systèmes d'exploitation. Il intègre aussi, des descripteurs pour la configuration, la définition de l'assemblage et le déploiement des composants.

Le modèle CCM est un ensemble de modèles qui permet de spécifier des composants, de les implémenter, de les empaqueter, de les assembler et enfin de les déployer dans un environnement distribué. Le contenu de la spécification est découpé en quatre modèles [ACCORD 2002b] qui sont :

1. **Le modèle abstrait de composants** explique comment décrire un composant en faisant appel à une nouvelle extension du langage IDL de CORBA (Interface Definition Language) [Seinturier 2002] ;

2. **Le modèle d'implantation** 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 déploiement** définit comment le composant sera distribué, assemblé et déployé dans une architecture CCM **;**

4. **Le modèle d'exécution** définit la structure et l'utilisation des conteneurs de composants.

Nous présentons ici que le modèle abstrait de composant (architecture de composants CCM) Les trois autres modèles sont présentés dans l'annexe B de ce rapport.

1.4.3.2 Architecture

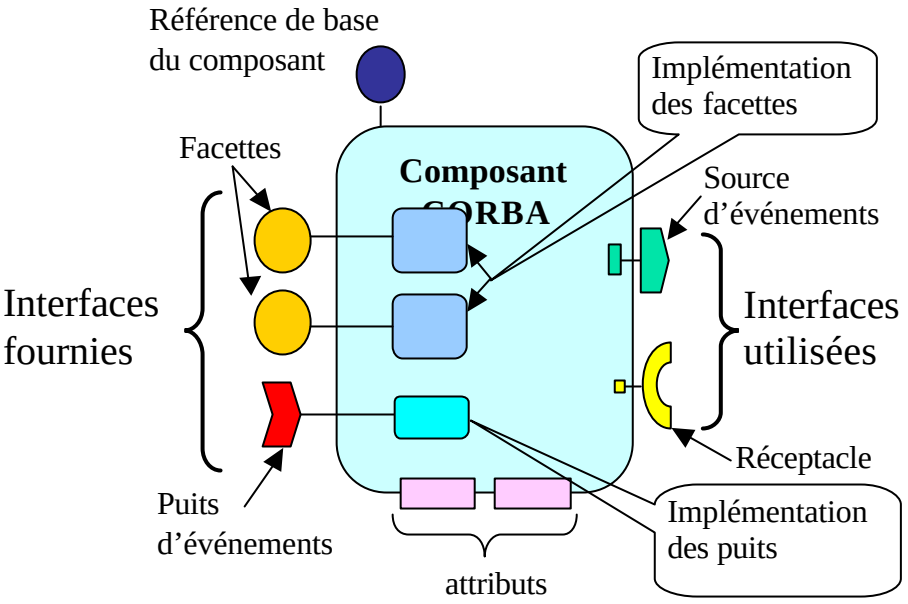

Figure 1.4 : Le modèle abstrait de composants CCM

[10] Nous présentons ici le modèle qui décrit l'architecture du composant CORBA. Les trois autres modèles sont présentés dans l'annexe A de ce rapport.

Le modèle abstrait d'un composant CCM décrit la structure interne d'un composant basé sur la technique de conteneur similaire à celle des EJBs. Il permet aux développeurs de définir les interfaces et les propriétés de composants. Le langage OMG IDL est étendu pour exprimer les interconnections d'un composant [Pérez 2003]. Il est baptisé IDL3 en rapport avec la version 3.0 de la spécification CORBA.
En résumé, un composant CCM est constitué de :

- Une référence de base ;
- Des interfaces : 4 types de *port* (la facette, le réceptacle, la source et le puits d'événements) ;
- Des attributs ;

A ces éléments constituant un composant CCM est associé une Fabrique *"home"* qui constitue un gestionnaire pour les instances de composants.

Comme le présente la figure 1.4, chaque composant CCM à une référence, dite **référence de base** et il peut avoir un ou plusieurs **attributs** qui représentent ses propriétés configurables. En plus, il est doté de multiples interfaces appelées "*ports*". Ces **interfaces** se classent en deux types : fournies et requises. Deux modes d'interfaces "*ports*" fournies (présentées sur le côté gauche du schéma) : facettes *"facets"* pour les invocations synchrones, et puits d'événements "*event sinks"* pour les notifications asynchrones. Les facettes ont le même cycle de vie que celui du composant qui les encapsule. Elles permettent d'avoir des points de vue différents sur un même composant. A partir de la référence de base d'une instance de composant, il est possible de naviguer entre les différentes facettes au cours d'exécution.

Chaque composant CORBA est associé à une **maison** "*home"*. Une maison est un gestionnaire des instances de composants qui permet de créer, à l'exécution, des instances de même type, et il permet également, la recherche d'une instance en basant sur des clés [OMG 2002]. Néanmoins, il est possible de définir différents types de home pour un type unique de composant. Elle est aux composants ce que l'opérateur *new* est aux objets.

Les types des composants sont définis indépendamment de types des maisons. Une définition de maison, cependant, doit indiquer exactement un type composant qu'elle contrôle. Différents types de maison peuvent gérer le même type de composant, bien qu'ils ne puissent pas gérer le même ensemble d'instances de composants [Samaha 2002].

1.5 Positionnement de notre travail

1.5.1 Comparaison et évaluation des modèles de composants existants

Après cette brève étude des trois modèles, le tableau suivant résume les principales caractéristiques de chaque modèle. Ce tableau décrit des points de comparaison sur lesquels nous nous sommes basés pour étudier les trois modèles et les caractéristiques de chacun d'eux.

	Composant / Critère de comparaison	**EJB**	**COM+**	**CCM**
Critères généraux	Architecture	J2EE	.NET	CORBA
	Editeurs de Composant	Plus de 30	Microsoft	OMG (850 membres)
	Interpréteur	JRE (Java Run-time Engine)	CLR (Common language Runtime)	IR (Interfaces Repository)
	Intégration	MV	Par un système de type commun +MV pour ce type de système	Au niveau IDL

Critères spécifiques	Langages de programmation	Mono-langage (Java)	Multi-langage (27 langages)	Multi-langage
	Plate-forme	Multi -plate-forme	Mono-plate-forme	Multi-plate-forme
	Interfaces	Deux interfaces : Home, Object	Multiple-interfaces	Multiple-interfaces
	Connecteur	Non	Non	Oui
	Relation de composition	Non	Non	Non

Ce tableau montre deux types de critères : critères généraux et critères spécifiques. Les critères généraux décrivent les plates-formes associées aux différents modèles de composants. Les critères spécifiques décrivent des propriétés liées directement à notre problématique (connecteur, interface, relation de composition, etc). D'autres critères complémentaires à ceux présentés ainsi qu'une étude comparative complète des différents modèles de composants sont présentés dans l'annexe B de ce rapport.

1.5.2 Pourquoi proposer un modèle de composants logiciels composites?

La réponse à la question : "pourquoi nous intéressons-nous à avoir un modèle de composants-composites dans le contexte des composants logiciels?" est, tout simplement, que les composants composites permettent de représenter des composants complexes par la composition de parties, elles-mêmes sont représentées par des composants.
En d'autres termes, certains composants logiciels sont très complexes et contiennent dans leurs compositions d'autres composants logiciels. Par exemple, un composant logiciel de "comptabilité" peut être composé d'autres composants logiciels "base de données", "base de règle", "interface de saisie", etc. De la même manière un composant logiciel "voiture" peut être composé d'autres composants logiciels tels que "roue", "moteur", "carrosserie", etc. Ainsi la composition (ou les liens de composition) est un concept essentiel et naturel. Ainsi l'introduction de ce type particulier d'assemblage permet à contribuer à compléter l'ensemble des différents types d'assemblage proposés dans les modèles de composants.

1.5.3 Choix de notre modèle support : CCM

Nous avons choisi comme modèle support pour l'implémentation de notre proposition le modèle CCM. Ce choix est motivé par plusieurs raisons. En effet, ce modèle a présenté des succès dans peu de temps. Ce succès est dû au fait qu'il est un modèle **multi-interfaces**, **multi-langage** et **multi-plate-forme.** Il fournit une simplicité de développement par rapport aux autres modèles. En plus, il est **interopérable** avec le modèle EJB. Le modèle CCM permet de concevoir des composants et de les interconnecter au travers de liens de même type.

1.6 Conclusion

Ce chapitre constitue la charpente sur laquelle nous nous basons pour introduire la relation de composition dans les modèles de composants logiciels. Il nous a permis de présenter les différentes notions de base telles que la notion de composant logiciel, les ADL, les connecteurs,…etc. Il nous a permet aussi de présenter les différents modèles de composants logiciels existants et de mettre en évidence leurs similarités, leurs différences, leurs avantages et leurs limites. Ainsi, nous consacrons le prochain chapitre de ce rapport à l'introduction de la notion de composition dans ces modèles.

Chapitre 2 : Proposition : Modèle de Composants-Composites

2.1 Introduction

Nous consacrons ce chapitre à la présentation de notre proposition concernant un modèle de composant-composite. Nous commençons tout d'abord par la présentation de l'étude que nous avons établie sur la relation de composition et de ses propriétés. Après, nous introduisons la relation de composition pour les modèles de composants logiciels en proposant de nouveaux concepts tels que le composant composite et le connecteur de composition. Nous terminons ce chapitre par la présentation du processus de conception des composants logiciels composites.

2.2 Etude de la relation de composition

2.2.1 Définition

Une relation de composition est la relation qui lie des entités avec leur composite par un ensemble de liens de composition. Elle assure une sémantique bien définie qui caractérise l'assemblage des composants de l'entité composite. Elle est représentée par un graphe (cf. figure 2.1), dénommé **graphe de composition**, qui peut dériver à un multi-graphe si au moins l'un des ses composants est une entité composite (c'est la propriété de transitivité, qui sera définie par la suite).

Le graphe de composition est construit de nœuds désignant les entités composites et les entités composants, et d'arcs étiquetés désignant les liens de composition. Les étiquettes représentent les noms de liens de composition.

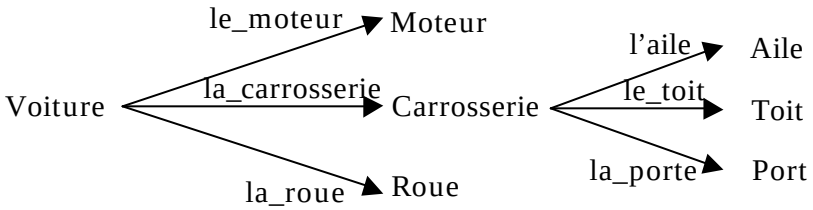

Figure 2.1 : Un graphe de composition

Ainsi, les entités composites sont des entités complexes créées par l'assemblage d'un ensemble d'entités élémentaires à l'aide de la relation de composition. En d'autres termes, une entité composite est formée par l'agrégation de plusieurs entités composantes qui décrivent chacune une partie de l'entité composite.

2.2.2 La relation de composition et ses propriétés

Les caractéristiques de cette relation sont les suivantes :

La transitivité : La relation de composition est transitive lorsque les mêmes types de relation de composition sont impliqués dans les prémisses. Par exemple[11] : L'entité pneu fait partie de l'entité roue et l'entité roue fait partie de l'entité composite voiture, alors l'entité pneu est une partie de l'entité voiture.

L'antisymétrique : Une entité fait partie d'une entité composite, mais l'inverse n'est pas possible. Par exemple : l'entité "moteur" est une partie de l'entité "voiture" alors l'entité composite "voiture" ne peut pas être une partie de l'entité "moteur" ;

[11] Pour expliquer les propriétés de la relation de composition, on fait référence à l'exemple support : "composant composite voiture".

L'existence : Cette propriété bidirectionnelle est définie par les deux propriétés déterminées au niveau des liens de composition : **la dépendance et la prédominance**. La dépendance spécifie une forte relation entité-composite / entité-composant. La destruction de l'entité composite implique immédiatement la destruction des ses composants. La prédominance spécifie une forte relation entité-composant / entité-composite. La destruction d'une entité-composants implique immédiatement la destruction de l'entité-composite ;

La multiplicité : Une entité ne peut être reliée qu'à une et une seule entité composite (cette propriété est vérifiée pour des liens de composition exclusifs). Cette propriété n'est pas vérifiée si les liens de composition sont partagés ;

La propagation : C'est une propriété importante qui permet le partage des valeurs d'attributs entre les entités composites et leurs composants. L'état de l'entité composite est déterminé par l'état de ses composants. Le sens de la propagation peut être celui de l'entité composant vers l'entité composite (par exemple : la couleur de la carrosserie et la même que celle de la voiture) ou bien celui de l'entité composite vers l'entité composant (par exemple : la marque de la voiture est la même que celle de son moteur) ou bidirectionnel. Sachant que la propagation n'est pas toujours évidente dans les deux sens, des conflits peuvent se produire. Si tous les liens sont exclusifs dans le sens composite vers composant, il ne peut y avoir de conflits.

2.3 Introduction de la relation de composition pour les composants logiciels

2.3.1 Définition de composant logiciel composite

Pour pouvoir introduire la relation de composition dans les modèles de composants logiciels afin de proposer par la suite un modèle et un prototype de composant-composite paramétrable (configurable). Nous avons défini de manière précise la notion de composant logiciel composite et ainsi que l'ensemble de propriétés de la relation de composition des composants logiciels.

Un composant-composite est un composant logiciel ayant des liens de composition avec d'autres composants logiciels, dénommés ses composants, qui décrivent chacun une de ses parties. Ces composants peuvent être à leur tour des composants-composites ou des composants simples.

A partir de cette définition nous pouvons établir les deux règles à respecter pour pouvoir introduire la notion de composant composite dans les modèles de composants logiciels :

1. Un composant composite est avant tout un composant logiciel. Il est une boîte noire qui publie des interfaces fournies et requises. Ses composants sont accessibles directement tant qu'ils ne font pas partie de ce composite. A partir du moment où ils deviennent des composants de ce composite, ils seront inaccessibles qu'à travers leur composite.

2. Un composant-composite est lié à ses composants par la relation de composition. Il doit refléter ses propriétés de composition telles que la propagation des interfaces ou la dépendance entre l'existence des composants vis à vis de l'existence de leur composite.

2.3.2 Réification de la relation de composition pour les composants logiciels

Pour définir notre modèle de composant composite qui est basé sur la relation de composition, nous avons étudié plusieurs choix pour l'introduction de cette relation dans les modèles de composants logiciels. Ces choix sont :

1. La sémantique de la relation de composition n'est pas réifiée, mais assurée par le système ;
2. La sémantique de la relation de composition est réifiée dans le composite ;
3. La sémantique de la relation de composition est réifiée dans les composants de composite ;
4. La sémantique de la relation de composition est réifiée dans les connecteurs de composition.

Dans le cas où la sémantique de la relation de composition serait directement intégrée dans les couches système, le modèle de composant-composite devient peu flexible et peu configurable. Aussi, le choix de réifier la sémantique de la relation de composition dans le composant-composite ou dans ses composants est en contradiction avec la définition d'un composant logiciel qui doit contenir que du code métier pour assurer des fonctionnalités précises (le calcul, le stockage,…etc.) et il doit se déchargé de toutes les fonctions concernant les interactions et la communication. Ces fonctions sont réservées pour les connecteurs logiciels.

Par ailleurs, les types de connecteurs existant à l'heure actuelle ne peuvent pas jouer le rôle d'un connecteur de composition, car ce dernier, en plus du rôle d'interconnexion des composants réifie les propriétés de la relation de composition. En se basant su ces constats, nous proposons un nouveau type de connecteur que nous baptisons **connecteur de composition**.

Parmi toutes les approches proposées par la création des connecteurs [Traverson et Yahiaoui 2002], nous avons opté pour celle basée sur la génération automatique des connecteurs[12].

En effet, l'approche de la génération automatique des connecteurs présente plusieurs avantages. Parmi celles-ci nous citons :

1. C'est une stratégie simple. En effet, les propriétés, reflétant la sémantique de la relation de composition, peuvent être identifiées, définies et générées facilement ;
2. Elle ne nécessite pas l'intervention du développeur, car la génération est faite automatiquement par un outil de génération à l'opposé des autres approches.

2.3.3 Les connecteurs de composition

Dans notre modèle, *un connecteur de composition est un composant logiciel. Il possède ses interfaces et ses propriétés. Il s'interpose entre des composants dits composites et d'autres composants logiciels pour réifier la sémantique de la relation de composition.*

Par conséquent, les connecteurs de composition assurent les propriétés de la relation de composition :

L'exclusivité/partage : Pour assurer cette propriété, le connecteur de composition qui lie le composite avec un type de composant doit stocké un attribut booléen. La valeur *true* désigne que le lien est exclusif, et *false* est partagé. Ainsi on peut l'assurer par la limitation de nombre d'interfaces requises du composite et de ses composants. ;

[12] Les approches de création des connecteurs sont : les connecteurs génériques, les connecteurs disponibles comme patterns ou dans des bibliothèques spécifiques et la génération automatique de connecteurs.

La dépendance/indépendance : Pour assurer la relation de dépendance entre un composant-composite et un de ses composants, une variable booléenne sera stockée dans le connecteur correspondant. Cette variable doit être testée pour toutes les opérations concernant, par exemple, la destruction du composant-composite ou du composant ;

La prédominance/non-prédominance : Pour assurer cette propriété, nous devrons vérifier une variable booléenne stockée dans le connecteur lors de destruction de tout composant sa composite. Si elle est *false* on peut détruire le composant sans toucher le composite. Si elle est *true*, l'existence du composite est liée à celle de ses composants. Dans ce cas, si la variable de dépendance est *true* nous devrons détruire le composite et tous ses composants ;

La cardinalité : La cardinalité est représentée par une variable représentant un nombre (n>=0) ou un intervalle [m n].

2.3.4 Processus de conception de composants composites

Pour pouvoir définir et utiliser un composant composite il est nécessaire d'identifier ses propriétés structurelles (ses liens de composition, les propriétés de ces lies, les interfaces requises pour chaque lien, etc) ainsi que de définir l'ensemble des opérations concernant sa manipulation.

1. **Description de la structure du composant-composite :**
   La description d'un composant consiste à définir ses caractéristiques. Cette description contient la définition du nom de composant, ses attribues, ses liens de composition ainsi que les interfaces nécessaires pour l'établissement de chaque lien.

2. **Utilisation des composants-composites :**
   L'utilisation de composant-composite nécessite la définition de l'ensemble des opérations permettant d'assurer les services associés à la relation de composition. Par conséquent, il est nécessaire de vérifier la cohérence de la sémantique de la relation de composition lors de l'utilisation de chaque opération. Par exemple, il est nécessaire de pouvoir ajouter ou de supprimer un composant (sous-composite) à un composite de manière simple, mais aussi, il faut s'assurer que la valeur de la cardinalité est vérifiée et que la propriété de l'antisymétricité l'est aussi.

Les opérations de manipulation de composant composite sont la création du composite, l'ajout d'un nouveau composant à un composite existant, la suppression d'un composant faisant partie d'un composite existant et l'invocation de l'un des interfaces du composite. Nous soulignons que les interfaces fournies par le composite incluent certaines interfaces définies par ses composants. L'utilisation de ces derniers est spécifiée lors de la définition des liens de composition. Elle fait appel à la propriété de propagation d'interface.

2.4 Conclusion

Dans ce chapitre nous avons présenté notre proposition concernant les composants logiciels composite. Nous avons commencé ce chapitre par l'étude de la relation de composition avant d'introduire cette relation dans les modèles de composants logiciels. Ainsi, nous avons défini la notion de composant logiciel composite et des connecteurs de composition réifiant les propriétés de la relation de composition. Enfin, nous avons présenté le processus de la réalisation des composants-composites.
Dans le chapitre suivant, nous présentons la projection du modèle étudié ici sur le modèle de composants CCM.

Chapitre 3 : Mise en Œuvre des composants-composites en CCM

3.1 Introduction

Dans le chapitre précédent, nous avons introduit la relation de composition pour les modèles de composants logiciels en général. Dans ce chapitre, nous allons mettre en œuvre ce modèle pour les composants CCM.

3.2 Le processus de conception de composants-composites en CCM

Pour pouvoir définir et utiliser des composants composites dans le modèle CCM, nous avons procédé comme suite :

1. La description du composant composite ou de son utilisation en IDL3+ ;
2. La projection de cette description ou de cette utilisation vers IDL3 par le compilateur **IDL3+ToIDL3** ;
3. L'interconnexion entre les composants, les connecteurs et le composite.

3.2.1 La description des composants composites : le langage IDL3+

La description d'un type de composant-composite nécessite l'introduction de nouveaux concepts. Ces derniers sont ajoutés à ceux définis par le langage IDL3. La description d'un composite, à l'aide de IDL3+ définit principalement les éléments suivants : le module de la classe composite, ses composants, ses liens de composition et leurs propriétés. La déclaration du composant-composite se fait à l'aide du nouveau mot-clé **composite**.

```
composite nom_Composite {..};
```

La déclaration de composants de composite se fait par le nouveau mot-clé **composeOf**.

```
composeOf nom_composant {..};
```

La description d'un composant-composite consiste à décrire le composant-composite lui-même et les propriétés de ses liens de connexion avec ses composants (les connecteurs de composition) et ses composants.
La déclaration en IDL3+ des composants de composite se fait tout simplement comme celle en IDL3. L'exemple suivant montre la description d'un composite et l'un de ses composants, ainsi les propriétés du lien qui le lie avec ce composant. L'exemple suivant présente la description d'un composant composite voiture.

```
//=================================================================
//   Le fichier de déclaration du composant-composite Car en IDL3+
//=================================================================
module demoCar {
//déclaration de l'interface WheelInterface
    interface WheelInterface {
        void WheelMove() ;
   };

//déclaration du composant composite
    composite Car {
       attribute string the_name;
```

```
//déclaration de l'interface du composant composite
       type : synchrone ;
       port_in WheelInterface ;

       composeOf Whell {
      attribute string the_name;
//déclaration des propriétés de la relation de composition
      exclusive : true;
      dependance : true;
      prevalence : true;
      cardinality : 4;
       };
//déclaration de la maison "home" du composant-composite
    home WheelHome manages Wheel{
    };
//=================================================================
//                  La déclaration du composant Wheel
//=================================================================
    component Wheel {
        attribute string the_name;

        provides WheelInterface  for_carW ;
    };
//déclaration de la maison "home" du composant Wheel
    home WheelHome manages Wheel {
    };
};
```

L'ensemble des définitions que nous avons introduits au langage IDL3 sont résumées dans le tableau ci-dessous.

Concept	La description du concept
composite	Pour spécifier un type de composant-composite
composeOf	Pour définir l'ensemble des liens de composition des composants de composite.
dependance	Prend deux valeurs : True : il existe une dépendance entre le composite et le composant false : (indépendance) il n'y a pas de dépendance entre le composite et le composant.
exclusive	Prend deux valeurs : True : Indique si le lien est exclusif (Généralement est true) false : Indique si le lien est partégé
prevalence	Cette propriété prend deux valeurs True : il y a une prédominance entre le composite et le composant false : (non-prédominance) dans le cas où n'y a pas de prédominance entre le composite et le composant.
cardinality	Définit le nombre d'instances "n" d'un composant (n>=1).
type	Prend les deux valeurs : synchrone et asynchrone.
port_in	Représente le type de l'interface requise (un réceptacle ou un puits d'événements)
port_out	Représente le type de l'interface fournie (une facette ou source d'événements)

3.2.2 La projection de la description d'un composant-composite de IDL3+ vers IDL3

La projection de la description d'un composant-composite, de IDL3+ en IDL3 par le compilateur **IDL3+ToIDL3** (cf. figure 3.1), suit des règles bien définies. Toutefois la règle principale est qu'un type de composant-composite est transformé en un ensemble de composants représentant le composite, ses composants et les connecteurs de composition.

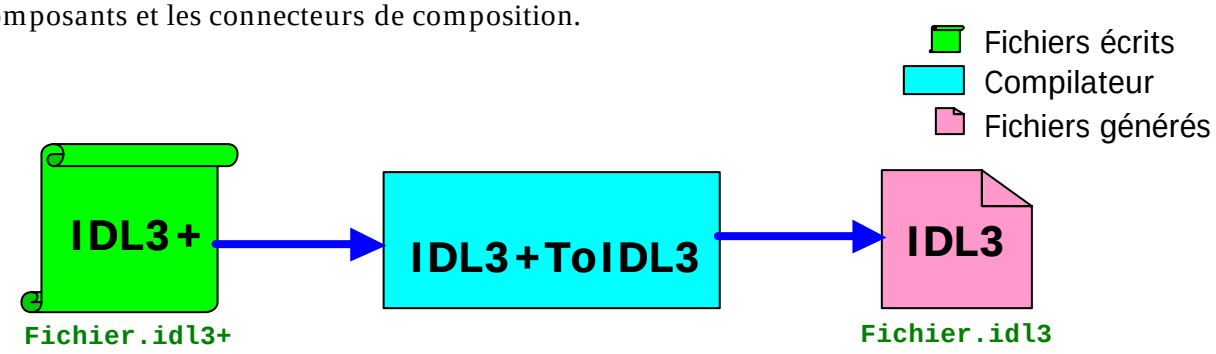

Figure 3.1 : Processus de projection des fichiers IDL3+ vers IDL3

La projection de la description d'un composant-composite de IDL3+ vers IDL3 produit :

1. La définition d'un composant qui prend le nom : " nom_compositeComposite" avec sa maison "*home*" qui possède le nom "nom_compositeCompositeHome";
2. Pour chaque description d'un lien, il définit un composant qui dispose du nom du composant suffixé par *Connector*. Le même suffixe sera ajouté à sa maison ;
3. La description des composants de composite reste elle-même en gardant aussi, la déclaration de la maison ;
4. En fonction de la valeur de type : si synchrone, il crée soit une facette (si port_in : true) pour le composant et un réceptacle pour le composite. Si asynchrone, il crée une source d'événements pour le composant et un puits d'événement pour le composite ;
5. La déclaration des attributs reste telle qu'elle est ;
6. S'il s'agit d'une interface, son nom sera la combinaison du nom d'interface et ToComposite (nom_interfaceToComposite) ;
7. Dans le cas d'un événement, le nom de l'interface sera suffixé par ToComposite (nom_événementToComposite) ;
8. Les ports d'entrée / sortie gardent le même nombre mais change la façon de déclaration.

Pour illustrer le résultat de la projection de IDL3+ vers IDL3, nous présentons la projection de l'exemple donné dans la section précédente (exemple d'un composant-composite voiture construite d'une roue) dans le tableau suivant[13] :

```
module demoCar {
//déclaration de l'interface WheelInterface
    interface WheelInterface {
        void WheelMove() ;
    };
//=====================================================================
```

[13] Dû à la limitation de nombre de pages, nous ne présentons pas ici l'exemple complet. Il sera présenté dans l'annexe B de ce rapport.

```
//    La déclaration du composant CarComposite (le composant-composite)
//=================================================================
    component CarComposite {
        attribute string the_name;
        uses WheelInterface  to_wheel ;
    };
//déclaration de la maison "home" du composant composant-composite
    home CarCompositeHome manages CarComposite {
    };
//=================================================================
//    La déclaration du composant Wheel
//=================================================================
    component Wheel {
        attribute string the_name;
//déclaration de l'interface du composant composite
        provides WheelInterface  for_carW ;
    };
//déclaration de la maison "home" du composant Wheel
    home WheelHome manages Wheel {
    };
//=================================================================
//       La déclaration du composant WheelConnector
//=================================================================
    component WheelConnector {
      attribute string the_name
//déclaration des propriétés de la relation de composition comme des attribues
      attribute boolean exclusive;
      attribute boolean dependance;
      attribute boolean prevalence;
      attribute int cardinality;

      provides WheelInterface for_carWC ;
        uses WheelInterface to_wheelC ;
  };
//déclaration de la maison "home" du composant WheelConnector (le connecteur)
    home WheelConnectorHome manages WheelConnector {
  };
};
```

3.2.3 L'interconnexion des composants

Après avoir implémenter le code métier du composant composite, ses composants et ses connecteurs, nous procédons à l'interconnexion de ces composants. Cette opération est automatisée dans notre modèle. En effet, un fichier java contenant l'ensemble de connexions entre connecteurs et composants est automatiquement généré. Nous avons implémenté un outil dédié à cette tâche : l'outil **Compositer.**

Notons que dans le modèle CCM, les connexions sont établies manuellement par le concepteur d'application logicielle. Les fichiers définissant la connexion sont des fichiers XML ou des fichiers Java.

3.3 Implémentation et prototype réalisé

Nous avons choisi, pour l'implémentation de notre modèle, la plate forme OpenCCM du LIFL. Cette plate-forme est la première implémentation Open Source du modèle CCM [06]. Elle est le résultat d'un projet initié depuis l'année 2000 et dirigé par le laboratoire LIFL. Elle fonctionne sur plusieurs ORB (compatible CORBA 2.4) et sur plusieurs systèmes d'exploitation. Notre choix de cette plate-forme est dû au fait qu'elle est "Open source" en plus d'être interopérable contrairement aux autres plates-formes.

Nous signalons qu'actuellement, la plate-forme OpenCCM n'implante pas l'intégralité de la spécification de CCM, et son implantation n'est pas encore achevée. C'est la raison pour laquelle nous avons eu certaines difficultés liées à la non-stabilité et à l'absence de documentation concernant cette plate-forme, sauf celle disponible pour linux. Or, que nous avons choisi l'environnement Windows et l'OpenORB pour l'implémentation de notre prototype. L'exemple de l'interface concernant l'implémentation du composant composite voiture est présentée sur la figure ci-dessous.

Figure 3.2 : l'exécution de l'implémentation du composant-composite voiture

3.4 Conclusion

Dans ce chapitre nous avons présenté la mise en ouvre de conception d'un composant-composite en CCM. Nous avons expliqué les étapes à suivre pour concevoir un composant composite, en commençant de la description jusqu'au lancement d'exécution sur la plate forme OpenCCM. Ainsi, il est à noter que l'interconnexion entre composant-composite avec ses composants, selon notre modèle, se fait automatiquement.

Conclusion et Perspectives

1. Bilan et apports du stage

Ce rapport est le résultat du travail réalisé au cours de mon stage effectué au sein de l'équipe CSL du département Génie Informatique et Productique de l'Ecole Nationale Supérieure des Mines de Douai. Il reflète trois types d'apport acquis pendant ce stage :

Apport par rapport à l'état de l'art

J'ai réalisé un état de l'art des modèles de composants logiciels de manière générale (leurs concepts et leurs principes) et des trois principaux modèles de composants (EJB, COM+ et CCM) en particulier. En plus de l'état de l'art de ces modèles, une comparaison entre leurs différentes caractéristiques est présentée.

Apport par rapport à la proposition d'une nouvelle technique d'assemblage de composants logiciels

Cette technique concerne l'introduction des composants-composites pour les modèles de composants logiciels. Cette proposition est basée sur une étude de la relation de composition en général et des connecteurs logiciels en particulier. Elle a été implémentée en dessus du modèle CCM. Cette proposition va faire le sujet d'une publication qui va être soumise à une conférence internationale au mois de septembre 2003. La publication est en cours de rédaction avec comme auteur mon encadreur de stage et moi-même. Cela présente pour moi une expérience dans le domaine de recherche scientifique.

Apport par rapport à la maîtrise d'une plate-forme de développement d'applications logiciels à base de composants

En effet, le travail de stage a été implémenté et validé en utilisant la plate-forme OpenCCM du LIFL, qui est une plate-forme complexe. Une formation spécifique au laboratoire GRID de l'INRISA à Rennes m'a été nécessaire pour maîtriser ce domaine.

2. Perspectives

L'étude des modèles de composants logiciels est un sujet vaste, et vu la durée limitée du stage, nous n'avons pas pu aborder tous les aspects liés à notre proposition de composant-composite. Plusieurs pistes des futurs recherches peuvent être envisagées :

1. Etudier le problème lié à la propagation des interfaces de composants vers le composite et vis versa.

2. Il serait intéressant de développer une nouvelle plate-forme supportant un langage de description qui intègre toutes les versions existantes du langage IDL (IDL3+, IDL4, etc) et qui prend compte de tous les types de connecteurs proposés jusqu'à l'heure actuelle (les connecteurs d'adaptation, de partage de charge et de composition). Ainsi, il est utile de combiner le modèle de connecteur de composition proposé avec les connecteurs d'adaptation et de partage de charge afin de faciliter la création de composants-composites dans le cas où ces derniers n'auraient pas les mêmes types d'interfaces que leurs composants.

3. Nous avons pu remarquer, à travers quelques exemples implémentés, la complexité du codage des descripteurs utilisés pour l'assemblage, le paquetage et le déploiement. Nous pensons que cela est plus compliqué lors du développement des grandes applications. Une automatisation de cette tâche est une voie envisageable.

Bibliographie

[ACCORD 2002a] Projet ACCORD, Assemblage des composants par contrats, état de l'art et de la standardisation, Livrable-1.1-1, Juin 2002.

[ACCORD 2002b] Projet ACCORD, *Le modèle de composants CORBA*, Livrable-1.1-4, 31, Mai 2002.

[ACCORD 2003] Projet ACCORD, Modèle abstrait d'assemblage de composants par contrats, Livrable-4, Juin 2003.

[Donsez 2001] D. Donsez, *Transactions et Composants d'entreprise*, Cours IMAG / LSR / ADELE – Université Joseph Fourier (Grenoble 1), 2001.

[Bellissard et al 1999] L. Bellissard, Noël de Palma, M. Riveill, *"COM+ des concepts à la pratiques"*, Cours – SIRAC, INRIA Rhône Aples, 1999.

[Belloir et al., 2001] Belloir N., Bruel J.M., Barbier F., *"Formalisation de la relation tout-paire : application à l'assemblage des composants logiciels"*, journée composants : flexibilité du système au langage, 2001.

[Edwards et Deshpande 1988] W. Edwards, S. Deshpande, *"Intégration de CORBA et d'EJB"*, Livre Blanc, page 1-14, Janvier 2000.

[Fabrice 2001] J-F. Fabrice, *Architectures distribuées et serveurs d'application*, Présentation technique – Calidis Group, 2001.

[Flissi et Merle 2002] A. Flissi, P. Merle, Getting Started With OpenCCM – Building a CORBA Components Application, version 1.0, Novembre 2002.

[Mougin et Barriolade 2001] P. Mougin, C. Barriolade, *"Services Web et Modèles de composants Métier"*, Orchestra Networks - Livre Blanc, Octobre 2001.

[Medidovic et al, 2000] Medidovic N., Taylor R.N., *A Classification and Comparaison Framework for software Architecture Description Languages*, IEEE Transactions on Software Engineering, vol. 26, no. 1, Janvier 2000.

[Nikunj et al, 2000] Nikunj R.M., Medidovic N., Phadke S., *Towards a Taxonomy of Software Connectors*, In Proceeding of the 22nd International Conference on Software Engineering, Limerick, Ireland, Juin 2000.

[OMG 2002] OMG. CORBA 3.0, *CORBA Component Chapters*, Object Management Groupe, Juillet 2002.

[Oussalah et al,. 1997] Oussalah C., Ingénierie objet - Concepts et techniques, pp 63-91, 1997.

[Patzer 2000] A. Patzer, "*Programmation Java côté serveur : Servlets, JSP et EJB*", Wrox Prss et Éditions Eyrolles, édition Française – ISBN 2-212-09109-5,pages 555-600, 2002. www.wroxfrance.com

[Pérez 2003] C. Pérez, *Introduction aux composants CORBA*, Une journée de formation – INSA-Rennes, 27 Avril 2003.

[RNRT 2000] Projet RNRT, Etat de l'art des modèles de composants, N° 98, Mai 2002.

[Riveill 2001] M. Reiveill, *Recherches et Développements autour des middlewares*, Projet RAINBOW - Laboratoire I3S (CNRS - UNSA), 2001.

[Samaha 2002] A. Samaha, *CORBA Component Model (CCM)*, Cours-Université Joseph Fourier (UJF / ISTG / RICM3), 2001-2002.

[Seinturier 2002] L. Seinturier, *Middleware/CORBA Component Model (CCM)*, Cours–Université Pierre & Marie Curie, Juillet 2002.

[Sourcé et Duchien 2002] Sourcé J-M., Duchien L., "Etat de l'art sur les langages de description d'architecture (ADLs), Livrable du projet RNTL ACCORD, 2002.

[Szyperski et Pfister 1996] C. Szyperski, C. Pfister, *WCOP'96 Workshop Report*, Workshop Reader ECOOP'96, Juin 1996.

[Traverson et Yahiaoui 2002] Traverson B., Yahiaoui N., "*Connector for CORBA Components"*, In 8th International Confernce on Object-Oriented Information Systems, September 2002.

[Winston et al., 1987] Winston M.E., Chaffin R., Hermann D.J., A Taxinomie of Part-Whole Relations, Cognitive Science 11, pp 417-444, 1987.

Webgraphie

Les composants CORBA, CCM

[01] http://corba.org
[02] http://www.omg.org/
[03] http://objectweb.org
[04] http://etna.int-evry.fr/cours/middleware/enseignement/polys/polyS.pdf
[05] http://www.ddj.com & http://www.paul-harmon.com

La plate-forme OpenCCM

[06] http://objectweb.org/OpenCCM
[07] http://corbaweb.lifl.fr/OpenCCM/CCM.html
[08] http://ditec.um.es/~dsevilla/ccm

Les Enterprise Java Beans, EJB

[09] http://java.sun.com/
[10] http://java.sun.com/j2se/1.3/docs/guide/rmi/
[11] http://java.sun.com/j2se/1.3/docs/
[12] http://www.ashita-studio.com/tutoriaux/ejb/chapitre_03.php
[13] http://www.ejbhome.com
[14] http://java.sun.com/products/ejb/docs.html

COM/DCOM et COM+

[15] http://www.microsoft.com/
[16] http://www.microsoft.com/com/default.asp
[17] http://www.microsoft.com/com/tech/DCOM.asp
[18] http://www.dotnet-fr.org/intro_dotnet_tmr.php3
[19] http://www.infres.enst.fr/projets/accord/lot1/

Concepts de base

Dans cette partie, nous définissons les principaux concepts nécessaires pour la bonne compréhension de la technologie des composants logiciels.

Concept	Description
Interface	Une interface est une collection nommée d'opérations abstraites (ou de méthodes) qui représente une fonctionnalité.
Classe	C'est une implémentation concrète nommée d'une ou plusieurs interfaces (ou objets).
Objet	Résultat de l'instanciation d'une classe.
Serveur	Un Serveur regroupant un ensemble de services permettant aux différentes parties d'une application répartie d'entrer en interaction.
Client	Un processus qui peut invoquer une méthode d'un Serveur.
Conteneur	C'est avant tout un composant. Il représente le contexte d'exécution.
Connecteur	Un mécanisme mis en œuvre pour interconnecter les composants.
Composant	Un composant est une entité logicielle qui exhibent ce qu'elle sait faire par le biais de son interface.
Composition	Assemblage dont le résultat est un composant. Action de composer un tout par l'agencement de plusieurs parties.
Composite	Composant résultat d'un assemblage de composants plus élémentaires.
Le type d'un composant	C'est la définition abstraite d'une entité logicielle. Il est caractérisé par trois éléments : ses interfaces, les modes avec les autres types de composants et ses propriétés configurables.
Instance d'un composant	Elle est, au même titre qu'une instance d'objet, une entité existante et s'exécutant dans le système.
Paquetage d'un composant	C'est une entité diffusable et déployable, bien souvent une archive, contenant le type de composant, une implantation de ce type et une description du contenu du paquetage : contraintes techniques pour l'implantation, par exemple.
C#	Langage vedette supporté par la plate-forme .NET. C# est très proche du langage Java. Il supporte l'héritage simple, l'implémentation multiple d'interface, les propriétés, la surcharge d'opérateur, les événements, …etc.
DCOM	DCOM est une extension de COM qui permet la communication entre objets situés sur des machines différentes. C'est l'équivalent au niveau de la couche transport du protocole IIOP de CORBA mais pour la plate-forme Windows.
MSIL	Microsoft Intermediate Language. C'est un langage intermédiaire. Tous les langages supportés par la plate-forme .NET peuvent être compilé en code intermédiaire MSIL.
Portabilité du code	La possibilité de recompiler et d'exécuter des applications préfabriquées avec d'autre nouveau produit sans les modifier.

Glossaire

Mot-clé	Description
COM	Component Object Model.
DCOM	Distributed Component Object Model ou bien Distributed COM.
CBSE	Component-Based Software Engeneering, elle est appelée aussi, CBSD : Component-Based Software Development. Elle définit la technique d'assemblage des composants logiciels.
POA	Portable Object Adapter. Il offre toutes les fonctions requises pour implémenter les stratégies requises en terme de transactions et de cycle de vie des composants.
MTS	Microsoft Transaction Server.
JNDI	Java Naming and Directory Interface. Il permet à chaque objet de toute machine virtuelle, à partir d'une chaîne, d'identifier et de localiser tout objet construit et déployé sous forme d'EJB dans toute machine virtuelle. http://java.sun.com/products/jndi
JOnAS	JOnAS c'est une implementation Open source de J2EE.
J2EE	Java 2 Enterprise Edition. J2EE est une plate-forme pour les solutions entreprises qui définit le standard de développement des applications entreprises multi-tiers.
ORB	Object Request Broker. Le bus d'objets répartis de l'architecture globale OMG. Il assure le transport des requêtes entre les objets CORBA.
JMS	Java Messaging Service. http://java.sun.com/products/jms
RMI	Remote Method Invocation API. Elle crée les interfaces distantes pour des traitement distribués (distributed computing) sur une plate-forme Java.
SOAP	Simple Object Access Procotol. SOAP est un protocole d'invocation des services Web.
UDDI	Universel Description, Discovery, and Integration. C'est un organisme central pour enregistrer, trouver et utiliser les services Web.
WSDL	Web Service Description Language. C'est un format de description des services Web.
XML	Extensible Markup Language. XML c'est un format utilisé pour décrire et échanger les données.
IDL	Interface Description Language.
IDL3	Interface Description Language3. C'est une extension étendue du langage de spécification IDL. On l'appel aussi, IDL étendu.
Java IDL	The Java Interface Definition Language API creates remote interfaces to support CORBA communication in the Java platform. Java IDL includes an IDL compiler and a lightweight, replaceable ORB that supports IIOP.

Annexe A : Les types de connecteurs logiciels

Les huit types de connecteurs classés par Medidovic [Medidovic 2000] sont les suivants :

1. **Les connecteurs d'appel de procédures** qui modélisent le flux de contrôle entre les composants au travers des méthodes d'invocation ;
2. **Les connecteurs d'événements** qui modélisent le flux de contrôle propagé par événements entre les composants ;
3. **Les connecteurs d'accès aux données** qui offrent la possibilité d'accéder aux données stockées par des composants de stockage ;
4. **Les connecteurs de liaison**, comme l'indique leur nom, lient un ensemble de composants et les maintiennent en un état précis durant leur exécution.
5. **Les connecteurs de flux** interviennent pour transférer une grande quantité de données entre les composants. Le principal domaine de leur utilisation est le système client-serveur.
6. **Les connecteurs d'arbitrage** qui fournissent des services d'équilibrage de charge et d'ordonnancement.
7. **Les connecteurs d'adaptation** qui offrent des facilités pour supporter l'interaction entre les composants non-intéroparables.
8. **Les connecteurs de distribution** qui en plus d'identifier le chemin de l'interaction conduisent l'information de communication / coordination entre composants.

Annexe B : étude comparative des EJB, COM+ et CCM

Dans cette annexe, nous allons établir un état de l'art sur les principaux modèles de composants logiciels côté serveur, à savoir le modèle EJB, CCM et COM+. Pour cela, nous allons présenter l'environnement de chaque modèle, ensuite son évaluation. Nous terminerons cette annexe par une conclusion.

1. Les composants Enterprise Java Beans, EJB

1.1 Introduction

Avec le succès remarquable du langage Java, Sun MicroSystems a standardisé, au sein de J2EE[14] (Java 2 Enterprise Edition), la technologie EJB (Entreprise Java Beans) en novembre 1997. La technologie EJB définit un modèle abstrait de composants Java côté serveur : EJB, qui décrit la structure des composants EJB et des modèles de développement et de déploiement des applications à base de ces composants.

En 2001, une nouvelle spécification "2.0" a été publiée. Elle a introduit des considérables évolutions de la technologie de développement des composants en simplifiant les solutions métiers [Patzer 2000]. Le but de cette spécification EJB est double :

1. Simplifier le développement en focalisant sur le fonctionnel et en offrant une spécification des services techniques que tout conteneur de composant EJB doit implémenter ;
2. Garantir une indépendance du fournisseur de produit (le serveur compatible EJB) par le biais d'une spécification. Tout serveur compatible EJB garantit l'intégration de tout composant codé selon la spécification EJB sans aucun ajout de code (l'intégration peut se faire à l'exécution s'il le faut).

1.2 Environnement d'exécution EJB

Les composants EJB s'exécutent dans un Conteneur EJB, ils peuvent être assemblés pour construire des applications distribuées.

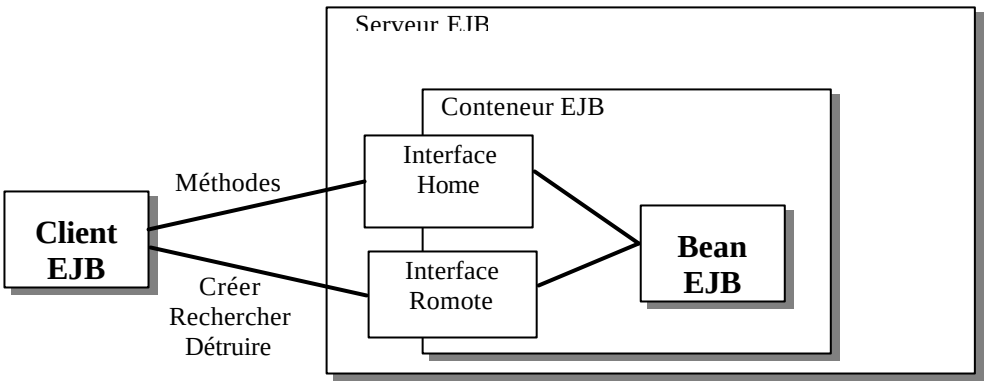

Figure B.1 : L'environnement d'exécution des composants EJB

[14] J2EE (Java 2 Enterprise Edition) : La plate-forme J2EE simplifie le développement des applications logicielles en se basant sur des composants standardisés et modulaires qui bénéficient d'un ensemble complet de services.

La figure B.1 présente l'architecture de l'environnement d'exécution des EJBs. Dans cette architecture EJB, nous distinguons les *Clients distants*, les *Serveurs*, les *Conteneurs* et les *EJBs*. Les Clients distants, appelés **Clients EJB,** peuvent appeler les méthodes définit dans l'interface distante. Les **Serveurs EJB** sont mis en œuvre pour héberger les Conteneurs EJB. Un **Conteneur EJB** assurant les services requis par un (ou plusieurs) **Bean EJB.** L'EJB présente à ses Clients deux interfaces : une **interface de maison** et une **interface distante** qui offre l'accès à ses traitements.

Le Client EJB fournit la logique de l'interface utilisateur sur la machine cliente. Il passe les appels aux composants EJB distants hébergés sur un Serveur. En revanche, un Client ne communique pas directement avec une instance d'EJB. Sa mission est de savoir comment trouver le Serveur EJB et d'interagir avec les composants EJB via les Conteneurs hébergés par le Serveur.

Le Conteneur (structure d'accueil) est l'environnement dans lequel un ou plusieurs composants EJB s'exécutent, il fournit les objets *proxy* mettant en œuvre les **Interfaces Home** et **Remote** des composants. Il a pour responsabilité (en collaboration avec le Serveur) de fournir aux EJBs des services non-fonctionnels tels que : la gestion des transactions, des autorisations, de la persistance, etc. Le développeur n'ayant plus qu'à se concentrer sur le développement de ses EJBs.

1.3 Les différents types d'Enterprise Java Beans

Les EJBs se classent, dans la spécification EJB 2.0, en trois types "catégories", chacun dans une optique différente. En fonction du comportement requis du composant, certaines caractéristiques déterminent lequel parmi ces trois types doit être utilisé. Les types sont :

1- Enterprise Java Beans de type "Session" :

Il exécute une tâche pour le client, c'est-à-dire : une instance *Session* est crée à chaque connexion d'un client. Lorsque le client termine sa session, l'instance du composant est détruite. Un EJB de type session peut être sans ou avec état :
Sans état (*Stateless Session Beans***) :** Un composant de session sans état n'a aucune connaissance du client ou du contexte concernant la requête. L'exemple typique est un convertisseur Euro/Franc qui aurait une seule méthode : `eurotofranc (double valeur)`.
Avec état (*Stateful Session Beans***) :** Un composant de session avec état est dédié à un certain client pendant toute la durée de son instanciation. L'exemple parfait est un EJB de panier d'achat pour le commerce électronique. L'utilisateur effectue les tâches standards en ajoutant des produits au panier, en saisissant son adresse et en émettant la commande. Le bean du panier conserve l'état et, par conséquent, la connaissance de toutes ces variables et les associant à un client unique.

2- Enterprise Java Beans de type "Entity" :

Il représente un objet métier qui reste après la fin d'une session. Une instance *Entity* dispose d'un identifiant unique qui permet au client de retrouver une instance particulière. Un objet représentant un utilisateur (nom, coordonnées, choix de la langue) est l'EJB *Entity* typique.

3- Enterprise Java Beans de type "Message-Driven" :

Il permet le traitement des messages de manière asynchrones au contraire des deux types précédents qui offrent des services aux clients EJBs de manière synchrone. En utilisant JMS (Java Messaging Service), un client produit un message et le publie dans une file d'attente de messages. Un EJB "Message-Driven" extrait, ensuite, le message et exécute son contenu.

1.4 Démarche de conception d'une application EJB

Dans cette partie, nous allons étudier en détail les étapes de développement d'une application à base de composants EJB exécutée sur la plate-forme JonAS (une implémentation de J2EE). Nous commençons par l'écriture du composant EJB et ses descripteurs de déploiement et d'empaquetage, puis l'écriture le composant Client, enfin la mise en ouvre [12]. Trois étapes sont à suivre :

Etape 1 : Développement du composant EJB

Cette étape est définit quatre phases à suivre pour réaliser un EJB. Ce sont :

a- L'écriture du composant EJB :

Un composant EJB se compose de trois entités (trois classes d'un point de vue d'un programmeur):

L'interface distante : cette interface définit les méthodes métier d'un EJB ;
L'interface maison : elle définit les méthodes qu'un client peut invoquer pour créer, trouver ou supprimer un EJB.

Lorsqu'un Client veut utiliser un EJB, il se sert de JNDI (**Java Naming and Directory Interface**) pour obtenir une référence à un objet qui est une instance de l'interface maison;

L'Enterprise Java Bean : Qui est l'implémentation de notre composant.

b- La création des descripteurs de déploiement de l'EJB
Pour un EJB, il y a deux descripteurs à spécifier : Le descripteur standard de Sun MicroSystems et le descripteur spécifique à JOnAS.

Descripteur de déploiement standard : le programmeur de l'EJB doit fournir avec son composant un descripteur de déploiement. Ce fichier décrit, par exemple, quelle classe est l'implémentation, l'interface locale et l'interface distante de l'EJB.

Descripteur de déploiement spécifique à JOnAS : certaines informations nécessaires au déploiement de l'EJB dans JOnAS ne sont pas dans le descripteur précédent. Pour résoudre ce problème, il faut créer un autre descripteur de déploiement spécifique à JOnAS. Ce document devra s'appeler jonas-ejb-jar.xml.

En résumé, un développeur écrit le composant, ses deux interfaces et les descripteurs de déploiement. Puis, il les propose à déployer sous forme d'un fichier JAR.

c- L'empaquetage de l'EJB
L'EJB doit être empaqueté dans une archive JAR (nom_fichier.jar) qui devra contenir : les classes de l'EJB et les deux descripteurs de déploiements (ejb-jar.xml et jonas-ejb-jar.xml).

d- Le déploiement de l'EJB
Pour déployer l'EJB sous JOnAS, le fichier ejb-jar, qu'on a définit dans la phase de packaging, doit posséder les classes d'interposition interfaçant notre Entreprise Java Bean avec les services offerts par notre Serveur d'EJB. Ces classes peuvent être crées à l'aide de l'utilitaire "GenIC" fourni avec JOnAS.

Ensuite, pour déployer notre EJB dans JOnAS, il suffira de recopier le fichier ".jar" dans un répertoire secondaire de jonas et de modifier la configuration de JOnAS.

Etape 2 : l'implantation du Client de l'EJB
Dans cette phase, nous allons implanter le code du client de notre EJB. Il suffit d'écrire les lignes de code qui permettent de trouver les interfaces locales, invoquer des méthodes, etc.

Etape 3 : La mise en ouvre
Pour réaliser la compilation, l'empaquetage (*Packaging*) et le déploiement, nous allons utiliser un script spécifique qu'on doit écrire (build.bat). Il lance la compilation, le packaging et le déploiement de notre ejb ainsi que la compilation de notre client. Aussi, il faut éditer le fichier *jonas.properties*. Enfin, il nous reste qu'à lancer l'exécution de notre application, après avoir lancer la plate-forme JOnAS, par le script build.bat.

1.5 Évaluation

Le modèle EJB était le premier modèle de composant serveur non visuel qui a vu le jour par la publication de la spécification 1.0 en novembre 1997. L'utilisation des *containers* pour accueillir des composants, la prise en compte du déploiement, ainsi que la gestion de certaines propriétés non-fonctionnelles sont autant d'atouts pour produire et utiliser des composants EJB réutilisables.
En résumé, les points forts de cette technologie sont :

Les EJBs contiennent la logique applicative : ils sont plus légères et plus souples, car elle ne contiennent pas du code non-fonctionnel ;

Les EJBs sont portables : une application peut être construire à partir d'EJBs existants et être déployée sur n'importe quel serveur compatible au J2EE ;

Mise en évidence de plusieurs catégories de programmeurs (Taxinomie des rôles), c'est-à-dire nous distinguons plusieurs métires tels que : fournisseur de conteneur EJB, fournisseur de serveur EJB, assembleur d'applications, etc. qui interviennent selon différentes phases (implantation, déploiement, exécution) pour concevoir une application.

Les points faibles du modèle EJB sont :

1. Les EJBs restent une solution propriétaire de Sun MicroSystems, destinés uniquement à une **utilisation en Java** ;
2. Le faible nombre de services non-fonctionnelles proposés ne satisfait pas la demande des utilisateurs ;
3. La fastidieuse configuration des descripteurs de déploiement XML, le packaging des classes et le déploiement au sein du serveur d'application deviennent rapidement d'une lourdeur extrêmement rebutante [Mougin et Barriolade 2001].

4. Les EJBs permettent une réutilisation aisée des composants développés. Bien que, leur conception paraisse, à notre vue, un peu plus compliquée par rapport aux composants CCM et COM+ qui génèrent plusieurs fichiers automatiquement.

5. Même si le modèle EJB était venu au bon moment, il n'a pas trop évoluée depuis, ce qui explique l'apparition d'autres modèles tels que COM+, CCM.

2. Les composants COM+

2.1 Introduction

Au début des années 90, le modèle COM (Component Object Model) s'est imposé comme une technologie Windows, importante et évolutive. Par définition, tout objet COM peut être distribué [14]. En d'autre terme, il peut être instancié à partir d'une autre machine via un simple appel réseau. On parle alors de DCOM (Distribueted COM), qui est l'extension de COM prenant en compte l'aspect distribué (il permet la communication entre objets situés sur des machines différentes) [15]. L'architecture Internet de Microsoft, pour développer et supporter les applications réparties, construite autour de DCOM s'appelle DNA (*Distribueted Network Application*) [Fabrice 2001].

Sous Windows 2000, DCOM se change de nom pour devenir COM+. On peut dire alors que COM+ est l'évolution de COM. Dernièrement, Microsoft a développé un environnement de création d'application Web nommé « .NET » (prononcer dot-Net) [13].

2.2 Le nouvel environnement des composants COM+ : .NET

La naissance de la technologie .NET a été annoncée en juillet 2001 lors des conférences PDC (Professionel Developers Conference). En réponse à l'évolution du Web, Microsoft a proposé la nouvelle technique .NET.

L'environnement .NET a été conçu pour supporter **tous les langages de MicroSoft**, quel que soit leur type (objet, procédural, etc.). Pour atteindre cet objectif, les applications .NET utilisent le langage pivot MSIL (Microsoft Intermediate Language). MSIL est un formalisme orienté objet qui se situe entre le langage de programmation et le code compilé. Le code intermédiaire est compilé lors de l'exécution du programme. Cette caractéristique permet principalement de rendre les programmes indépendants des systèmes d'exploitation. L'aspect objet de ce formalisme facilite la définition de programmes écrits en C#, C++ ou VB. Bien que conçu pour supporter des langages très différents les uns des autres, seuls les langages orientés objets proposés par Microsoft peuvent actuellement être traduits vers le langage intermédiaire.

2.3 L'environnement d'exécution .NET

L'environnement .NET peut être définie de la façon suivante : Il s'agit d'un ensemble de Services communs, utilisables depuis plusieurs langages.
.NET définit une architecture proche de CORBA. Plus précisément, il utilise le langage intermédiaire MSIL[15] (MicroSoft Intermediate Language) pour uniformiser la représentation des applications et le protocole de communication SOAP pour permettre l'invocation de méthodes distantes à travers le réseau Internet. Toutes les applications écrites dans le langage intermédiaire ne sont pas disponibles pour une méthode distante.

[15] MSIL : c'est un langage intermédiaire constitué d'instructions élémentaires de type "assembleur évolué". Tous les langages supportés par la plate-forme .NET peuvent être compilés en code intermédiaire MSIL. Ce code intermédiaire peut, ensuite, être exécuté par le CLR, l'environnement d'exécution de la plate-forme .NET. MSIL permet l'indépendance au langage de la plate-forme .NET.

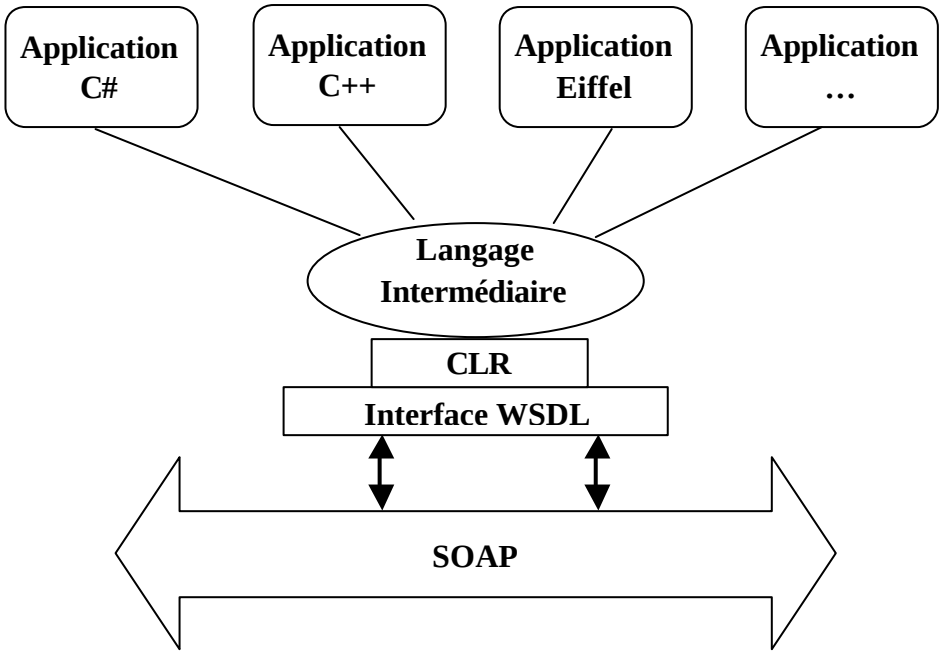

Figure B.2 : Architecture de l'environnement .NET

Les applications dans .NET s'exécutent en code intermédiaire MSIL (MicroSoft Intermediate Language) dans une sorte de machine virtuelle dénommée CLR (Common Language Run-time). MSIL est un langage pivot, il permet de représenter dans un même formalisme des applications programmées dans différents langages tels que C#, C++, Eiffel, VB, etc. Les applications compilées sous la forme de code intermédiaire se présentent sous la forme de binaires exécutables portables
CLR assure la compilation en code natif des programmes (**les logiciels .NET sont compilés, contrairement** à Java), la gestion mémoire, la gestion des erreurs et les contrôles de sécurité. Il fournit l'accès à l'OS et aux ressources système. Il est au cœur de l'unification des langages. Le langage C# (prononcer C Sharp), crée par Anders Hejlsberg, est un nouveau langage né avec .NET. Il est actuellement le plus proche du langage intermédiaire.

2.4 Évaluation

Le développement d'applications client-serveur distribuées, à l'aide de CCM ou DCOM/COM+, a démontré comment définir des services au moyen d'interfaces. Les objets peuvent ainsi résider sur toute machine (transparente de l'emplacement), et le client n'a pas besoin de les rechercher. Notons que l'architecture de ce modèle, au contraire aux modèles EJB et CCM, se base sur les objets COM.

COM+ est un modèle multi-langage (il peut être programmé en l'un des 27 langages de programmation, cités dans sa spécification), mono-plate-forme (Windows). Or le langage Java est exclu. Donc, on peut dire qu'il ne supporte que les langages de Microsoft. En plus, un composant COM+, pour qu'il puisse s'exécuter, doit être installé.

Le principe suivi par Microsoft est très important (on parle de la possibilité de programmer en n'importe quel langage), mais malheureusement, et eu égard des intérêts commerciaux, le nouveau environnement de Microsoft .NET, ne nous permet pas de programmer les composants par d'autres langages **que ceux de Microsoft**.

Enfin, si on est obligé de traduire tous les programmes en MSIL, d'un côté on double le travail des programmeurs, et d'autre côté on peut juger que la technologie .NET ne respecte pas l'aspect multi-langage. Bien que Microsoft permet d'écrire le code MSIL avec une syntaxe Visual Basic, ou C++, ou Eiffel…etc.

3. Les Composants CORBA, CCM

3.1 Introduction

L'utilisation des objets répartis CORBA (Common Object Request Broker Architecture) établis en 1991 par l'OMG (Object Management Group), n'a pas réussit d'atteindre la simplicité visée pour concevoir des applications distribuées à base d'entités logicielles hétérogènes.

La spécification CORBA 3.0 a été publiée par l'OMG (Object Management Group) en juillet 2002. Elle a introduit le modèle CCM, *CORBA Component Model*. Ce modèle propose toute une structure pour définir un composant CORBA, son comportement, son intégration dans un conteneur (ou application) et son déploiement dans l'environnement distribué CORBA[16]. CORBA supporte l'interaction, à travers le Web, entre des composants écrits en différents langages distribués et exécutés sur des ordinateurs avec différents systèmes d'exploitation. Il intègre aussi, des descripteurs pour la configuration, la définition de l'assemblage et le déploiement des composants.

3.2 L'environnement d'exécution des composants CCM

L'architecture typique du modèle CCM, présentée sur le schéma ci-dessous, 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, Events, etc.

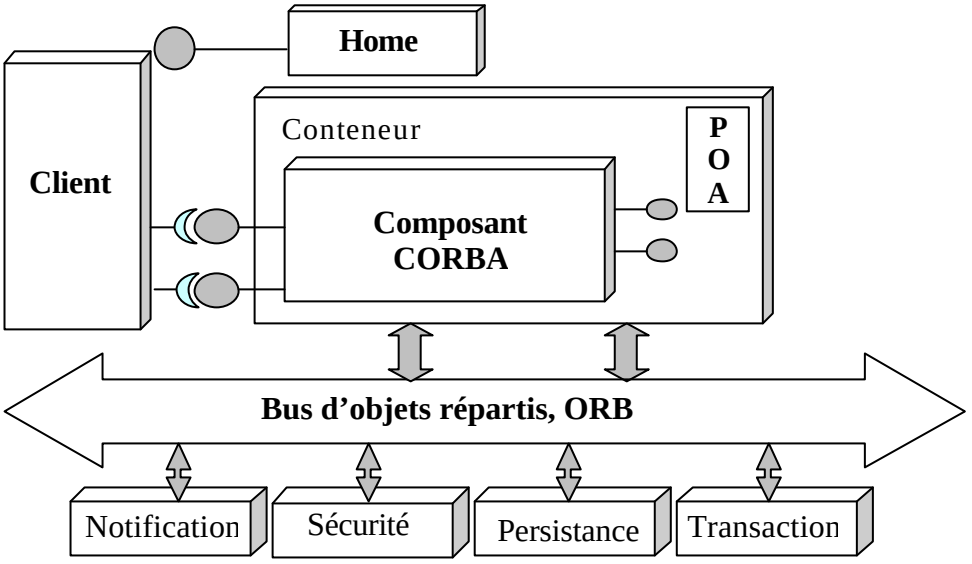

Figure B.3 : L'environnement d'exécution des composants

Les Conteneurs CCM jouent le rôle d'une interface entre les composants CORBA et leurs clients de manière à gérer toutes ses communications en reproduisant les interfaces du composant. Ainsi, le Conteneur est l'intermédiaire entre le composant et les services fournis par l'ORB.

3.3 Démarche de conception d'une application CCM

L'objectif de CCM est de décomposer le développement d'applications à base de composants en plusieurs métiers (rôles) : le programmeur s'intéresse aux fonctionnalités intrinsèques des

[16] CORBA (Common Object Request Broker Archtecture) : C'est une architecture basée sur un bus ORB (Object Request Broker). Ce dernier est chargé d'assurer les collaborations entre les applications.

composants, l'assembleur sélectionne et connecte les composants entre eux pour former une application. Avant le déploiement, un empaqueteur (*packager)* est chargé d'empaqueter (packaging) les composants et l'application en choisissant notamment les politiques concernant les services à appliquer aux composants [Flissi et Merle 2002].

Les étapes pour la conception et la mise en œuvre d'une application sur la plate-forme OpenCCM proposée par le laboratoire LIFL (Laboratoire d'Informatique Fondamentale de Lille), en citant les acteurs qui participent aux différentes étapes de conception, sont les suivantes :

1. **Mise en place de l'environnement :** pour cela il suffit de lancer un script spécialisé (envi_OpenCCM.bat) ;

2. **Définition du type de composants :** le concepteur définit les types des composants, dans un fichier IDL, à l'aide du IDL3 ;

3. **Projeter des définitions du composant (le fichier IDL3) en IDL2 :** pour permettre aux développeurs l'implantation des composants. Cette projection produit également, un descripteur de composants qui sera compléter par le développeur ;

4. Générer les squelettes et les souches à partir du fichier IDL2 ;

5. Compléter, par le développeur, les fichiers Java suivants :
   - les applications serveur et leurs Homes ;
   - les applications Cliente et leurs Homes ;
   - le fichier qui prend en charge le déploiement de l'application ;

6. Compiler l'ensemble des fichiers ;

7. Empaquetage, au sein d'un archive JAR, chaque type de composant avec son descripteur et une configuration par défaut ;

8. Déployer des archives des composants.

3.4 Évaluation

Le modèle CCM a montré son efficacité et sa robustesse par la proposition d'une série de modèles à suivre pour concevoir une application distribuée (c'est-à-dire il a tracé une démarche typique du processus globale de production d'une application distribuée en commençant de l'implantation, puis le paquetage, ensuite le déploiement et enfin nous finirons par l'exécution). En plus, il définit le modèle abstrait qui décrit la structure interne d'un composant. Ces modèles permettent, en outre, de conceptualiser les différents éléments d'un composant et de son intégration dans une application.

Les composants CCM sont des composants multi-interfaces. Ils permettent d'être indépendants de la plate-forme et du langage de programmation. En plus, ils sont interopérables avec les composants EJB. Cependant, ce modèle présente quelques anomalies, il ne permet pas de définir des composants composites (assemblage de plusieurs composants pour construire un seul composant) car il définit des interfaces co-localisés [ACCORD2 2000]. Un autre regret, d'après [Yahiaoui 2002], est que le modèle CCM ne définit pas la notion de **Connecteur**[17] dans son modèle abstrait. C'est-à-dire, le boutstrap (l'interconnexion des composants de l'application) se fait par un fichier écrit par le développeur.

[17] Un Connecteur est une entité définissant l'interaction entre plusieurs composants. Il possède un certain nombre d'interfaces (appelées rôles) et une description globale de son comportement.

Le principe suivi par le modèle CCM est : "*mieux décrire, puis programmer*". Mais la question qui se pose : Pour quoi l'obligeance de projeter les fichier en IDL3 en IDL traditionnel ?.Conclusion et Bilan

Vu la grande interopérabilité entre les deux modèles : EJB et CCM, nous établirons une comparaison entre ces modèles.

4. CCM vs. EJB

Le modèle EJB, puis CCM, ont introduit la notion de conteneur permettant de fournir des services non-fonctionnels aux composants : Transaction, Persistance, Sécurité, Activation / Passivation des composants, etc (cf. figure B.4). Ces services sont gérés par les conteneurs.

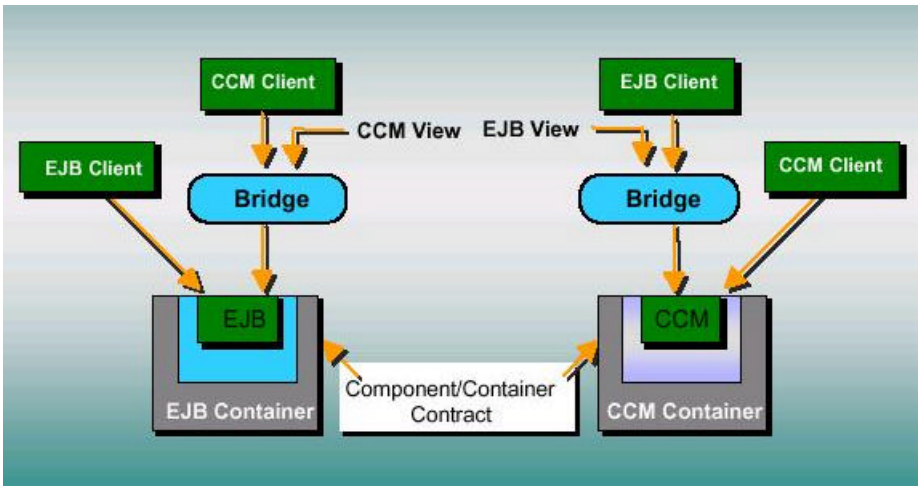

Figure B.4 : L'interopérabilité entre EJB et CCM

Le modèle CCM est, comme le disent David Frankel [5], "*specifically designed to play very well with EJB*". Donc ce sont deux modèles composants serveur interopérables. Ils ont les points communs suivants :

1. Ils sont destinés à n'intégrer que du code fonctionnel ;
2. La *home* existe pour les deux modèles EJB et CCM, qui s'occupe des mêmes fonctionnalités ;
3. Les deux modèles utilisent les fichiers XML pour écrire les descripteurs de déploiement des composants ;
4. Ils proposent une taxinomie de rôles tout au long le cycle de conception d'applications à base de composants;

En plus de la projection standard d'IDL (Interface Description Language) vers Java, définit par l'OMG, une projection Java-vers-IDL a été définit. Ceci autorise la projection standardisée des types Java utilisés par les EJB vers l'IDL CORBA [Edwards et Deshpande 1988].

La forte interopérabilité entre ce dernier et les EJBs a mené à naître un mariage entre ces deux modèles, monté dans l'implémentation EJCCM (Enterprise Java CORBA Component Model) développée par l'équipe de recherche de technologie de la programmation de CPI (Computational Physics, Inc).

Bien que, le modèle CCM se base sur la génération automatique des souches et squelettes, ce que lui donne plus de souplesse. Mais c'est le même cas pour les EJBs, c'est le développeur qui va coder tous les fichiers de l'application. Cependant, il y a des différences entre les EJBs et les composants CORBA qui sont :

CCM est compatible à tous les langages, tandis que EJB doit être implanté entièrement en Java ;
Le nombre d'interfaces fournies par CCM est illimité, tandis que pour les EJBs il égale à deux (Home et Object) ;

En résumé, CCM a reprend la majorité des concepts des Enterprise Java Beans et a introduit de nouvelles techniques comme la description des ports et des connexions statiques entre composants.

5. Conclusion

Dans cette annexe, nous avons présenté les trois principaux modèles de composants existants : EJB, COM+ et CCM. Ils sont apparus en résultat de l'impossibilité d'atteindre la simplicité pour concevoir des applications distribuées générées par l'emploi des objets (leurs prédécesseurs : CORBA pour le CCM, JavaBeans pour les EJB et COM pour le COM+). Ces modèles fournissent une simplification de développement des applications logicielles distribuées. La manière de leur assemblage pour concevoir des logiciels "CBSE" permet, en favorisant la réutilisabilité, aux développeurs de produire des applications à base d'entités logicielles hétérogènes de meilleure qualité et de façon plus rapide et moins coûteuse.

L'une des avancées les plus importante dans l'approche composant est la capacité offerte par l'infrastructure de gérer certains services non-fonctionnels qui restent jusqu'à l'heure actuelle statiques. Cette gestion s'effectue souvent en utilisant des **Conteneurs** qui maîtrise les communications entre les composants. Avec cette séparation, les composants contiennent que du fonctionnel, ce qui permet aux développeurs de se concentrer sur le code applicatif de l'application.

Le modèle EJB était le premier modèle de composant serveur, il a vu le jour en novembre 1997. En revanche, on trouve plus de documentations et d'implémentations de ce modèle. Sun a visé dès la proposition des EJBs de les rendre utilisables sur n'importe quelle plate-forme **supportant Java**. D'où la nécessité de programmer en Java.

Ensuite, le modèle COM+ puis son environnement .NET sont apparu. Notons que l'environnement .Net est proposé avec 27 langages de programmation, mais le fait qu'il faille passer par le langage intermédiaire MSIL, nous nous permettons de dire que .NET ne supporte qu'un seul langage. Ce qui est le problème des EJBs, car ils supportent que le langage Java.

Devant l'obligeance de programmer en Java imposée des EJBs, et l'exigence d'installer les composants pour pouvoir les exécutés imposée par DCOM/COM+, l'OMG a annoncé dernièrement la naissance de modèle CCM en 2001. Un modèle interopérable avec les EJBs et qui supporte tous les langages de programmations.

Les modèles CCM et COM+ se basent sur la définition des composants par OMG IDL, pour bénéficier des fichiers générés automatiquement, et on doit coder que le code fonctionnel des composants. Tandis que, avec le modèle EJB on doit tout écrire à la main, ce qui vient pénible. Or, les trois modèles présentés proposent une taxinomie de rôles durant le cycle de conception des logiciels, mais celui du modèle CCM couvre mieux toutes les étapes. Bien que, les trois modèles sont persistants, car leurs composants peuvent s'archivés dans des fichiers (.jar).

Annexe C : Exemple de description du composant-composite voiture

La description du composant-composite voiture en IDL3+

```
module demoCar {

// définition des interfaces
    interface WheelInterface {
        void WheelMove() ;
    };

    interface BodyInterface {
        void BodyColor() ;
    };

    interface EngineInterface {
        void StartEngine();
    };
//=================================================================
//             La description du composant composite Car
//=================================================================
    composite Car {

       attribute string the_name;
 //La déclaration des variables des interfaces du composant composite Car
       type : synchrone ;
       port_in WheelInterface ;
       port_in BodyInterface ;
       port_in EngineInterface ;

       composeOf Whell {

      attribute string the_name;
//La déclaration des variables de la relation de composition Car-Wheel
      exclusive : true;
      dependance : true;
      prevalence : true;
      cardinality : 4;
       };
//=================================================================
       composeOf Body
       {
        attribute string the_name;
//La déclaration des variables de la relation de composition Car-Body
      exclusive : true;
      dependance : true;
      prevalence : true;
      cardinality : 1;
       };
//=================================================================
       composeOf Engine {

        attribute string the_name;
//La déclaration des variables de la relation de composition Car-Engine
      exclusive : true;
      dependance : true;
      prevalence : true;
      cardinality : 1;
       };

   //=================================================================
```

```
};
//La déclaration de la maison du composant composite Car
    home CarHome manages Car
    {
    };

//=================================================================
//              La description du composant Wheel
//=================================================================
    component Wheel
    {
        attribute string the_name;
//La déclaration des interfaces du composant Wheel
        provides WheelInterface  for_carW ;
    };

//La déclaration de la maison du composant Wheel
    home WheelHome manages Wheel
    {
    };
//=================================================================
//             La description du composant Body
//=================================================================
    component Body
    {
        attribute string the_name;
//La déclaration des interfaces du composant Body
        provides BodyInterface for_carB ;
    };

//La déclaration de la maison du composant Body
    home BodyHome manages Body
    {
    };

//=================================================================
//             La description du composant Engine
//=================================================================
    component Engine
    {
        attribute string the_name;
//La déclaration des interfaces du composant Engine
        provides EngineInterface for_carE ;

    };

//La déclaration de la maison du composant Engine
    home EngineHome manages Engine
    {
    };
};
```

Comme le montre le tableau ci-dessous, la projection de ce fichier écris en IDL3.1 donne la description de neuf composants : le composite et ses trois autres composants. En plus trois composants qui correspondent aux connecteurs de composition.

La projection de la description de la voiture en IDL3

```
module demo1
{
// définition des interfaces
    interface WheelInterface
    {
        void WheelMove() ;
    };

    interface BodyInterface
    {
        void BodyColor() ;
    };

    interface EngineInterface
    {
      void StartEngine();
    };
//=====================================================================
//          La description du composant composite CarComposite
//=====================================================================
    component CarComposite
    {
        attribute string the_name;
//La déclaration des interfaces du composant composite CarCompositer
        uses WheelInterface  to_wheel ;
        uses BodyInterface   to_body ;
        uses EngineInterface to_engine ;
    };
//La déclaration de la maison du composant composite CarComposite
    home CarCompositeHome manages CarComposite {
    };
//=====================================================================
//             La description du composant Wheel
//=====================================================================
    component Wheel
    {
        attribute string the_name;
//La déclaration des interfaces du composant Wheel
        provides WheelInterface  for_carW ;
    };
//La déclaration de la maison du composant Wheel
    home WheelHome manages Wheel {
    };
//=====================================================================
//             La description du connecteur WheelConnector
//=====================================================================
    component WheelConnector
    {
      attribute string the_name;
//La déclaration des variables de la relation du connecteur WheelConnector
      attribute boolean exclusive;
      attribute boolean dependance;
      attribute boolean prevalence;
      attribute int cardinality;
//La déclaration des interfaces du connecteur WheelConnector
      provides WheelInterface for_carWC ;
        uses WheelInterface to_wheelC ;
    };
//La déclaration de la maison du connecteur WheelConnector
    home WheelConnectorHome manages WheelConnector {
    };
//=====================================================================
//       La description du composant Engine
```

```
//=================================================================
    component Engine
    {
        attribute string the_name;

        provides EngineInterface for_carE ;
    };
//La déclaration de la maison du composant Engine
    home EngineHome manages Engine{
   };
//=================================================================
//             La description du connecteur EngineConnector
//=================================================================
    component EngineConnector
    {
      attribute string the_name;
//La déclaration des variables de la relation du connecteur EngineConnector
      attribute boolean exclusive;
      attribute boolean dependance;
      attribute boolean prevalence;
      attribute int cardinality;

      provides EngineInterface for_carEC ;
        uses EngineInterface to_engineC ;
    };
//La déclaration de la maison du connecteur EngineConnector
    home EngineConnectorHome manages EngineConnector {
    };
//=================================================================
//             La description du composant Body
//=================================================================
    component Body {

        attribute string the_name;

        provides BodyInterface for_carB ;
    };
//La déclaration de la maison du composant Body
    home BodyHome manages Body {
    };
//=================================================================
//             La description du connecteur BodyConnector
//=================================================================
    component BodyConnector
    {
      attribute string the_name;
//La déclaration des variables de la relation du connecteur BodyConnector
      attribute boolean exclusive;
      attribute boolean dependance;
      attribute boolean prevalence;
      attribute int cardinality;

      provides BodyInterface for_carBC ;
          uses BodyInterface to_bodyC ;
    };
//La déclaration de la maison du connecteur EngineConnector
    home BodyConnectorHome manages BodyConnector {
    };
};
```

La figure ci-dessous montre la relation de composition du composant-composite voiture décri dans le fichier en IDL3 de cette annexe. Elle présente clairement que cette voiture est composée d'un moteur, une carrosserie et des roues. Ces derniers sont liés au composant-composite voiture via trois connecteurs. La communication entre le composant-composite voiture, les connecteurs et les composants de la voiture (Moteur, Carrosserie et Roues) se faite par des liens facette-receptacles en raison que l'invocation des méthodes est synchrone.

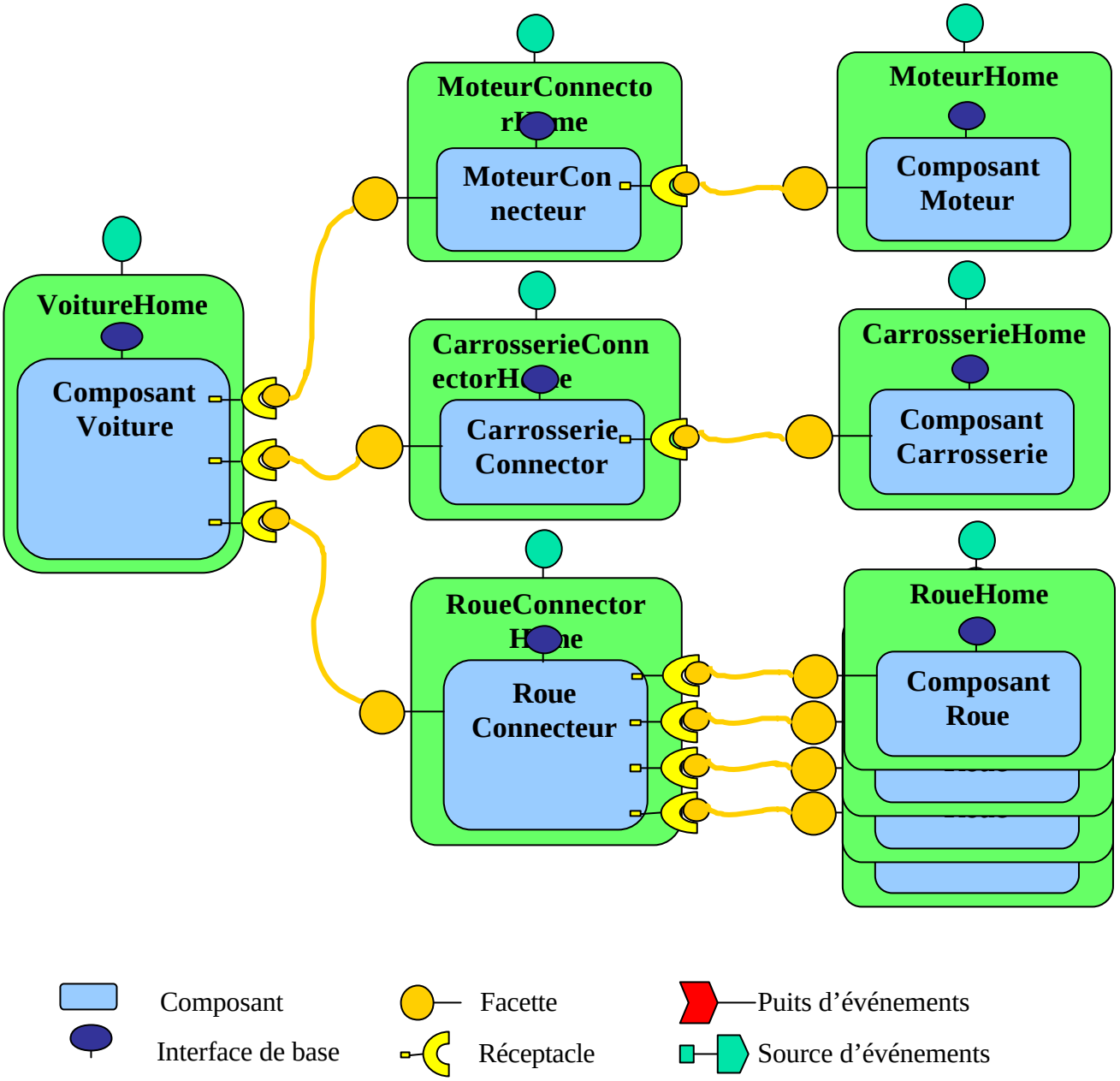

Figure B.1 : Interconnexion entre le composant voiture et ses composants.