République Algérienne Démocratique et Populaire
وزارة التعليـــــم العـــالي والبحث العلمي
Ministère de l'Enseignement Supérieur et de la Recherche Scientifique
UNIVERSITÉ DES SCIENCES ET DE LA TECHNOLOGIE D'ORAN Mohamed Boudiaf

Faculté des Mathématiques et Informatique
Département d'Informatique

Option : Système, Réseaux et bases de données
(SRBDD)

MÉMOIRE
Présenté par

M. BENKECHIDA Fouad

Pour l'obtention du diplôme de Magister en Informatique

Thème

L'Appréciation de la qualité d'une Application Composée

Devant la commission d'examen composée de :

Qualité	**Nom et Prénoms**	**Grade**	**Ets d'origine**
Présidente :	M^{me}. BELBACHIR Hafida	Professeur	USTO-MB
Rapporteur :	M.A.CHOUARFIA	Professeur	USTO-MB
Examinateurs :	M. B.YAGOUBI	Professeur	Université d'ORAN
	M^{elle}. NOUREDDINE Myriam	M.conf. A	USTO-MB
	M^{me}. ZAOUI Lynda	M.conf. A	USTO-MB

Année Universitaire : 2013-2014

Remerciements

Je remercie tout d'abord le dieu le tout puissant pour le courage qu'il m'a donné afin d'arriver à réaliser ce travail,

Je tiens à exprimer mes vifs remerciements à mon encadreur Mr A.CHOUARFIA, pour ma voire donné l'opportunité de mener à bien ce travail de recherche. J'ai particulièrement apprécié sa disponibilité, ses précieux conseils, son ouverture d'esprit et les nombreuses discussions qui ont animé ces dernières années,

Mes remerciements vont aussi aux membres du jury :
Mme. BELBACHIR Hafida,
M. B.YAGOUBI,
Melle. NOUREDDINE Myriam,
Mme. ZAOUI Lynda
D'avoir bien voulu juger ce travail.

Mes remerciements les plus profonds, du fond du cœur, vont à mes parents pour ses expressions inégalement encourageantes, leurs patiences et toutes les valeurs qu'ils m'ont inculquées, maman merci pour tout le soutien moral.

Je remercie mes amis pour leur soutien, leurs encouragements et leur aide.

Je dédie cette thèse
à tous ceux qui m'ont soutenu moralement
pendant la rédaction de ce mémoire.
Plus particulièrement à Papa & Maman,

SOMMAIRE

Sommaire

Introduction Générale……04

Chapitre 1 : Les composants Logiciels

Introduction 06
1- Des objets aux composants 07
2- Terminologie 08
3- Définition d'un composant logiciel 09
4- Architecture d'un composant 10
5- Caractéristiques d'un composant logiciel 11
6- Principes des composants logiciels 12
6.1- La définition……...13
6.2- L'implémentation……...13
6.3- Le déploiement……...14
6.4- L'exécution……...14
7- Etude de quelques modéles de composants 15
7.1- Java beans...……...16
7.2- Entreprise JavaBeans : modèle composants serveur pour la plate-forme Java……...17
7.3- Le modèle COM……...20
7.4- Le Modèle de composant CCM……...23
8- Comparaison et évaluation des modéles de composants existants 26
Conclusion 29

Chapitre 2 : L'application Composée

Introduction 30
1- La notion de contrat 31
2- Systèmes construits par assemblage de composants 32
3- Architecture logicielle 33

4- Régles d'évolution dynamique 38
5- Cycle de développement d'une application logicielle composée 39
6- Assemblage de composants logiciels 40
6.1- Les connecteurs logiciels……...42
6.2- Classification des connecteurs……...43
7- Assemblage correct de composant logiciels 44
Conclusion……..…..……..44

Chapitre 3 : La qualité Logicielle

Introduction 45
1- Définitions..... 45
2- Importance de la qualité des logiciels 47
3- Les facteurs, critères et métriques de qualité 48
4- Normes et modéles de qualité logicielle 50
Conclusion……53

Chapitre 4 : Contribution

Introduction 54
1- Modéle de qualité pour les composants logiciels..... 54
2- Aide multicritère à la décision 58
3- Méthodologie- démarche 59
4- Le processus d'évaluation de la qualité d'un assemblage de composant 60
4.1- Etape des exigences de qualité...……...60
4.2- Préparation de l'évalution..……...60
4.3- La qualité des attributs...……...61
4.4- Spécification de la qualité des composants logiciels..……...62
4.5- Illustration……..……...63
4.6-Architecture du logiciel ……. ……...64

Conclusion générale……..……70
Bibliographie……...……...71

Liste des figures

Figure 1.1 : Représentation graphique des ports présentant à la fois des interfaces requises et offertes 09
Figure 1.2 : Architecture d'un composant 10
Figure 1.3 : Interfaces, connexions et composition des composants 13
Figure 1.4 : Conteneur et structure d'accueil 15
Figure 1.5 : Architecture de la plate-forme EJB 18
Figure 1.6 : L'environnement d'exécution des composants EJB 20
Figure 1.7 : Objet COM 21
Figure 1.8 : Containment 21
Figure 1.9 : Agrégation 22
Figure 1.10 : Architecture de l'environnement .Net 23
Figure 1.11 : Le modèle abstrait de composants CCM 24
Figure 1.12 : Composant CORBA 25
Figure 1.13 : L'environnement d'exécution des composants CCM 26
Figure 2.1 : Processus de production et d'utilisation de composants 30
Figure 2.2 : Vue générale d'un système à base de composants 32
Figure 2.3 : L'architecture d'un système logiciel 34
Figure 2.4 : Vue structurelle d'une architecture 35
Figure 2.5 : Vue syntaxique 36
Figure 2.6 : Vue sémantique 37
Figure 2.7 : Vue d'évolution 38
Figure 2.8 : Cycle de développement 39
Figure 2.9 : Relation de dépendance entre les interfaces de composants 41

Liste des figures

Figure 3.1 : Relation entre erreur, faute et panne. 45
Figure 3.2 : Différents types de qualité 48
Figure 3.3 : Relation entre facteurs, critères et métriques de qualité 49
Figure 4.1 : Spécification de la qualité d'un composant par un fichier XML. 62
Figure 4.2 : Le schéma de la base de données 65
Figure 4.3 : L'interface principale 66
Figure 4.4 : Le menu contextuel 66
Figure 4.5 : Définition des attributs de composant 67
Figure 4.6 : Définition des composants d'assemblage 68
Figure 4.7 : Définition des attributs d'assemblage. 68
Figure 4.8 : Calcul de la valeur globale de la qualité 69
Figure 4.9 : Liste d'assemblages avec leur qualité……...69

Liste des tableaux

Tableau 1.1 : Comparaison des modèles de composants. 28

Tableau 4.1 : Modèle de qualité pour les composants logiciels 54

Tableau 4.2 : Les attributs de qualité relatifs au sous caractéristiques observables à l'exécution 55

Tableau 4.3 : Les attributs de qualité relatifs au sous caractéristiques observables pendant le cycle de vie. 57

INTRODUCTION GENERALE

Introduction Générale

Face à une croissance constante de la taille et de la complexité des systèmes informatiques, leur conception est devenue modulaire c'est-à-dire que le développement de ces systèmes est réparti entre différentes équipes.

Une première approche de la modularité porte sur la notion de procédure. La programmation systémique consiste à analyser un problème par décomposition fonctionnelle descendante. Les types de données manipulées sont définis par ailleurs. Cette séparation dans l'analyse entre les données structurelles et les fonctions rend très difficile la maintenance et la réutilisation du code.

Le but de la programmation objet est de regrouper les fonctions et les types de données qu'elles manipulent au sein d'une même entité logique : la classe. Un problème est alors analysé comme un ensemble d'objet en interaction. **[YAH08]**

La technologie objet qui a apporté de grands bouleversements dans le domaine informatique dans les années 1990, n'est pas suffisante pas pour répondre aux besoins devenus de plus en plus complexes.

La réutilisabilité des objets est limitée à cause de certains problèmes liés au paradigme objet comme la réutilisation manuelle des services nécessaires tels que la persistance, la sécurité, etc. de plus il est difficile d'intégrer des objets hétérogènes conçus pour différents contextes applicatifs. A cause des limites de la technologie objet et aussi aux nouveaux besoins, comme la complexité croissante des applications (Concurrence/distribution, Hétérogénéité des plates-formes), et l'adaptabilité des applications (reconfiguration dynamique, déploiement), que le paradigme « composant » est apparu.

L'ingénierie du logiciel basée composant, appelée *CBSE* (Component-Based Software Engineering), est actuellement la technique la plus recommandée en terme de réutilisation et de réduction des coûts de développement. Non seulement cette approche permet de réutiliser les développements déjà réalisés en interne, mais aussi d'utiliser directement des composants commerciaux. Le succès à long terme de développement par composant repose sur la capacité de prédire la qualité des systèmes obtenus. Pour cette raison, les chercheurs et les praticiens sont désireux de développer des techniques pour la sélection des composants efficaces et de la composition.

Le développement d'applications basées sur cette technique repose sur le principe d'assemblage de composants interagissant entre eux. Cet assemblage se fait notamment en mettant en relation les interfaces de composants fournissant un service avec les interfaces d'autres composants nécessitant ce service. L'interaction entre les composants d'un assemblage est essentielle à la qualité globale du système. Deux étapes doivent être faites quand un système est construit à partir d'un ensemble de composantes: construire toutes les assemblées possibles et ensuite d'analyser chacune de ces assemblées, afin d'obtenir la solution finale.

L'assemblage de composants n'est cependant pas une tâche aisée et il pose de nombreux problèmes. Parmi ceux-ci, notons les problèmes d'incompatibilité d'interfaces, de langages, de méthodes de conception, ou encore de comportements.

De nombreux travaux portent sur ces sujets. Cependant, ils sont pour la plupart orientés très fortement sur l'aspect implémentation de ce qu'il est d'usage appeler la composition logicielle. Nous pensons pour notre part qu'il est important de continuer sur cette axe avec l'objectif de trouver une bonne combinaison de composants qui conduit à un système logiciel qui répond le mieux sur le plan qualitatif aux exigences spécifiques du client.

L'objectif de notre travail est de présenter un processus d'évaluation qui permet de choisir, devant plusieurs compositions possibles, la composition dont la qualité est meilleure et qui répond le mieux aux caractéristiques fonctionnelles et non fonctionnelles du client.

L'originalité de cette approche est de permettre selon une spécification des exigences en termes de caractéristiques, en utilisant une technique de prise de décision multicritère (des critères d'appréciation pondérés), d'obtenir un résultat global de l'évaluation de la qualité pour chaque assemblage. Les qualités obtenues sont comparées pour la prise de décision du choix du meilleur assemblage.

Ce mémoire est organisé en quatre (04) chapitres :

- **Le chapitre 1** donne un aperçu sur les composants logiciels : définition, architecture, caractéristiques et les différents types d'abstraction d'un composant logiciel,

- **Le chapitre 2** décrit l'assemblage de composants logiciels : définition d'un assemblage correct, les différentes catégories des dépendances entre les interfaces des composants, et présente aussi le cycle de développement d'une application logicielle composée.

- **Le chapitre 3** présente les différentes notions concernant la qualité logicielle, les différentes terminologies, qu'est-ce que la qualité de service ?, l'importance de la qualité des logiciels, les différents types et niveaux de qualité, et présente aussi les normes et modèles de qualité logicielles.

- **Le chapitre 4** est consacré à notre approche d'appréciation de la qualité d'un assemblage de composant en proposant :

    - Un modèle de qualité pour les composants logiciels,
    - Un processus d'évaluation de la qualité d'un assemblage de composant,
    - L'usage d'une technique de prise de décision multicritère qui permet de choisir le meilleur assemblage.

En fin nous concluons avec des perspectives.

Chapitre 1

Les composants logiciels

Introduction

En 1968, face à la crise que traversait l'industrie du logiciel, **M. McIllroy** lança un appel pour le développement massif de composants logiciels réutilisables. Selon lui, « *l'industrie du logiciel repose sur des bases fragiles et l'un des aspects de cette fragilité est l'absence d'une sous-industrie du composant logiciel*». Depuis cet appel, plusieurs concepts ont vu le jour, tels que ceux de module, de type abstrait de données, d'objet et plus récemment de composant. L'idée générale de ce nouveau paradigme de développement est de pouvoir construire des applications, comme on assemble un puzzle ou un lego.

Les pièces de ce puzzle seraient des unités logicielles déjà développées, prêtes à l'emploi et réutilisables. Cette approche rend possible une diminution drastique des délais et des couts de développement. Cela explique son succès croissant**. [GEO07]**

L'approche permet la construction d'application à partir de l'assemblage de composants, elle est fondée sur l'idée que le développement et l'assemblage peuvent être réalisés de manière totalement séparée par des acteurs différents et dans des emplacements différents.

L'approche composant est motivée par des arguments économiques, tels que la réduction du temps et du coût de développement, ou bien la spécialisation des acteurs intervenant dans le cycle de développement.

Dans ce chapitre, nous présentons, dans un premier temps, le passage du paradigme de l'objet au paradigme de composant, après nous allons nous pencher sur quelques terminologies attachées aux composants logiciels. Nous allons voir en quoi consiste un composant d'un point de vue fonctionnel et non-fonctionnel ?, ainsi que son architecture, les caractéristiques et les différents modèles de composants existent. Enfin nous donnons une comparaison des modèles de composants.

1- Des objets aux composants [RIM06]

Les années 1990 ont vu l'explosion de l'utilisation du paradigme de l'objet pour la construction de logiciels aussi bien au niveau des méthodes de conception (OMT,OOSE,HOOD,..) des langages de programmation (C++, Java), des interfaces graphiques, des bases de données orientées objet (OQL, ODMG) que des systèmes distribués (CORBA). Il est maintenant notoirement reconnu que l'objet est un paradigme incontournable pour construire du logiciel évolutif et maintenable. Néanmoins, l'objet n'est pas sans défaut et il est de plus en plus question de composants comme ceux de Microsoft, les Java Beans, les Enterprise Java Beans ou bien les composants CORBA pour ne citer que les plus connus. Si les bénéfices de l'utilisation des objets sont assez bien caractérisés dans la littérature, il faut noter que le paradigme composant reste très flou. Il est difficile de placer une frontière entre les objets et les composants : un composant n'est-il pas tout simplement un objet évolué ? Comment caractériser alors ces évolutions ? Ce paragraphe expose l'évolution conduisant les objets vers les composants en illustrant quelques apports de ces derniers.

Rappelons brièvement quelques concepts clés des objets, à savoir l'encapsulation, l'instanciation, le polymorphisme et l'héritage. Un objet est une boîte noire regroupant des traitements (ou méthodes) agissant sur des données (ou attributs). Les attributs ne sont accessibles que par des méthodes (le principe d'encapsulation). L'instanciation d'un objet est faite à partir d'un moule (souvent une classe) définissant la liste des attributs et des méthodes de ses instances. L'ensemble des méthodes accessibles depuis l'extérieur forme l'interface publique de l'objet. Des objets de classes différentes peuvent partager une interface commune (polymorphisme). L'héritage permet de classifier, spécialiser et étendre les classes et les interfaces.

Il est important de noter que les concepts précités et les constructions des langages de programmation favorisent seulement la réutilisation des moules (classes et interfaces) mais s'occupent rarement de la réutilisation des objets. Par exemple, la persistance d'un objet doit être prévue dans son implantation. Si on désire ajouter de la sécurité sur cet objet, il faut créer une nouvelle implantation qui peut réutiliser l'ancienne. Ainsi, on peut étendre les classes, par réutilisation, durant la conception, mais il est rarement possible d'étendre un objet à l'exécution. Cela provient tout simplement du fait que tous les aspects fonctionnels d'un objet sont codés « en dur » dans sa classe : il n'y a pas de séparation des aspects. Par ailleurs, une application est constituée d'un ensemble d'objets interconnectés.

Cependant, peu de langages offrent des constructions sémantiques pour exprimer cette notion de composition. Par exemple, si deux objets doivent s'autoréférences, il est nécessaire de prévoir une méthode publique affectant un attribut privé dans chacune des deux classes. De plus, il est nécessaire d'écrire quelque part dans le logiciel l'appel à ces deux méthodes d'affectation. Ici, la solution est donc une construction technique et non sémantique. Les composants ont pour objectif de proposer des solutions à la réutilisation des objets.

Du point de vue externe, un composant ressemble tout à fait à un objet, il est décrit par une ou plusieurs interfaces (figure 1.1). Par contre, du point de vue interne, un composant est constitué d'un ou de plusieurs objets. Dans le modèle COM de Microsoft, un composant peut être construit par agrégation/composition de plusieurs objets binaires préexistants.

Dans ce type d'environnement, un effort particulier est fait sur la propriété de réutilisabilité des composants logiciels. Pour comprendre cette propriété, il peut être utile de faire l'analogie avec des composants matériels. Par exemple, selon le niveau de granularité choisi, construire un ordinateur peut être une tâche aisée. C'est en particulier le cas pour les sociétés d'assemblage qui se contentent de mettre ensemble différents composants disponibles sur le marché. Il suffit de connaître le principe de fonctionnement de chaque composant et la manière dont chacun d'eux peut communiquer avec son environnement. Chaque composant peut être considéré comme une boîte noire pour laquelle il suffit de connaître les services rendus et quelques règles d'interconnexion. Les constructeurs d'application souhaitent aujourd'hui appliquer la même démarche à des composants logiciels. La programmation par composition permet de construire une application par assemblage de composants.

Les composants sont des modules logiciels de natures très diverses qu'un constructeur d'application peut composer pour construire une application.

2- Terminologie [ACC03] [YAH08]

- ***Opération :*** est une action d'un composant. Elle est invoquée par un composant client et exécutée par un composant servant.

- ***Interface :*** est un ensemble d'opérations nommées qui vont être invoquées par le client. On considère généralement qu'une opération n'appartient qu'à une seule interface. En d'autre terme, les interfaces définissent un partitionnement de l'ensemble des opérations.

- ***Service :*** est l'ensemble des opérations accessibles d'un composant.

- ***Port :*** est la réunion de plusieurs interfaces.

- ***Opération requise :*** caractérise une opération de point du vue des besoins du client. Le caractère requis indique le besoin de l'existence d'une opération selon la spécification qui lui est associée dans l'environnement du client.

- ***Opération offerte :*** caractérise une opération du point du vue de la réalisation qui est effectuée par le servant. Le composant invoqué réalise sur invocation un traitement qui est défini par la spécification associée à l'opération offerte.

- ***Une interface (respectivement un service ou un port) requise :*** est une interface (respectivement un service ou un port) qui ne comporte que des opérations requises.

- ***Une interface (respectivement un service ou un port) offerte :*** est une interface (respectivement un service ou un port) qui ne comporte que des opérations offertes.

- ***L'implantation du composant :*** est la réalisation exécutable du composant, obéissant aux règles du module de composant. Elle peut être fournie en code compilable, en forme binaire, etc...

- ***Encapsulation :*** un composant comme un objet encapsule une structure de données qui n'est accessible que via les opérations offertes. Cette structure de composant représente l'état du composant. cet état prend un ensemble de valeurs possibles qui sont les valeurs des types implantés par le composant.

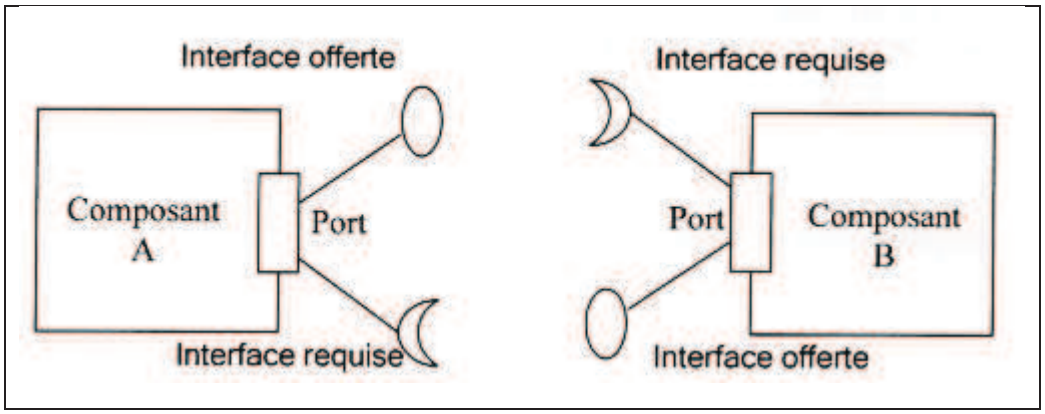

Figure 1.1 : Représentation graphique des ports présentant à la fois des interfaces requises et offertes.

3- Définition d'un composant logiciel

Parmi toutes les définitions des composants logiciels qui ont été proposées, celle de C. Szyperski, fait le plus consensus, selon laquelle. **[GEO07]**

« ***Un composant est une unité de composition qui spécifie par contractualisation ses interfaces, et qui explicite ses dépendances de contexte uniquement. Un composant logiciel peut être déployé indépendamment et est sujet à une composition par de tierces entités*** ».

La spécification UML 2.0 de l'OMG définit le terme composant comme :

« ***Une partie modulaire d'un système qui encapsule son contenu et dont la manifestation est remplaçable dans son environnement. Un composant définit son comportement en termes d'interfaces fournies et requises. En tant que tel, un composant sert comme un type, dont la conformité est définie par ces interfaces fournies et requises (incluant à la fois leur sémantique statique et dynamique)*** ».

Dans la première définition, on se concentre sur la composition, les interfaces contractuelles et le déploiement. On se place dans un contexte d'assemblage où ces entités (composants) ont déjà été développées et où l'on voudrait les assembler pour former une application. Les composants spécifient des interfaces sous forme de contrats, et peuvent être déployés séparément.

Dans la seconde définition, on met l'accent sur la modularité, la substitution et le typage. Ici, on se place dans un contexte de modélisation architecturale où l'on perçoit le système comme étant un ensemble d'unités modulaires et substituables. Chaque unité (composant) est typée par ses interfaces, qui spécifient les services requis et fournis par ce composant.

On retrouve cette notion d'interfaces fournies et requises dans toutes les technologies à base de composants telles que CCM, où l'on parle de facettes et de réceptacles ou Fractal où l'on parle d'interfaces client et serveur.

4- Architecture d'un composant

L'architecture d'un composant présentée dans la figure 1.2 spécifie les entrées et les sorties d'un composant afin de faciliter la description de son comportement (l'ensemble des services offerts).
Un composant logiciel possède principalement 03 éléments **[CER04] [CHO09]**:

- Interfaces fonctionnelles fournies et requises et propriétés de configuration.
- Interface de contrôle.
- Dépendances et propriétés de déploiement.

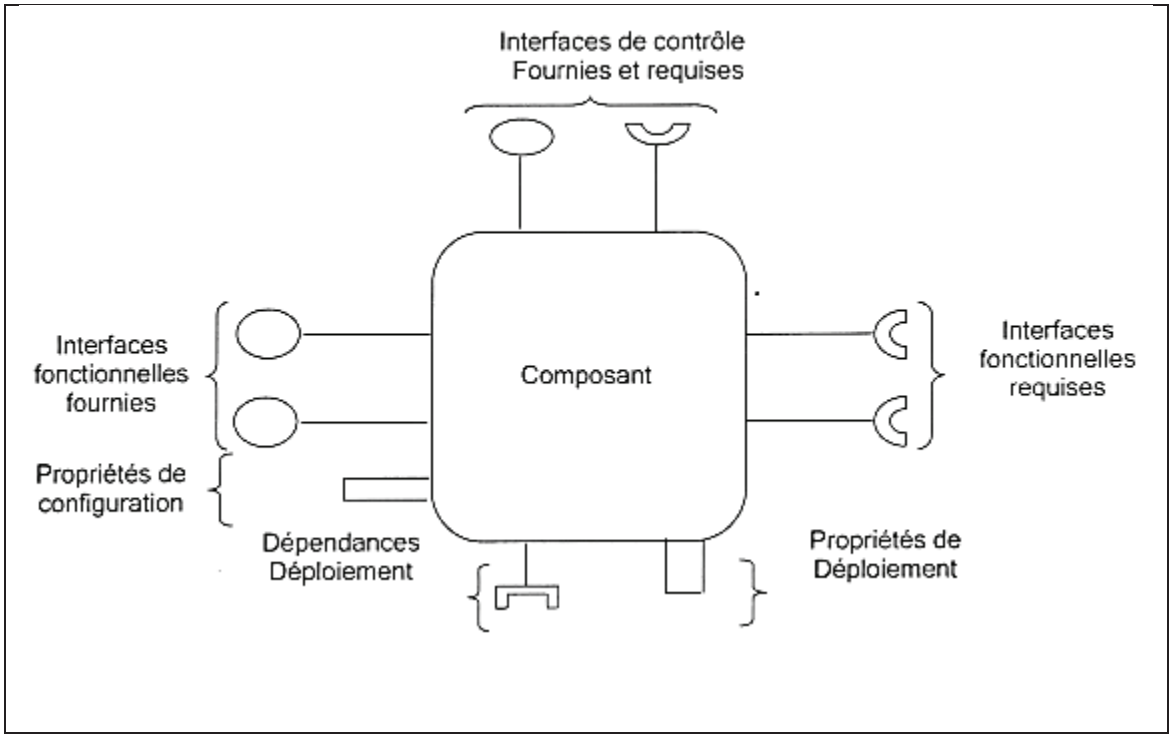

Figure 1.2 : Architecture d'un composant.

a) <u>Interfaces fonctionnelles et propriétés de configuration :</u>

Les interfaces fournies (sorties) et les interfaces requises (entrées) peuvent être des méthodes (les fonctions promises aux clients). Les interfaces fonctionnelles requises doivent être satisfaites lorsqu'une instance de composant est créée pour que celle-ci puisse être utilisée à travers les interfaces fournies.
Les propriétés de configuration permettent de configurer une instance de composant, par exemple changer le nom d'un bouton.

b) Interfaces de contrôle (fournies/requise) :

Les interfaces de contrôle présentent l'ensemble des méthodes qui permettent de gérer le cycle de vie des instances du composant pendant l'exécution. Ces méthodes sont destinées à être appelées par l'environnement d'exécution du modèle à composants.

c) Dépendance et propriétés de déploiement :

Les dépendances sont propres à une implémentation de composant. Elles doivent être satisfaites au moment du déploiement d'une classe de composant pour permettre son utilisation. Elles peuvent représenter par exemple une dépendance envers une version particulière de l'environnement d'exécution, des librairies des ressources binaires (image, son), des fichiers de configuration...

Les propriétés de déploiement, définies au niveau de l'implémentation, sont similaires aux variables de classe dans l'approche objet. Les propriétés d'implémentation sont employées pour configurer des caractéristiques communes à toutes les instances.

5- Caractéristiques d'un composant logiciel

Les caractéristiques globales d'un composant sont : **[ACC02]**

- ***L'interface d'un composant :*** est la description de l'ensemble des services offerts et requis par le composant. L'interface est un moyen d'expression des liens du composant ainsi que ses contraintes avec l'extérieur.

- ***Le type d'un composant :*** est un concept représentant l'implantation des fonctionnalités fournies par le composant. Il s'apparente à la notion d'abstraction ou de classe que l'on trouve dans le modèle orienté objet.

- ***La sémantique d'un composant :*** est exprimée en partie par son interface. Cependant, l'interface ne permet pas de préciser complètement le comportement du composant. La sémantique doit être enrichie par un modèle plus complet et plus abstrait permettant de spécifier les aspects dynamiques ainsi que les contraintes liée à l'architecture.

- ***Les contraintes d'un composant :*** définissent les limites d'utilisation d'un composant et ses dépendances inter composants. Une contrainte est une propriété qui doit être vérifiée sur un système. Si celle-ci est violée, le système est considéré comme un système incohérent et inacceptable.

- ***Les propriétés fonctionnelles et non fonctionnelles d'un composant :*** **[GEO07]**

Les propriétés fonctionnelles des interfaces prennent la forme d'opérations dont on donne la signature et parfois la sémantique. La signature d'une opération est de la forme *NomOp*=(*param***1**,*param***2**, ..., *param**n***)→*result*, où *NomOp* est le nom de l'opération, (*param*1, *param*2, ...,*paramn*) est l'ensemble de ses paramétres ou arguments, et *result* est son résultat.

Les aspects sémantiques prennent la forme de prédicats associés aux interfaces et aux opérations. Ces prédicats représentent des spécifications comportementales regroupées sous la forme d'invariants, de pré-conditions et de post-conditions. Les pré-conditions assurent que

l'opération possède certaines propriétés avant d'être exécutée, tandis que les post-conditions garantissent qu'elle en possédera d'autres après son exécution. L'invariant, quant à lui, garantit que certaines propriétés seront respectées avant et après l'exécution.

Si les propriétés fonctionnelles décrivent les aspects syntaxiques des services fournis et requis par un composant, alors les propriétés non-fonctionnelles, ou propriétés de qualité représentent la manière de rendre ou d'attendre ces services. Le standard ISO-8402 définit la qualité comme : *la totalité des caractéristiques d'un produit ou d'un service qui définissent son habilité à satisfaire des besoins établis ou implicites...* Un sous-ensemble particulier de la qualité logicielle est la qualité de service. Le standard ISO-9309 la définit comme : *un ensemble de qualités liées au comportement collectif d'un ou plusieurs objets.* Il s'agit de *l'effet combiné des performances d'un service qui déterminent le degré de satisfaction d'un utilisateur de ce service*.

- ***Les différentes abstractions d'un composant***

L'abstraction d'un composant correspond à la visibilité de son implantation. Selon **[CHO09]** Il existe 03 sortes d'abstraction :

- **Boite noire (Blackbox) :** le client ne connaît aucun détail au-delà des interfaces et de leurs spécifications.

- **Boite blanche(Whitebox) :** L'implantation d'une boite blanche est entièrement disponible et peut donc être étudiée afin d'augmenter sa compréhension. On peut trouver dans la littérature, le terme de boite transparente (Glassbox). Quand la distinction est faite cela signifie que la boite blanche permet la manipulation de l'implantation alors que la boite transparente permet l'étude de l'implantation

- **Boite grise (Graybox) :** Seule une partie contrôlée de l'implantation est visible.

6- Principes des composants logiciels [HPR11]

Nous voyons les composants comme des entités autonomes d'encapsulation (boîtes noires) qui fournissent et requièrent des fonctionnalités à leur environnement en utilisant des interfaces ouvertes et bien définies (figure 1.3(a)). Les interfaces définissent la syntaxe et la sémantique des fonctions qu'elles représentent. En effet, elles déterminent le contrat entre un composant et son environnement. Étant donné que l'environnement d'un composant est structuré en termes de composants, les interfaces définissent les contrats entre composants.

Afin de construire une application à base de composants, les composants sont interconnectés à l'aide de leurs interfaces (figure 1.3(b)). Certains composants, ainsi que l'application sont donc assemblés par composition de composants, (figure 1.3(c)).

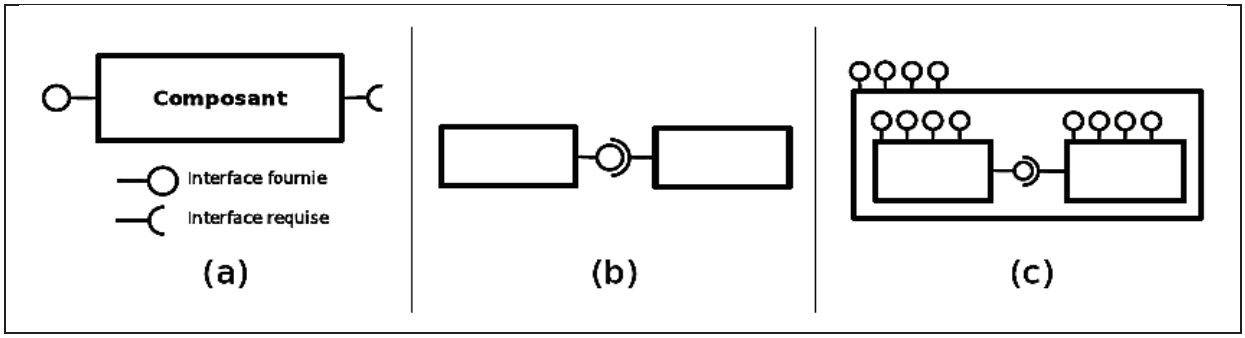

Figure 1.3 : Interfaces, connexions et composition des composants.

Par rapport à l'approche orientée objets, les composants apportent plusieurs améliorations. Ils permettent de définir des besoins non liés au code métier (au niveau de la sécurité, la fiabilité, la gestion du cycle de vie, etc.) et explicitent mieux l'architecture des applications. L'approche par composants permet un déploiement sur une plate-forme complexe, de manière relativement simple et facilement modifiable.

Nous identifions quatre étapes nécessaires pour la mise en œuvre et pour la gestion du logiciel à base de composants logiciels. Il s'agit de la définition, l'implémentation, le déploiement et l'exécution des composants.

6.1- La définition

Cette étape consiste à décrire l'architecture du logiciel en termes de composants et des liens établis entre eux. Pour cela, plusieurs modèles à composants, comme par exemple Fractal et CCM, utilisent un langage de description d'architecture (ADL) et un langage de description d'interfaces (IDL). L'ADL permet de décrire explicitement la structure de l'application en termes de composants. L'IDL permet de définir les fonctions fournies pour chaque interface, et donc les services que les composants offrent les uns aux autres. Parmi les interfaces fournies par les composants, les modèles définissent souvent des interfaces d'introspection et de contrôle. Les interfaces d'introspection révèlent des informations internes au composant comme sa structure, les fonctionnalités offertes, son état, etc. Les interfaces de contrôle varient d'un modèle à un autre et peuvent concerner la gestion du cycle de vie ou la configuration (valeurs de paramètres, choix de services systèmes utilisés, etc.) des composants.

6.2- L'implémentation

L'implémentation concerne le développement du code métier des composants, la configuration des services système utilisés, ainsi que l'assemblage des composants dans une application. L'assemblage peut être fait selon un modèle de composition plat ou hiérarchique. Si le modèle est plat, le logiciel se base sur des composants primitifs, qui sont des éléments indivisibles de composition et déploiement. Si l'organisation suit des hiérarchies nous pouvons, comme dans le modèle Fractal, définir des composants composés d'autres composants, appelés composites. Ces derniers peuvent être décomposées ou réutilisés intégralement.

Le code métier est pris en charge par un développeur de l'application à composants. Les services utilisés sont typiquement fournis dans des canevas et leur utilisation est faite à l'aide d'une chaîne d'outils de génération du code.

6.3- Le déploiement

Le déploiement d'une application comprend l'installation de ses composants, la configuration par rapport au contexte d'exécution et le démarrage. Les environnements à composants, par exemple OpenCCM ou EJB, fournissent explicitement des fonctions de déploiement.

6.4- L'exécution

L'exécution des composants est possible grâce à des supports d'exécution, qui permettent le lancement des composants et fournissent des mécanismes de contrôle pour la gestion de leur exécution. Le contrôle de l'exécution est aussi connu comme la gestion du cycle de vie, qui consiste à contrôler les phases comportementales du composant, comme le démarrage et l'arrêt.
Les supports d'exécution servent à la création d'instances du composant, autrement dit, des entités d'exécution concrète créées à partir de types de composants. L'exécution est possible grâce à des environnements, qui fournissent les services nécessaires pour le lancement sur une plate-forme donnée. Parmi les services, nous citons le nommage, la communication entre composants, la sécurité, etc.

Une fois que les composants ont été assemblés visuellement, il est intéressant d'enregistrer cette configuration dans un package pour une utilisation future. Les packages de composants contiennent alors le code binaire des classes des objets formant les composants, les valeurs initiales des propriétés des composants et la description de la composition de ceux-ci. En fait, on peut considérer qu'un package est un moule décrivant l'architecture d'une application permettant son instanciation/installation. On trouve cette fonctionnalité dans les composants Java Beans, Enterprise Java Beans et CORBA. Avec ces derniers, un même package peut contenir les binaires pour plusieurs architectures matérielles. Le déploiement d'une application est donc grandement simplifié car il suffit de distribuer le package aux futurs utilisateurs. **[RIM06]**

Pour assurer les fonctions des composants, ces derniers ont besoin d'un support logiciel. On distingue généralement *la structure d'accueil* et *le conteneur*. **[YAH08]**

La structure d'accueil fournit tous les services systèmes dont peuvent avoir besoin les conteneurs : protocole de communication, service de persistance, gestion des ressources, etc...

Un conteneur est la partie système correspondant à un composant. Le conteneur encapsule le composant, au sens où toutes les interactions du composant avec l'extérieur doivent normalement passer par le conteneur. Grâce à cette interposition (délégation), le conteneur peut gérer les propriétés non-fonctionnelles du composant.

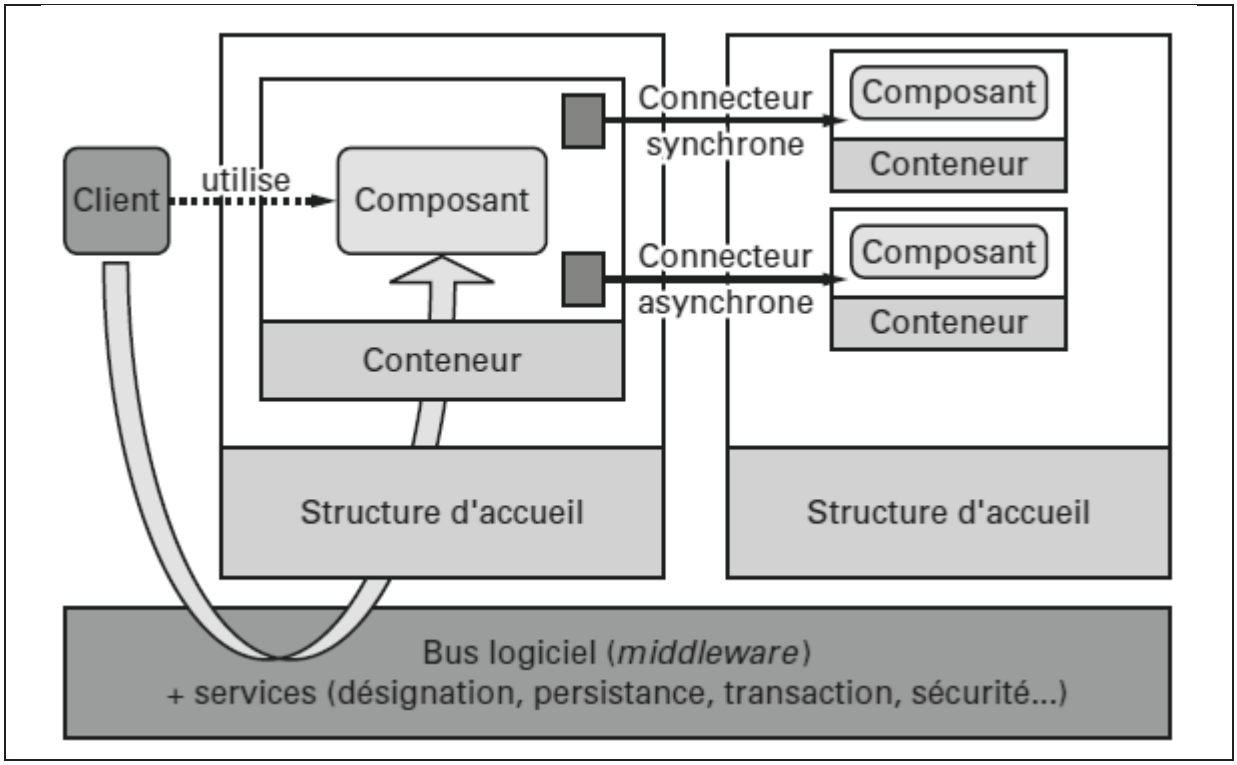

Figure 1.4 : Conteneur et structure d'accueil. [RIM06]

7- Étude de quelques modèles de composants

Le modèle de composants permet de définir les entités et leur mode d'interaction et de composition. Cette section présente quelques modèles de composants utilisés aujourd'hui dans l'industrie pour construire des applications.

Nous avons fait le choix de présenter de manière succincte les modèles de composants suivants :

1. **Java Beans**
2. **EJB (**Enterprise Java Beans) de Sun Microsystem,
3. **COM+** de Microsoft,
4. **CCM** d'OMG (Objet Management Group).

Cette présentation n'est pas exhaustive : elle omet de nombreux autres modèles de composants. Cependant, elle offre un large tour d'horizon des fonctionnalités qui sont ou devraient être offertes par tout modèle de composants.

7.1- Java Beans

Les Java Beans sont des composants Java persistants construits selon un gabarit de construction spécifique afin de garantir des propriétés de persistance, d'introspection, de configuration et d'assemblage pour construire une application ou une partie d'application Java. **[RIM06]**

Les Java Beans peuvent être utilisés dans n'importe quel environnement Java (par exemple, Symantec's Visual Cafe, IBM's Visual Age, SunSoft's Java Workshop ou Borland's Jbuilder). Ces environnements permettent la construction d'applications par drag and drop (copier-coller) de composants à partir d'une palette que l'on relie graphiquement. Un éditeur de propriétés permet de spécialiser chacun des composants.

Un composant Java Beans (ou « bean ») est une classe Java qui hérite de la spécification associée à l'architecture Java Beans. C'est à la charge du programmeur de décider ce qui doit être une classe Java standard et ce qui doit être un bean. Actuellement, la plupart des beans concernent la construction de l'interface graphique de l'application et le programmeur peut par exemple, considérer que chaque élément de cette interface graphique est mis en œuvre par un bean ainsi que certains éléments de l'application. Il n'est pas nécessaire de transformer toutes les classes de l'application sous la forme de beans, mais si la réponse à une des questions suivantes est oui, il peut être intéressant de construire cette classe sous la forme d'un bean : **[RIM06]**

- ce module logiciel peut-il être utilisé dans un autre contexte ?
- d'autres utilisateurs peuvent-ils être intéressés par ce module ?
- ce module logiciel peut-il être aisément configuré ?
- son exécution peut-il être paramétrée ?
- le principe de fonctionnement du module et les services qu'il rend sont-ils facilement compréhensibles ?
- le module contient-il toutes les données nécessaires pour fonctionner par lui-même ?
- le module possède-t-il de bonnes propriétés d'encapsulation ?

Chaque bean possède les caractéristiques suivantes : **[RIM06]**

a) Persistance

Chaque bean est persistant et implémente pour cela l'interface sérializable. Il peut être intéressant de spécialiser le comportement du bean afin de contrôler la partie de l'état qui sera sauvegardée. En particulier, Java ne permet pas de rendre persistants des flots d'exécution ; il est donc inutile de sauvegarder les données liées à une exécution particulière. Il suffit pour cela de marquer les variables qu'il n'est pas souhaitable de sauvegarder avec l'attribut *transient*.

```
class MyBeanClass implements Serializable{
                    transient Thread t; // les donnes associes
                  //au thread ne sont pas persistantes.
                 public MyBeanClass() {} // constructeur sans argument
                              }
```

Les outils de manipulation des beans créant une instance de ceux-ci par appel du constructeur Class.newInstance() sans argument, il est nécessaire de placer ce constructeur dans la classe définissant le bean. Il est possible de mettre d'autres constructeurs qui pourront être appelés explicitement par d'autres beans ou par d'autres objets Java.

b) Manipulation visuelle

Un bean peut être configuré à l'aide d'outils génériques. La configuration porte sur les propriétés du bean et elle peut être mise en œuvre, par exemple, par un éditeur de propriété graphique. La définition des propriétés est basée sur l'utilisation de gabarits de conception (design patterns) portant sur des conventions de nom pour les méthodes définies dans la classe du bean. La syntaxe générique est getProperty() et setProperty() pour lire ou modifier une propriété de nom Property. Par exemple, un bean qui possède une méthode déclarée sous la forme DataType getProperty() possède une propriété de nom Property et de type DataType qui peut être lue. De manière symétrique, un bean qui possède la méthode void setProperty(DataType x) possède une propriété de nom Property et de type DataType qui peut être modifiée. Le nom de la propriété, le nom de la variable qui contient la valeur de la propriété et le nom de la propriété utilisé pour les méthodes get et set doivent être identiques.

c) Introspection

Les beans supportent le mécanisme d'introspection qui permet à une application cliente de découvrir le fonctionnement du bean. Le mécanisme d'introspection repose sur le respect de gabarit de conception (convention de nommage et design pattern) et utilise de manière transparente l'API Reflection du langage Java. Une autre manière de décrire le comportement d'un bean est d'associer une classe d'information. Si la classe d'information lève une exception, l'application cliente utilise le mécanisme d'introspection du bean pour rechercher la propriété qui a provoqué cette exception.

d) Communication par événements

Les beans communiquent entre eux par événements. Ce mode de communication repose sur le respect de gabarits de conception qui permettent d'enregistrer la référence des beans intéressés par les différents événements qu'un bean peut émettre.

e) Configuration

Un bean fournit aussi un mécanisme permettant de configurer certaines de ses propriétés ou une partie de son comportement. Un environnement visuel de développement permet cette configuration par l'utilisation d'une fenêtre permettant l'édition des propriétés. Cette étape de configuration peut être difficile et fastidieuse, en particulier dans le cadre de beans complexes. Pour cela, il est aussi possible de construire sur demande une feuille d'édition de propriétés spécifiques en utilisant une classe construite qui sera activée à la place du mécanisme par défaut (Customizer class).

f) Packaging

Une fois le bean développé, celui-ci doit être sauvegardé dans une archive Java. La plupart des environnements de conception génèrent automatiquement ces fichiers.

7.2- Entreprise JavaBeans : modèle de composants serveur pour la plate-forme Java:

La spécification EJB définit un modèle d'architecture à base de composants qui est destiné à faciliter la mise en œuvre d'applications d'entreprise en utilisant le langage Java. Le concept de base de cette architecture est celui de composants EJB. Il s'agit de composants logiciels

transactionnels applicatifs (qui implémentent une logique métier) réutilisables et configurables. **[RIM06]**

a) Le modèle

Le modèle de composant EJB étend le modèle de composant JavaBeans pour supporter les composants serveurs qui sont des morceaux de codes prépackagés et réutilisables. Ils peuvent être combinés avec d'autres composants pour créer une application spécifique. **[HME09]**

Le serveur EJB gère implicitement un certain nombre de services middleware pour les composants EJB. Les développeurs d'entreprise Beans peuvent ainsi se concentrer sur l'écriture du logique métier plutôt que sur les complexes aspects middleware.

- ***Serveur EJB***

Le serveur fournit un environnement qui supporte l'exécution d'applications développées à l'aide de la technologie EJB.

- ***Conteneur EJB***

Le serveur EJB doit fournir un ou plusieurs conteneurs EJB qui fournissent eux-mêmes l'environnement d'exécution nécessaire aux entreprises beans. Pour chaque entreprise Bean, le conteneur gère :

- L'enregistrement de l'objet, en fournissant une interface distante pour l'objet.
- La création et la destruction des interfaces de l'objet.
- La sécurité de l'objet.
- La coordination des transactions distribuées.

Une entreprise bean est assignée à un seul conteneur EJB, mais ce conteneur peut contenir plusieurs composants EJB.

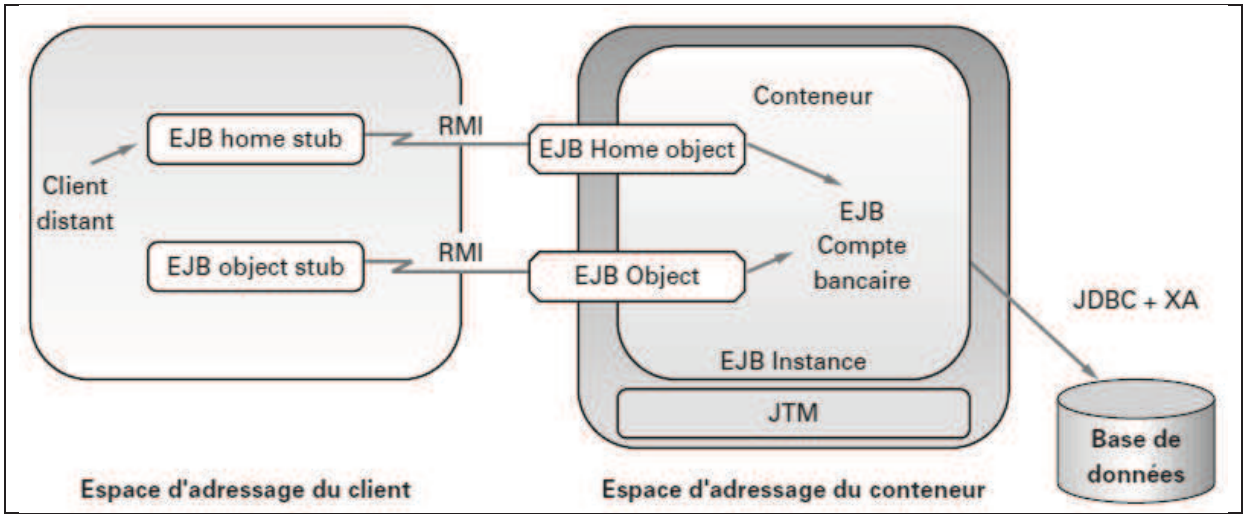

Figure 1.5 : Architecture de la plate-forme EJB. [RIM06]

b) Topologie des composants EJB

La technologie EJB gère à la fois les objets persistants, appelés *EJB entity*, et les objets non persistants, appelés *EJB session*.

Un EJB entity permet de représenter une donnée persistante ayant une existence au niveau d'un support de stockage telle une base de données relationnelle. Une clé primaire identifie chaque instance de l'EJB entity. Un EJB entity peut gérer sa propre persistance ou déléguer les services de persistance à son conteneur EJB.

Un EJB session créé par un client et n'existe que le temps d'une session client-serveur simple. Un EJB session exécute des opérations pour le client telles que l'accès à une base de données (via le service déjà existant dans le serveur EJB). Le conteneur gère la suppression éventuelle d'un EJB session de la mémoire.

c) Environnement d'exécution EJB

Comme il est illustré dans la figure 1.6, dans l'environnement d'exécution des EJBs on trouve : **[YAH08]**

- **Client EJB (client Distant) :** peuvent appeler les méthodes définit dans l'interface distante (Remote Interface) ; cette interface définit toutes les méthodes fonctionnelles qu'un client peut invoquer sur le composant (vue cliente de l'EJB). L'interface maison (Home Interface) définit les méthodes qu'un client peut invoquer sur le composant pour créer, supprimer ou rechercher des instances de l'EJB sur le serveur.

- **Serveur EJB :** est mis en œuvre pour héberger les conteneurs EJB.

- **Conteneur EJB :** assure les services requis par un (ou plusieurs) Bean EJB, c'est l'environnement dans lequel un ou plusieurs composants EJB s'exécutent. 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 modèle EJB représente un grand pas vers un modèle opérationnel de composants serveurs. Mais il reste une solution propriétaire de Sun, destinée uniquement à une utilisation en Java (mono-langage).

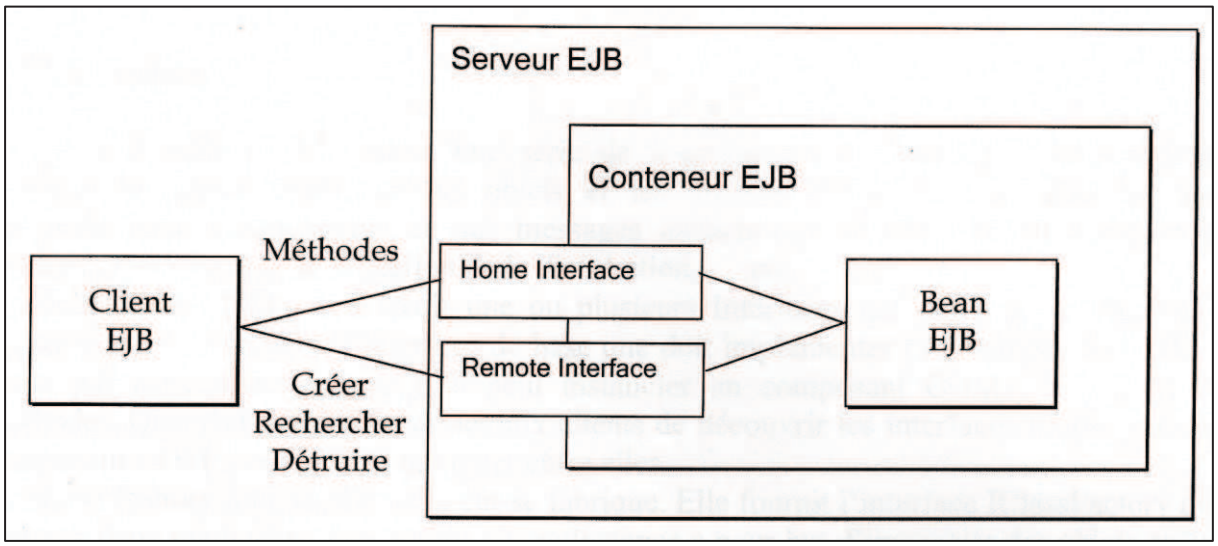

Figure 1.6 : L'environnement d'exécution des composants EJB.

7.3- le modèle COM [HME09]

Au début des années 90, le modèle COM s'est imposé comme une technologie Windows, importante et évolutive. Tout objet COM peut être distribué. En d'autres termes, 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.

DCOM permet la communication entre objets situés sur des machines différentes. Sous Windows 2000, DCOM change de nom pour devenir COM+. On peut dire alors que COM+ est l'évolution de COM. Microsoft a développé un environnement de création d'application Web nommé « .NET ».

La programmation d'un composant nécessite un compilateur (C, C++, Visual Basic,..) qui produit un binaire conforme à la norme COM.

a) Le modèle

Un objet COM doit implémenter l'interface *IUnknown* (qui permet de gérer le cycle de vie des objets et d'obtenir les autres interfaces gérées par le composant), et toute interface doit proposer la méthode *QueryInterface*.

Du fait qu'il ne définisse que la partie interface, COM n'a pas de mécanisme de réutilisation d'implémentation.

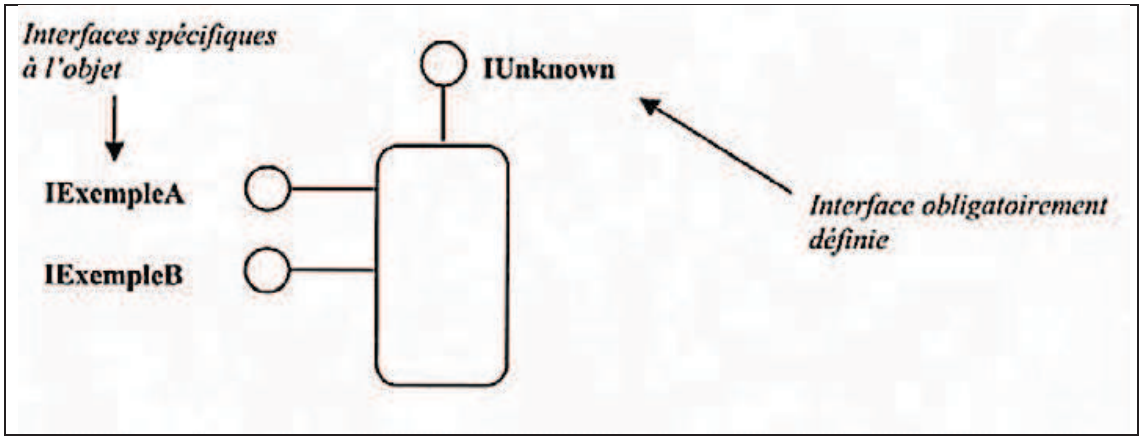

Figure 1.7 : Objet COM.

COM définit deux méthodes de composition :

1. ***Containment :***

Un objet reçoit les requêtes d'un client (le conteneur) et les redirige vers ses objets fils. Les objets contenus ne sont accédés que par le conteneur.

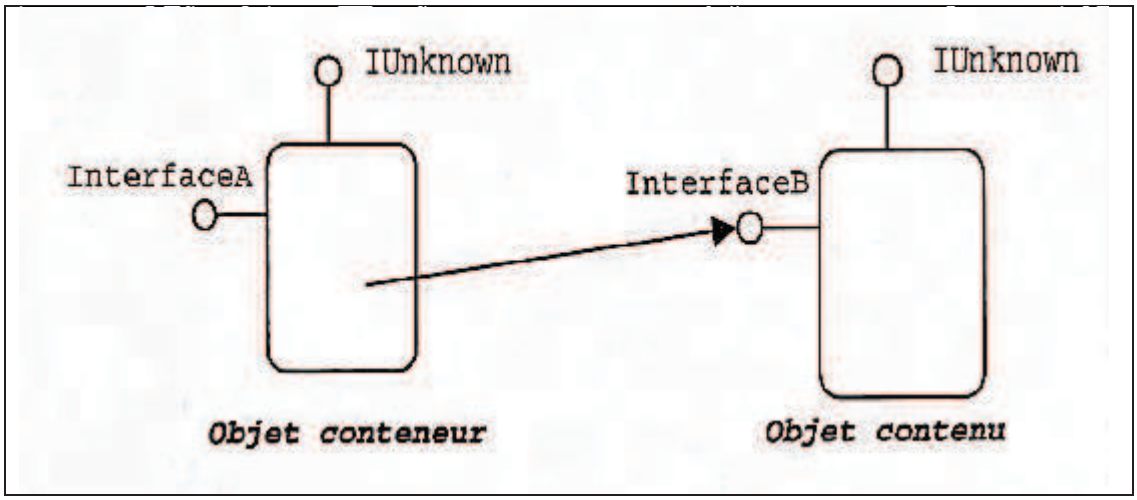

Figure 1.8 : Containment.

2. ***Aggrégation :***

Les interfaces des objets fils sont directement exposées aux clients. L'interface « IUknown » d'un fils pointe sur l'interface« IUknown » du conteneur pour que cette construction apparaisse comme un objet unique.

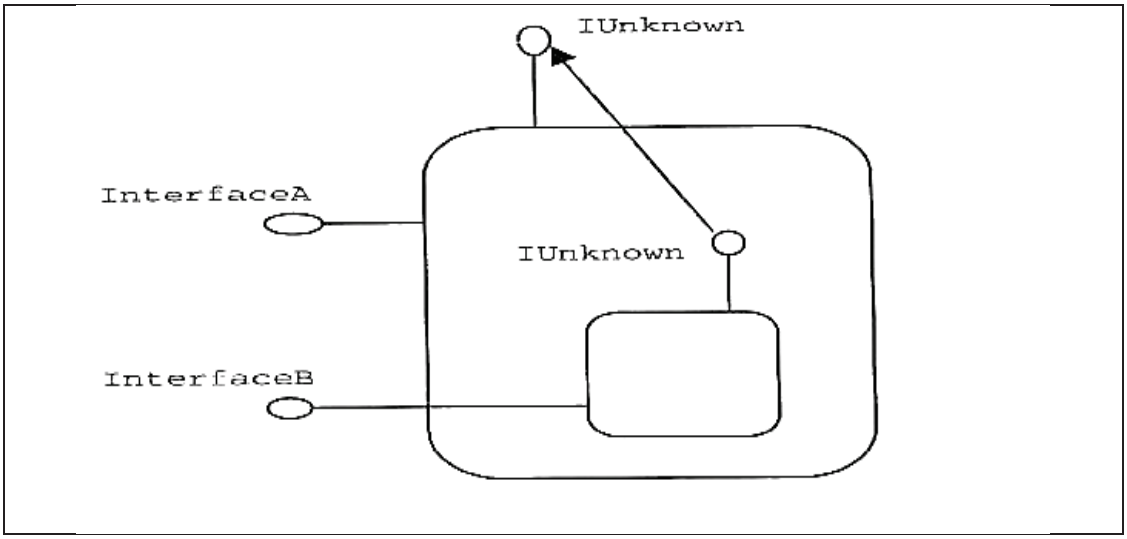

Figure 1.9 : Agrégation.

b) Environnement d'exécution .Net

L'environnement .NET peut être défini de la façon suivante : il s'agit d'un ensemble de services communs, utilisable depuis plusieurs langages. **[YAH08]**

La plate-forme .NET support nativement un grand nombre de langages de programmation grâce au CLR (Common Langage Runtime), qui permet de compiler le code source de l'application en un langage intermédiaire, baptisé MSIL (MicroSoft Intermediate Lauguage), ce dernier permet de représenter dans un même formalisme des applications programmées dans différents langages tels que C#, C++, Eiffel, VB,…

Au-dessus du CLR se trouve l'ensemble des librairies de classes, couche sur laquelle repose la gestion des données.

Au sommet de cette construction, se trouvent deux blocs, celui des windows Forms, ensemble de classe permettant la conception d'IHM (Interface Homme Machine) Windows et celui ASP.NET (Web Services+Web Forms). ASP.NET est un ensemble de classes permettant à la fois la conception des sites dynamiques, la création d'IHM pour le Web et la conception des services Web.

COM+ est un modèle multi-langages (supporte 27 langages, le langage Java est exclu). COM+ oblige de traduire tous les programmes en MSIL, ce qui va doubler le travail des programmeurs.

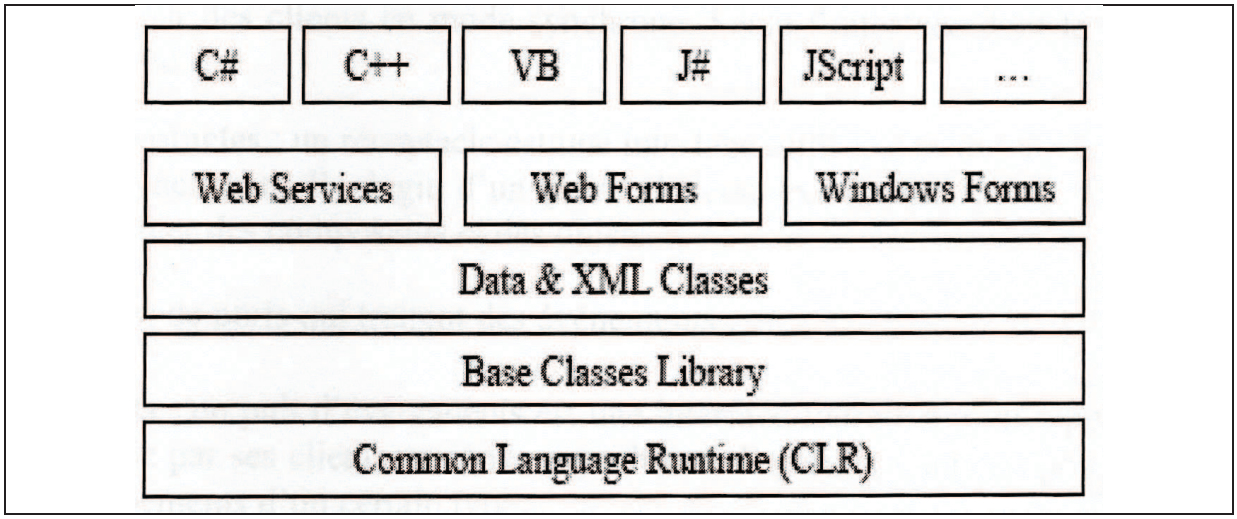

Figure 1.10 : Architecture de l'environnement .Net .

7.4- Le modèle de composant CCM

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 supporte l'intégration, à travers le web, entre des composants écrits dans 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.

a) Le modèle

Le modèle CCM est un ensemble de modèle 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 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 Définition Language).
2. Le modèle d'implantation définit la façon d'implanter un composant à l'aide de 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.

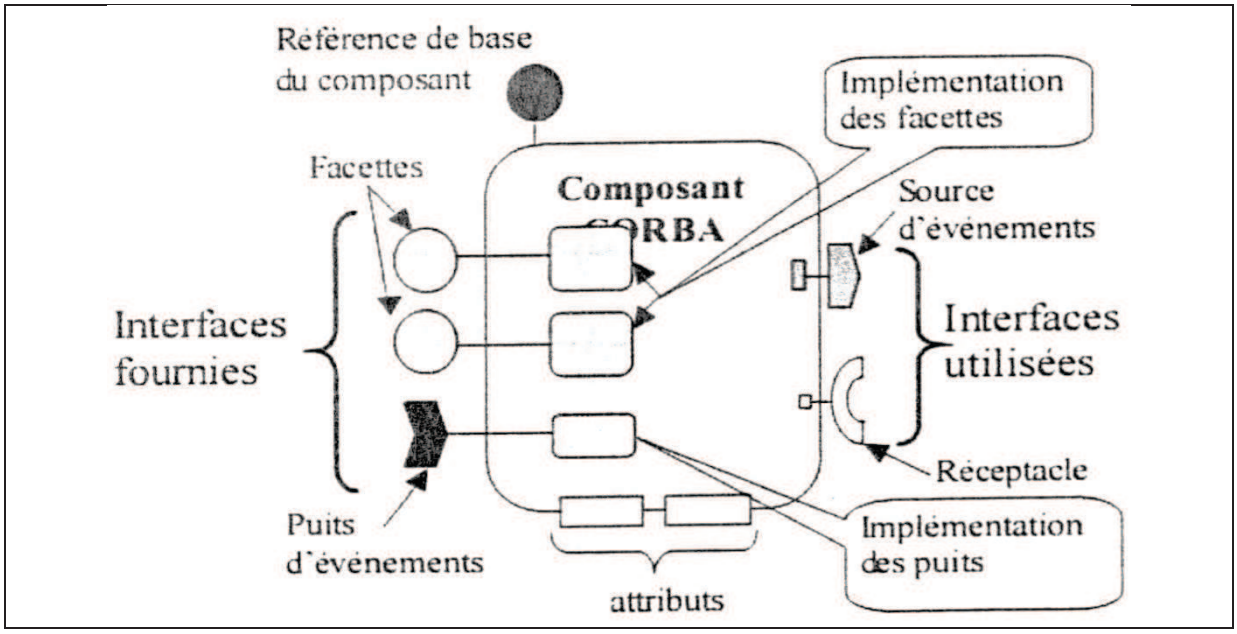

Figure 1.11 : Le modèle abstrait de composants CCM. [HME08]

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 composant. Le langage OMG IDL est étendu pour exprimer les interconnections d'un composant. Il est baptisé IDL3 en rapport avec la version 3.0 de la spécification CORBA.

Un composant CCM peut supporter une interface grâce au mécanisme classique d'héritage des objets CORBA. Cela permet de transformer facilement un objet CORBA déjà existant en composant.

Comme le présente la figure 1.11, 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 : fournis et requis.

Comme en EJB, il existe dans CCM plusieurs types de composants :

1. Les composants CCM de type Service : sont l'équivalent des Stateless Session Beans des EJB.
2. les composants CCM de type Session : sont l'équivalent des Statefull Session Bean des EJB.

3. Les composants CCM de type Entité : sont l'équivalent des Entity Beans des EJB.

4. Les composants CCM de type Processus : est un composant persistant. Le choix de l'instance du composant n'est cependant pas laissé au libre arbitre du client. Chaque client à sa propre instance qu'il utilisera toujours. En résumé, le client n'a pas le choix : c'est le middleware qui lui attribue « à vie » une instance du composant.

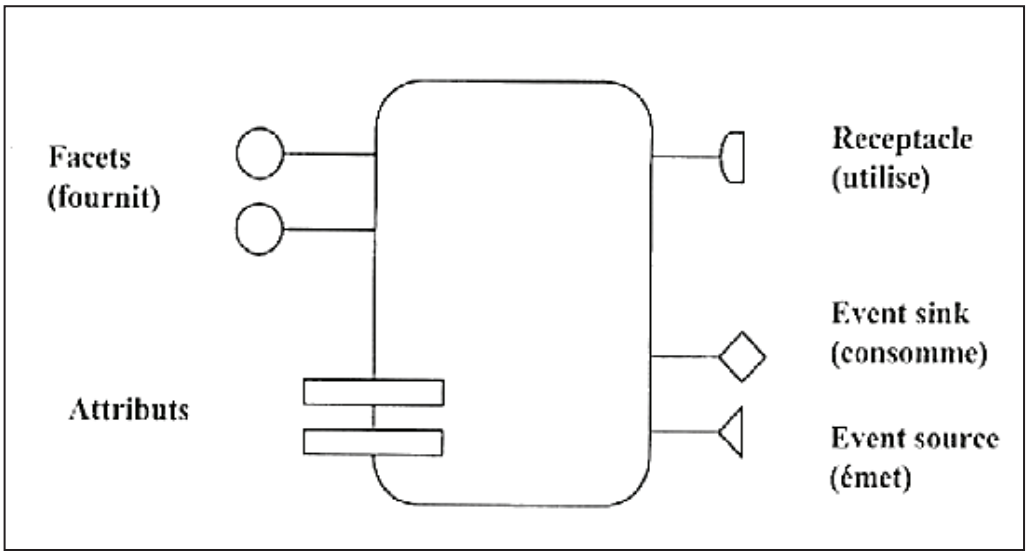

Figure 1.12 : Composant CORBA.

b) Environnement d'exécution CCM

L'architecture du modèle CCM, présentée dans la figure 1.11, 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.… **[YAH08]**

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

Les composants CCM sont des composants multi-interfaces. Ils permettent d'être indépendants de la plate-forme et du langage de programmation. Cependant, ce modèle ne permet pas de définir des composants composites (assemblage de plusieurs composants pour construire un seul composant).

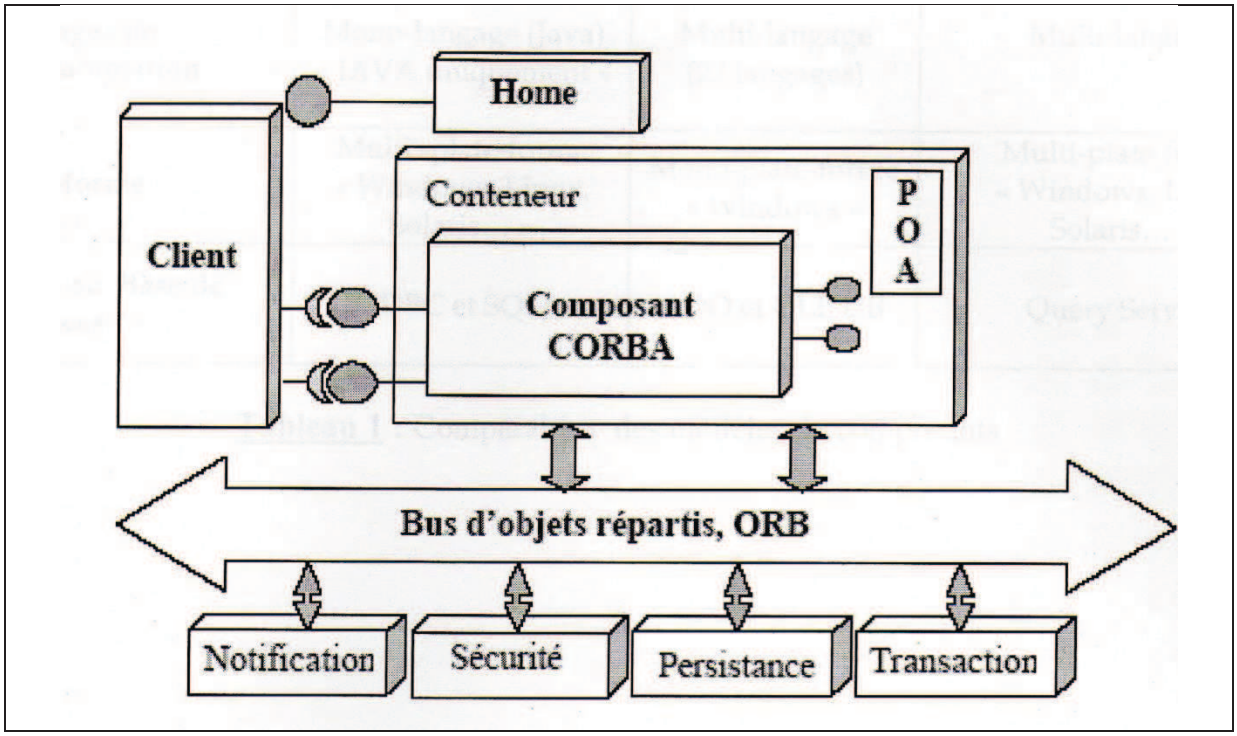

Figure 1.13 : L'environnement d'exécution des composants CCM.

8- Comparaison et évaluation des modèles de composants existants

La comparaison entre ces différents modèles peut être faite à deux niveaux. **[RIM06]**

1. JavaBeans et COM/Active X ont les mêmes objectifs : faciliter la communication entre des composants logiciels placés dans un environnement d'exécution spécifique, le conteneur. Les modèles sont relativement proches et diffèrent presque uniquement par leur environnement :

- JavaBeans est un environnement Java, permettant d'écrire des composants logiciels s'exécutant sur n'importe quelle station. L'accent est mis sur les qualités de portabilité plutôt que d'efficacité ;

- COM/Active X est un environnement pour plate-forme Windows, où l'intégration se fait au niveau du code binaire lui-même. L'effort a porté sur la possibilité de programmer dans de très nombreux langages des composants COM, de les intégrer au niveau du code objet pour construire de nouvelles applications efficaces. La richesse de l'ensemble repose sur la base installée.

2. Enterprise Java Beans, les composants CORBA et ceux de Microsoft ont des objectifs relativement similaires : permettre la construction d'applications réparties par assemblage de composants logiciels. On retrouve les mêmes aspects différenciateurs que précédemment : EJB est une solution Java et favorise donc la portabilité du composant au détriment de l'efficacité, COM+ est une solution propriétaire pour celui qui choisit de travailler uniquement dans un environnement Windows et CORBA

favorise la possibilité d'écrire des composants dans différents langages de programmation et d'interconnecter des composants qui s'exécutent sur n'importe quel type de plates-formes.

Sous cette apparente convergence, d'importantes différences existent néanmoins entre la solution Microsoft, la solution EJB et la solution CORBA.

- Dans la solution Microsoft, un constructeur propose seul un ensemble de solutions et décrit comment les utiliser. La portabilité est assurée par l'unicité du matériel, du système d'exploitation et l'intégration peut se faire au niveau du binaire.

- Dans la solution EJB, un constructeur propose une interface ; d'autres constructeurs réalisent des implémentations de cette interface et proposent éventuellement des extensions au modèle d'origine. La portabilité est assurée par l'unicité du langage de programmation et par la spécification précise des différentes interfaces (interface interne à la plate-forme EJB mais aussi interface propre à la machine virtuelle Java dans le cas présent).

- Dans la solution CORBA, on a affaire à un processus ouvert ou chacun peut proposer des interfaces, les faire adopter et mettre en œuvre des interfaces précédemment adoptées. La portabilité des solutions n'est pas toujours garantie, mais l'interopérabilité de l'ensemble par un bus CORBA est possible. À chaque instant, de nouveaux services peuvent être ajoutés, remplacés, utilisés. Le processus d'évolution est ouvert.

	Composant / Critères 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)
	Integration	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
	Accès aux bases de données	JDBC et SQLJ	ADO et OLE DB	Query Service

Tableau 1.1 : Comparaison des modèles de composants.

Conclusion

Dans ce chapitre, nous sommes revenus sur le passage de l'objet au composant après quelques définitions concernant les composants logiciels puis nous avons spécifié par la suite les différentes notions de base sur les composants logiciels. Nous avons en particulier, présenté les différents modèles de composants logiciels existants et mis en évidence, leurs similarités, leurs différences, leurs avantages et leurs limites.

Nous concluons que l'entité composant est apparue en réponse aux limites de l'approche de conception par objet. Le composant logiciel 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 composant. 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. Dans le chapitre suivant nous nous étalerons sur l'assemblage de composants et la composition d'application.

Chapitre 2

L’application composée

Introduction

Le génie logiciel à base de composants a pour objectif le développement de gros logiciels par l'intégration de composants existants. Le but de *l'assemblage* est d'obtenir une application, dite *application composée*, cohérente à partir d'un ensemble de composants logiciels, Figure 2.1.

Le développement orienté composants ambitionne d'améliorer la réutilisation de composants logiciels en palliant aux limites de l'orienté objets. L'approche orienté objets qui avait permis de maîtriser la complexité de systèmes de taille moyenne a désormais trouvé ses limites. Bien que l'architecture de ces systèmes puisse être en partie maîtrisée lors de la phase de conception à l'aide de diagrammes UML, la compréhension globale de ces systèmes ainsi que leur maintenance devient quasiment impossible une fois que le système a été codé, la cohérence entre le code et la spécification n'étant quasiment plus respectée dans la majorité des cas. **[DES08]**

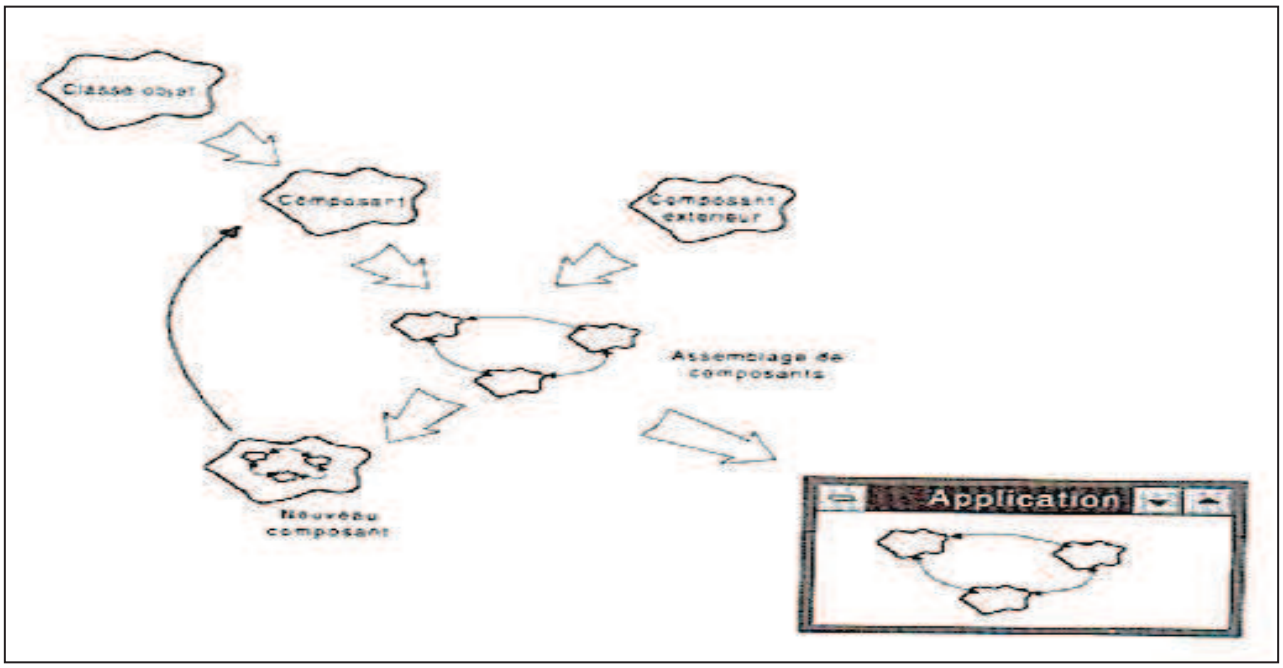

Figure 2.1 : Processus de production et d'utilisation de composants.

Beaucoup de problèmes à résoudre pour la phase d'assemblage, tels que : comment formaliser les connexions entre les composants ? Comment décrire un système dans son ensemble ? Plusieurs travaux de recherche notamment sur les ADL (Application Definition Language) tentent de répondre à ces questions. Cependant, ils imposent l'utilisation de leur propre langage. Si on veut réellement profiter de la grande promesse des composants (la réutilisation), il faut choisir les composants qui répondent à nos besoins et de savoir s'ils sont *compatibles*. La description d'un composant est le plus souvent résumée à la liste de ses interfaces, ce qui est insuffisant pour comprendre le composant. Il devient donc indispensable de développer la notion de *contrat*. **[YAH08]**

Dans ce chapitre, nous présentons, dans un premier temps, la notion de contrat des composants, après nous revenons sur la vue générale d'un système construit par assemblage

des composants, ainsi que sur les différentes vues de l'architecture. Nous voyons les différentes phases de cycle de développement d'une application composée, et aussi les approches d'assemblages, la définition et la vérification de l'assemblage correct.

1- La notion de contrat [ACC01] [YAH08]

La spécification d'un composant se traduit par un ensemble de contrats qui définissent le plus précisément possible ses exigences et ses aptitudes pour pouvoir l'assembler correctement. La notion de contrat est facilement appréhendée par tous les acteurs du développement logiciel, quelque soit leur niveau de technicité sachant que.

- ***Un service :*** est une opération qui va être appelée par un composant et exécutée par un autre.

- ***Un client :*** est un composant souhaitant utiliser un service d'un autre composant.

- ***Un fournisseur :*** est le composant exécutant un service demandé par un autre composant.

Un contrat spécifie les droits et les obligations entre un client et un fournisseur de service. Les contrats déterminent les relations entre les différents acteurs, on peut distinguer deux formes d'obligations :

- ***Les obligations contractuelles :*** sont définies par le contrat. Elles permettent d'inclure les contraintes entre objet. Il peut y avoir incompatibilité entre l'objet et certaines variables.

- ***Les obligations causales :*** forcent les participants à effectuer une séquence ordonnée d'actions. Grâce à cette deuxième famille d'obligations, les contrats capturent les dépendances comportementales entre les objets.

Il existe plusieurs niveaux de contrat dans le monde des composants logiciels :

a) ***Contrat syntaxique :*** comporte les opérations que le composant peut exécuter, les paramètres d'entrées et de sorties qu'il doit avoir et les exceptions qui pourraient survenir lors de l'opération. Il permet d'assurer que l'assemblage structurel se déroulera bien c'est-à-dire de vérifier la conformité entre les interfaces fournies et requises des composants.

b) ***Contrats comportementaux*** : ce niveau décrit pour chaque service offert les conditions d'utilisation et le résultat attendu. Les contrats syntaxiques n'apportent aucune information sur les fonctionnalités réelles. A ce niveau le comportement d'une opération est spécifié en définissant des assertions logiques qui sont les pré et post-conditions de chaque service offert.

c) ***Contrats de synchronisation :*** garantit qu'en cas d'utilisation concurrent le fonctionnement sera correct. Les contrats comportementaux supposent que les services sont

atomiques ou exécutés dans des transactions, ce qui n'est pas toujours le cas. Le niveau suivant de contrat consiste donc à spécifier le comportement global des objets en termes de synchronisation entre les appels de méthodes.

d) Contrats de qualité de service : ces contrats doivent permettre de qualifier le comportement attendu et d'offrir la possibilité de négocier ce comportement. Ils intègrent la réservation de ressources (CPU, bande passante, etc) ou des contraintes non fonctionnelles d'exécution. D'une façon générale, ils permettent la précision des conditions quantitatives d'utilisation du composant en termes de qualité de service. On trouve de nombreuses descriptions de contrat de QoS pour la fiabilité, la disponibilité, la performance, la sécurité ou encore sur les aspects temporels.

2- Systèmes construits par assemblage de composants

Depuis les années 2000, les approches à composants ont pris une part centrale dans le génie logiciel. CBSE (Component-Based Software Engineering) fait référence dans le domaine de la construction de systèmes logiciels et propose de construire des systèmes logiciels par assemblage de composants réutilisables afin d'assurer la sûreté, la qualité et la maintenance du système tout en minimisant l'investissement en ressources matérielles, en temps et en argent. Il est intéressant de noter que l'idée selon laquelle un système peut être construit à partir de composants sur étagère n'est pas nouvelle et qu'elle a été proposée par McIlroy en 1968. Les composants sont généralement décrits par un ensemble d'interfaces qui définissent ce que le composant est capable de fournir et ce que le composant requiert. Les assemblages de composants sont construits en connectant leurs interfaces. Un système logiciel basé sur une approche à composants peut être décomposé en trois parties: ses objectifs fonctionnels et non fonctionnels, son architecture, et son code. La figure 2.2 illustre cette décomposition du système. **[DES08]**

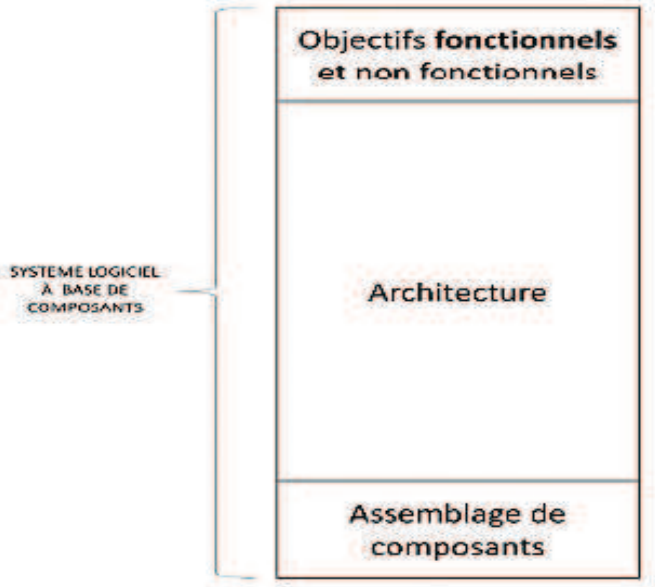

Figure 2.2 : Vue générale d'un système à base de composants.

Les objectifs fonctionnels correspondent à des besoins fonctionnels exprimés par les utilisateurs du système. L'architecture logicielle définit les fondements structurels et comportementaux du système et s'appuie sur les concepts de composant, de connecteur et de connexion. Cette décomposition modulaire permet d'augmenter la compréhension de gros systèmes. Dans **[GAR00]**, un état de l'art consacré à l'étude des architectures logicielles est présenté et renforce l'idée selon laquelle l'architecture fait le lien entre les besoins fonctionnels et le code. On peut noter que le code est obtenu par déploiement (instanciation) d'une architecture à composants (modèle conceptuel) sous la forme d'un assemblage de composants (code exécutable). **[DES08]**

3- Architecture logicielle

L'architecture permet de décrire les éléments de haut niveau qui constituent la structure et le comportement d'un système logiciel en vue de réduire sa complexité. Dans la littérature, la définition la plus citée est donnée par ***Kazman, Bass*** et ***Clements*** dans **[KBC98].**

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

L'une des métaphores souvent utilisée pour expliquer ce qu'est une architecture logicielle est celle du bâtiment. Lors de la construction d'une maison, l'architecte réalise les plans en y intégrant tous les éléments qui composent la maison : briques, tuiles, planches, poutres, etc … Plusieurs vues sont aussi décrites : structurelle, services.

Dans la littérature certains types d'architectures (utilisées de manière récurrente dans les approches à composants) ont été classés et généralisés pour définir des styles architecturaux. L'intérêt d'utiliser un style architectural pour définir une architecture est que chaque style a ses conventions : ils permettent de faciliter la validation des architectures conformément à des modèles dont les qualités non fonctionnelles sont connues par avance. Un exemple de style d'architecture très commun est celui du client-serveur. **[DES08]**

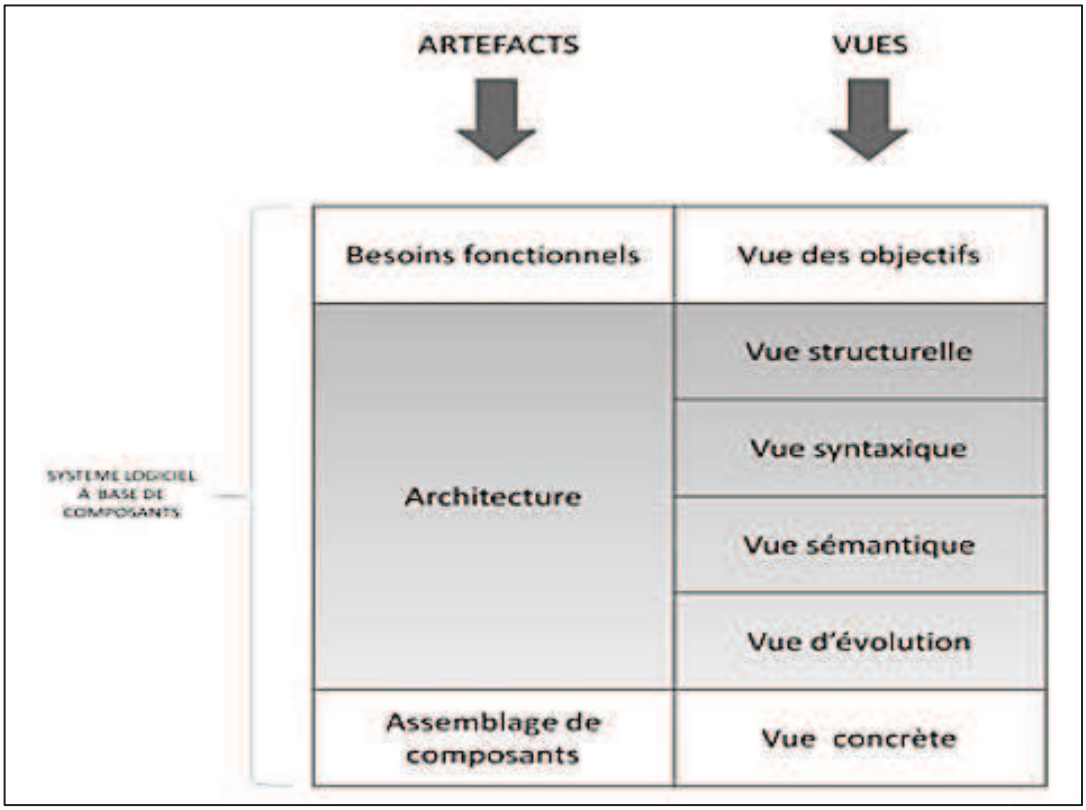

Figure 2.3 : L'architecture d'un système logiciel.

La description que nous proposons dans cette section peut être assimilée à la définition d'un style générique d'architecture. Il est définit suivant quatre vues : la vue structurelle, la vue syntaxique, la vue sémantique et la vue d'évolution (figure 2.3).

a) Vue structurelle

La vue structurelle d'une architecture décrit les interactions structurelles qui existent entre les différentes entités du système (figure 2.4).

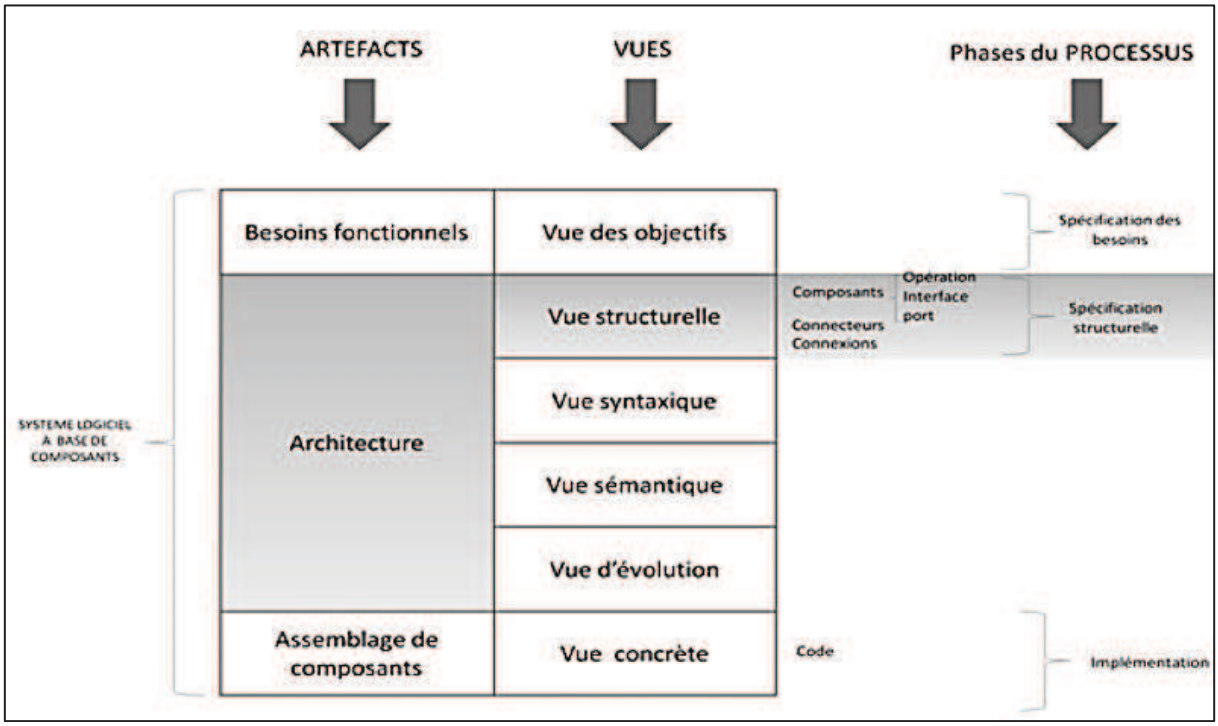

Figure 2.4 : Vue structurelle d'une architecture.

Les éléments structurels peuvent être partitionnés en plusieurs classes : les composants, les connecteurs et les connexions. D'autres éléments comme les opérations, les interfaces et les ports sont des concepts de deuxième niveau utilisés pour représenter des points d'interaction entre le composant et son environnement. D'autres concepts structurels tels que la composition hiérarchique et le partage de composants sont utilisés pour structurer l'information.

b) Vue syntaxique

Les approches à composants, tout comme les approches objets, utilisent la notion de typage. La vue syntaxique permet d'assurer un premier niveau de compatibilité d'échanges entre les composants. Elle utilise la notion de type tel qu'il est défini dans les langages de programmation classiques. Un type syntaxique peut être associé aux points d'interaction d'un composant ainsi qu'au composant (figure 2.5).

- ***Typage des points d'interactions.*** Nous avons présenté, dans le chapitre précédant les trois sortes de points d'interaction que pouvait utiliser un composant pour interagir avec son environnement. A titre de rappel, ces points d'interaction sont: l'opération, l'interface et le port. Un type syntaxique est associé à chaque point d'interaction en vue des contrôles de validité de l'architecture. Le type d'une opération est défini par sa signature et par sa direction. Dans la plupart des approches à composants, le type d'une interface est défini par l'ensemble des types syntaxiques de ses opérations et par sa direction. Le type d'un port est défini par l'ensemble des types syntaxiques de ses interfaces.

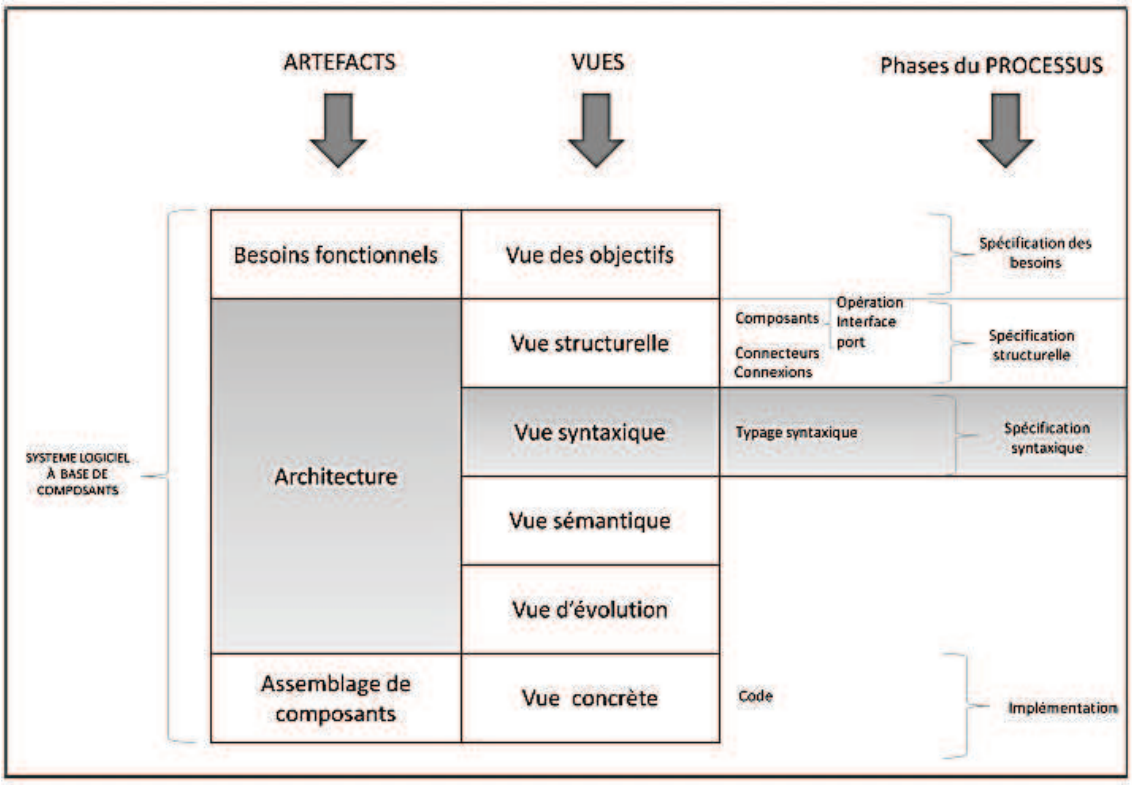

Figure 2.5 : Vue syntaxique.

De manière générale, le type d'un composant est défini par l'ensemble des types syntaxiques de ses points d'interaction.

c) Vue sémantique

La vue sémantique a pour objectif de décrire les aspects dynamiques du système. Elle s'exprime généralement sous la forme de contrats ou de protocoles de comportement. A chaque élément structurel peut être associé un type sémantique. La figure 2.6 illustre la partie du système décrite dans cette section.

- ***Protocole de comportement.*** C'est un ensemble de méta-informations qui peut s'attacher à un élément structurel. Il exprime toutes les séquences valides d'échanges de message transitant par l'entité qu'il décrit. Il peut être représenté au moyen d'une expression régulière. La plupart des approches utilisent des protocoles qui sont des spécifications de séquences valides d'invocations d'opérations. SOFA exprime les protocoles avec des expressions régulières et permet d'atteindre une définition riche et précise des protocoles, qui, en plus de permettre la description des invocations d'opérations entrantes (vers le composant) et sortantes (vers un composant connecté), permettent de détailler l'exécution d'opérations internes. Carrez exprime les protocoles grâce à des expressions régulières et y intègre la modalité (may et must). D'autres modèles utilisent des machines à états pour spécifier les protocoles. OMEGA utilise des *statecharts*.

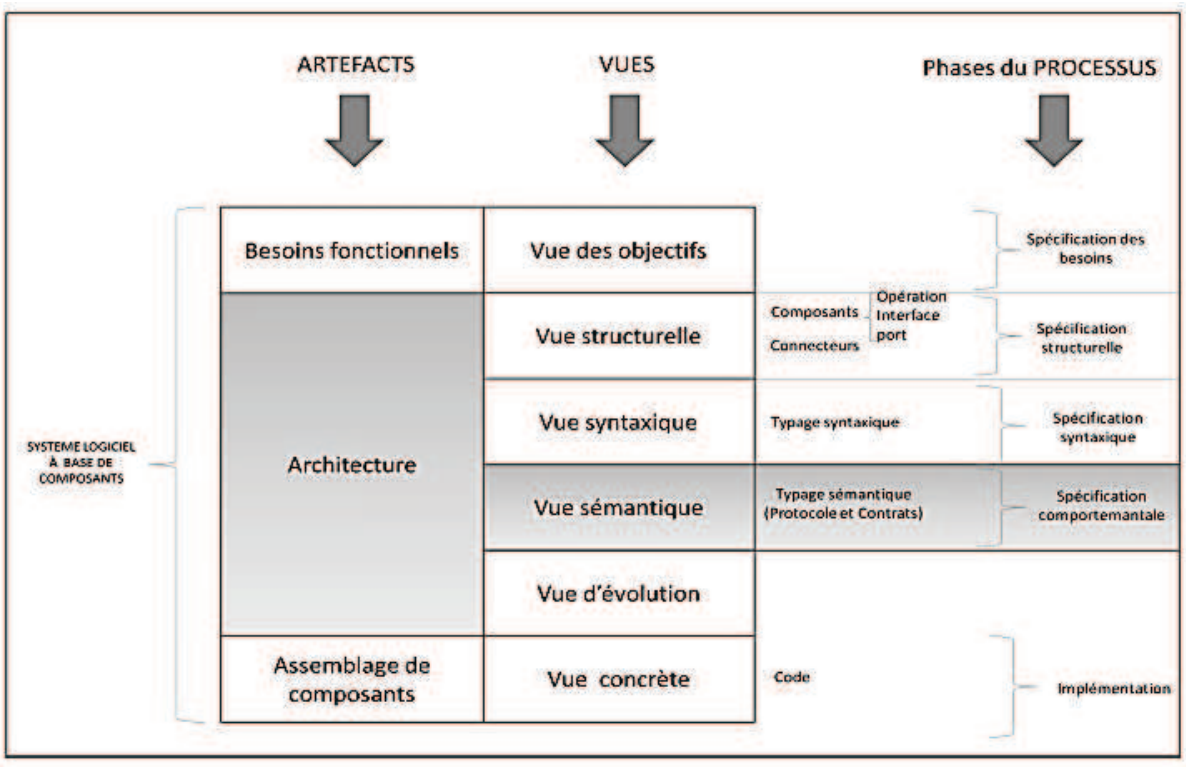

Figure 2.6 : Vue sémantique.

UML 2.0 et Java utilisent des machines d'états de protocoles (PSM) dans le but de décrire les protocoles de façon simple. Cependant, les PSM ne peuvent prendre en compte que des messages orientés dans une même direction. Pour dépasser cette contrainte, Mencl **[MEN04]** propose les PoSM, une adaptation des PSM qui permet de décrire des protocoles de ports qui sont aussi précis que ceux de SOFA. La vérification de protocoles repose sur des comparaisons d'automates dont la principale difficulté est l'explosion combinatoire.

- ***Contrats.*** L'utilisation des contrats a pour but la description de contraintes statiques et dynamiques qui doivent être respectées lors des échanges de services entre composants. Ils sont souvent représentés par des invariants ou des pré/post conditions. Les premiers contrats ont été proposés par Bertrand Meyer dans le langage Eiffel. Dans ce langage, ils peuvent être assimilés à de la méta-information visant à faciliter la compréhension des programmes ainsi qu'à prévenir des erreurs. Meyer précise qu'une assertion est une formule logique ou un prédicat permettant d'exprimer ce que doit faire le module alors que le code exprime comment il doit le faire. Le projet ACCORD propose des contrats utilisant des machines à états. Dans ce modèle, le contrat syntaxique correspond à la description des contraintes liées au typage. Le contrat sémantique correspond aux pré/post conditions. Enfin, certains modèles utilisent des pré/post conditions pour contrôler le séquencement des opérations.

Remarque : Pour les aspects non fonctionnels, les propriétés définissant les services échangés sont généralement exprimées par des assertions. Ces propriétés peuvent résulter d'une combinaison de propriétés requises par différents composants de l'assemblage : elles sont négociées et recueillies dans des contrats qui spécifient comment les composants coopèrent réellement au travers des assemblages.

- ***Typage sémantique des points d'interactions.*** Un type sémantique peut être associé à un point d'interaction. Des assertions, sous la forme de pré/post conditions, peuvent être associées à une opération. Une interface possède un protocole de comportement qui décrit toutes les séquences valides d'invocations de messages correspondant aux opérations de l'interface. Lorsque l'interface est requise, tous les messages sont émis. Lorsque l'interface est fournie, tous les messages sont reçus. Un contrat ou un protocole de comportement peuvent être rattachés à un port.

- ***Type sémantique d'un composant.*** Un contrat ou un protocole de comportement peuvent être rattachés à un composant.

4- Règles d'évolution dynamique

Les approches à composants qui veulent gérer l'évolution dynamique de manière sûre doivent l'anticiper au moment de la phase de conception. Elles proposent un ensemble d'éléments de modèle permettant de décrire comment l'architecture peut évoluer au cours de l'exécution. Nous appelons l'ensemble de ces éléments les règles d'évolution dynamique et la vue du système qu'elles constituent la vue d'évolution.

Les règles d'évolution dynamique d'une architecture visent à assurer que l'assemblage de composants restera valide au cours de l'exécution. La Figure 2.7 illustre la partie du système décrite dans cette section.

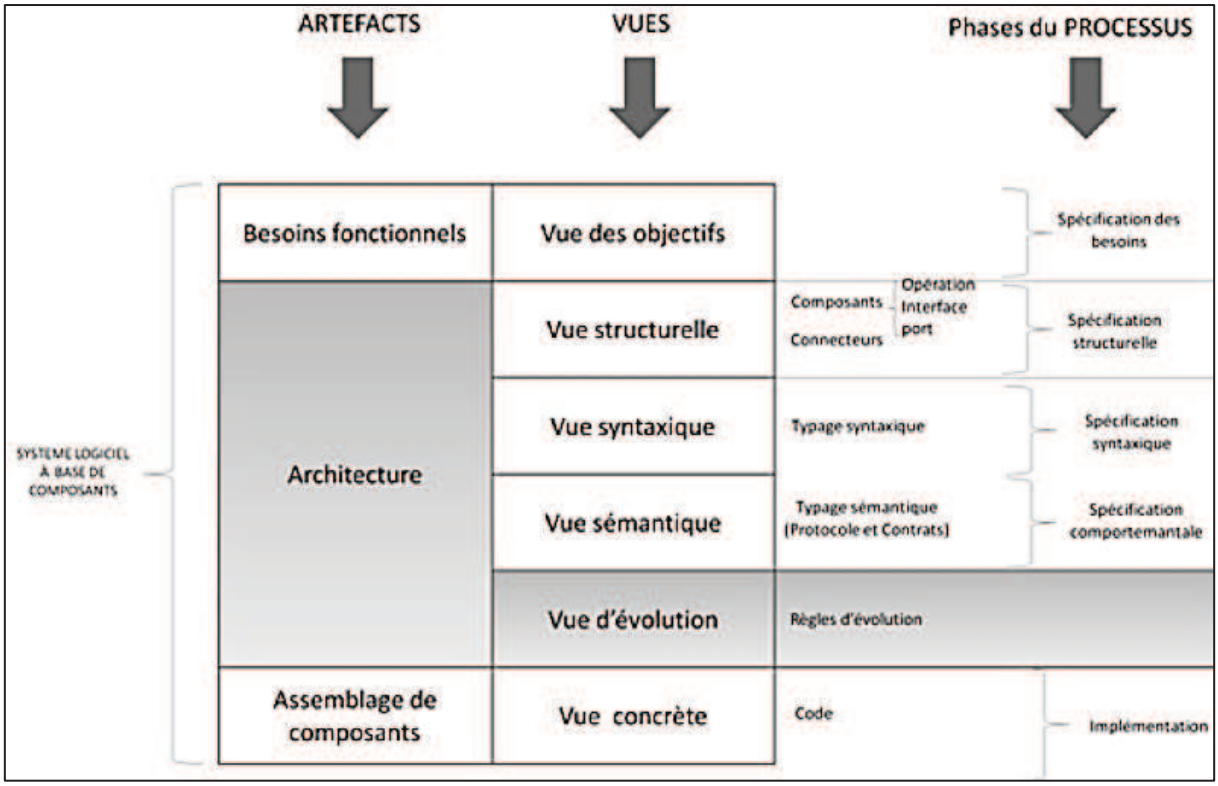

Figure 2.7 : Vue d'évolution.

5- Cycle de développement d'une application logicielle composée

La CBSE (*Component-Based Software Engineering*) se distingue d'une approche non-orientée composants dans le sens où un système logiciel à base de composants est construit à partir de composants pré-existants. On doit donc faire la distinction entre deux cycles de vie : un pour le développement de composants réutilisables (Design for Reuse) et un pour le développement d'applications à partir de ces composants réutilisables (Design by Reuse). Les différentes phases de ce second processus telles qu'elles ont été identifiées dans **[YAH08]** , sont : l'analyse et la spécification des besoins, la conception générale du système logiciel qui intégre les composants, la sélection et le test des composants qui vont être intégrés, l'intégration proprement dite des composants dans le système, la vérification et la validation du système complet, et enfin sa maintenance. Ainsi, la phase d'implémentation, qui prenait l'essentiel de l'effort de développement dans les approches non-orientées composants, se voit remplacée dans la CBSE par la phase de sélection des composants.

Dans le cycle de développement d'une application logicielle composeé, il existe 03 phases **[CER04]**:

- Phase de développement
- Phase d'assemblage
- Phase de déploiement

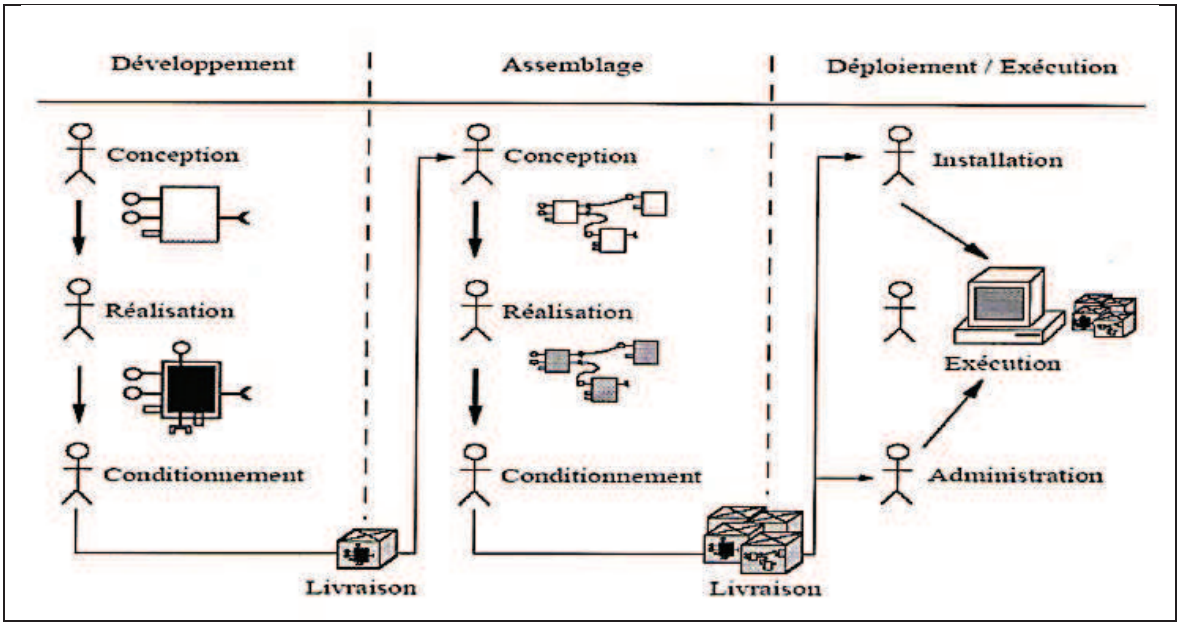

Figure 2.8 : Cycle de développement.

a) Phase de développement : constituée de 03 activités

▪ ***Conception :*** inclut la spécification de la vue externe d'une classe de composants ainsi que son comportement.

▪ ***Réalisation :*** consiste à la création de l'implémentation de la classe de composants qui implémente les vues externes et internes.

▪ ***Conditionnement :*** la classe de composants est introduite dans un paquetage de composants qui permet de réaliser la livraison et le déploiement indépendants.

b) Phase d'assemblage : suppose l'existence de composants développés auparavant, Cette phase inclue :

▪ ***Conception :*** la conception d'un assemblage est réalisée en étudiant la façon d'intégrer un ensemble de composants préexistants pour créer une composition représentant soit une partie ou la totalité d'une application.

▪ ***Réalisation :*** la réalisation de l'assemblage consiste à l'écriture du code permettant de réaliser la composition des instances de composants.

▪ ***Conditionnement :*** l'assemblage est introduit dans un paquetage d'assemblage qui peut inclure les composants employés dans l'assemblage ainsi que des ressources propres à l'assemblage telles que des fichiers de configuration ou des fichiers binaires (librairies, images,etc).

c) Phase de déploiement(Exécution) : Après le développement et l'assemblage dans cette phase on trouve :

▪ ***Déploiement :*** lors de l'installation, des paquetages contenant des compositions et des composants sont installés et des activités de configuration peuvent être réalisées.

▪ ***Exécution :*** l'application est exécutée. A ce moment-là, l'environnement d'exécution est actif.

▪ ***Administration :*** l'administration inclut l'application de mises à jour ainsi que la désinstallation d'une application et de ses composants.

6- Assemblage de composants logiciels

L'assemblage de composants peut être découpé en 02 parties **[ACC03]**:

- L'aspect ensembliste de l'assemblage : la composition d'un assemblage (les différentes entités qui constituent un composant).
- L'aspect de communication entre composants : les interactions entre composants.

Les relations de dépendance entre les interfaces de composant sont : **[YAH14]**

a. Dépendances de services offerts (liaisons de type 1) : désignent les liaisons entre interfaces offertes visibles du composite et interfaces offertes par des composants membres du composite. Dans ce cas, la spécification offerte du composite, rendue visible, est identique à la spécification offerte d'un composant membre (principe de délégation).

b. Dépendances de services requis (liaisons de type 2) : désignent les liaisons entre interfaces requises de composants membres et les interfaces visibles du composite (principe de délégation).

c. ***Dépendances inter-composants (liaisons de type 3) :*** il s'agit de liaisons d'assemblage entre interfaces requises et interfaces offertes. Dans le cas général, il existe une relation de dépendance entre services offerts et requis.

d. ***Dépendances intra-composants (liaisons de type 4) :*** ces liaisons sont créées pour supporter les dépendances d'implantation entre interfaces offertes et interfaces requises d'un composant.

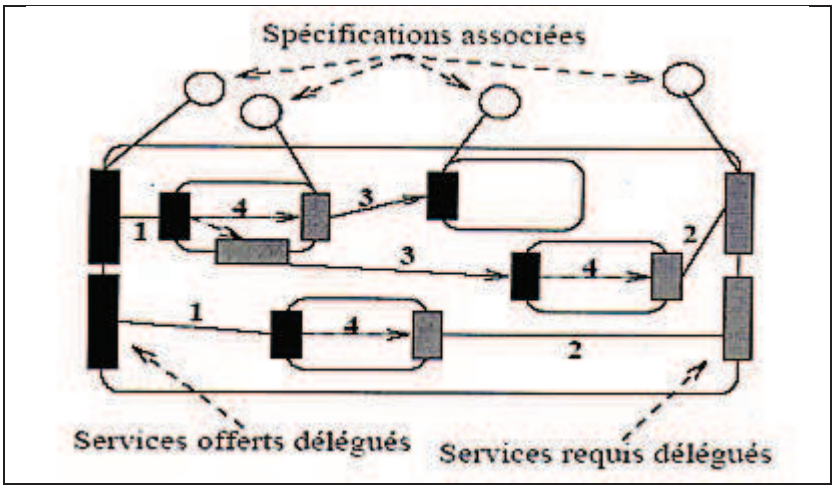

Figure 2.9 : Relation de dépendance entre les interfaces de composants.

Les deux principales approches d'assemblage actuelles sont *les langages de description d'architecture* (**ADL** : Architecture Description languages) et *les langages de coordination*. **[ACC01]**

La définition d'une architecture logicielle est une étape importante dans la conception d'un logiciel. Elle décrit l'ensemble des composants et donne les règles de leur assemblage. Elle correspond à l'établissement du plan de construction du logiciel. En maitrisant l'architecture conceptuelle, il est alors plus facile de gérer ses éventuelles évolutions. En effet, la modification d'un plan est plus simple que la modification d'un système réalisé.

Les concepts des ADL sont :

- ***Composant :*** unité de composition et de déploiement, qui remplit une fonction spécifique et peut être assemblé avec d'autres composants. A cet effet, il porte une description des interfaces requises et fournies.

- ***Connecteur :*** élément permettant d'assembler des composants en utilisant leurs interfaces fournies et requises. Il remplit les fonctions de liaison et de communication.

- ***Configuration :*** la configuration structurelle de l'application correspond à un graphe connexe des composants et des connecteurs formant l'application. La configuration comportementale, quant à elle, modélise le comportement en décrivant l'évolution des liens entre composants et connecteurs, ainsi que l'évolution des propriétés non fonctionnelles.

L'approche langages de coordination rend les modules plus indépendants, elle permet l'évolution d'applications en modifiant seulement les éléments concernés et non en intervenant sur toute l'application. Elle permet de réutiliser d'une part les objets sans la façon dont ils sont coordonnés, et d'autre part les entités de coordination sans les objets à coordonner.
Cette théorie a conduit à la proposition d'un grand nombre de modèles de coordination et de leur langage de programmation associé.

Les concepts de base de langages de coordination

- ***La coordination :*** il existe plusieurs définitions. Elle est définie comme :

- l'intégration et l'ajustement harmonieux des efforts de travaux individuels à travers la réalisation d'un objectif plus large.
- Le processus de construction de programmes en collant ensemble des pièces actives.

La coordination peut être exprimée en termes de modèles et langages de coordination.

- ***Un modèle de coordination :*** se définit par « le lien qui relie des activités séparées dans un ensemble. Il est généralement caractérisé par trois éléments :
  1. Les *entités* à coordonner.
  2. Le *médium* qui définit le lieu ou se fait la coordination.
  3. Le *type de coordination* qui est la technique utilisée pour coordonner les entités.

- ***Un langage de coordination :*** est l'incarnation linguistique d'un modèle de coordination. Un modèle de coordination fournit donc la sémantique, alors qu'un langage de coordination correspondant fournit une syntaxe pour utiliser le modèle dans une implantation.

6.1- 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 (La réification d'un concept consiste à créer les entités nécessaires pour l'implémentation de sa sémantique) par les connecteurs logiciels. Ces derniers sont proposés principalement pour faciliter l'assemblage des composants en jouant le rôle d'intermédiaires entre les composants non-compatibles. **[HAD03]**

« *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.* »

L'objectif des connecteurs est d'atteindre une meilleure réutilisabilité lors de l'assemblage des composants. 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.

6.2- 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 complexes en jouant le rôle d'adaptateurs d'interfaces.

Medidovic **[MTA00]** a classé les connecteurs en huit classes, 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.

Chaque type de connecteur offre un ou plusieurs service (s) d'interaction. Les services sont les suivants :

1) **Le service de communication.** Un connecteur assure ce service s'il s'occupe des transmissions des données entre composants.

2) **Le service de coordination** qui supporte le transfert de contrôle entre composants. Les appels des fonctions est un exemple de cette catégorie de connecteurs.

3) **Le service de conversion** convertit les interactions inter-composants si nécessaire. Il permet aux composants hétérogènes d'interagir.

4) **Le service de facilitation** négocie et améliore l'interaction entre composants. Ce service est assuré par le connecteur d'équilibrage de charge.

7- Assemblage correct de composant logiciels

Dans **[BEU05]** la définition de l'assemblage correct est donnée comme suit :
Un assemblage correct est un assemblage :

- ✓ ***Compilable :*** tous les objets (données, méthodes) sont définis, les types compatibles.
- ✓ ***Exécutable :*** il existe une exécution du résultat de la compilation qui s'exécute sans erreur inattendue (exception, interblocage, non respect de la spécification).
- ✓ ***Sûr :*** toutes les exécutions du résultat de la compilation s'exécutent sans erreur inattendue.
- ✓ ***Correct :*** sûr et conforme aux exigences non-fonctionnelles telles que performance, fiabilité,...etc.

La vérification d'un système à composants consiste une vérification de type correction syntaxique et comportementale d'une architecture. Mencl **[MPA05]** met l'accent sur le fait qu'il est important de vérifier que le raisonnement exprimé sous forme de cas d'utilisations est cohérent et que le comportement du système est correct. Muccini **[MPR07]** exprime aussi le besoin de vérifier que le système satisfait bien certaines propriétés de la spécification. De manière intuitive, un assemblage de composants est valide s'il permet de répondre de manière correcte aux objectifs fonctionnels du système.

Conclusion

Cependant, face au nombre croissant et à la complexité accrue des logiciels, ainsi qu'à des impératifs de plus en plus forts de mise sur le marché, les entreprises peuvent de moins en moins s'offrir le luxe de construire systématiquement leurs propres composants. Elles sont de plus en plus obligées d'avoir recours à des composants pré-développés et achetés à des tierces personnes. Le plus souvent, les composants intégrés dans les applications sont donc des composants sur étagère.

Nous concluons que l'effort de développement réside désormais dans la sélection des composants qui répondent le mieux aux besoins apparaissant lors du développement d'une application.

Chapitre 3

La qualité logicielle

Introduction

La qualité fournie est la principale préoccupation de l'utilisateur. Elle ne dispose pas d'une définition consensuelle qui convient à tous les domaines auxquels elle s'applique.
Quelque soit le contexte, la qualité recouvre des propriétés non fonctionnelles d'une entité informatique. Le niveau de qualité est caractérisé par les « valeurs » particulières des propriétés non fonctionnelles déterminées par les types de gestion choisis.
Maintenir le niveau de qualité stable malgré les variations d'environnement est l'un des grands défis des recherches aujourd'hui.

Définir la qualité de service d'un assemblage de composants, consiste à définir l'ensemble des propriétés, des caractéristiques, des performances du service fourni au client présentées en comparaison avec le service attendu par celui-ci. **[DLR00]**

Dans ce contexte, il existe plusieurs méthodes proposées pour apprécier la qualité d'une application composée.

Dans ce chapitre nous introduisons les différentes notions concernant la qualité, les différentes terminologies, qu'est-ce que la qualité de service ?, importance de la qualité des logiciels et les différents types et niveaux de qualité, après nous donnons les définitions des termes facteurs, critères et métriques de qualité. Enfin nous présentons quelques normes et modèles de qualité logicielles.

1- Définitions [MRO06] [YAH08] [GPA00]

- **Erreur :** une erreur est commise par le développeur, elle conduit à un défaut.
- **Défaut :** un défaut est une imperfection dans le logiciel, il peut conduire ou non à une panne.
- **Panne (bug) :** une panne est un comportement anormal d'un programme (arrêt du programme ou résultat non attendu).

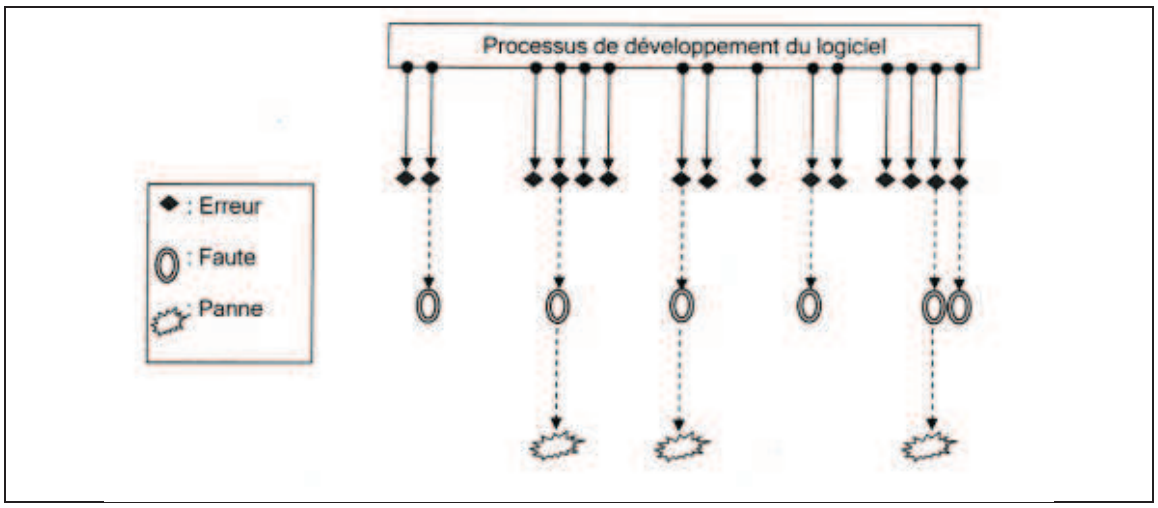

Figure 3.1 : Relation entre erreur, faute et panne.

Parmi des sources d'erreur on trouve :

1. Mauvaise définition des exigences
2. Problèmes de communication entre le client et développeur.
3. Déviations délibérées des exigences du logiciel.
4. Erreur de conception.
5. Erreur de programmation.
6. Non-conformité à la documentation ainsi qu'aux instructions standards de programmation.
7. Insuffisances du processus de tests.
8. Erreur de l'interface usagers.
9. Erreur de documentation.

- ## Qualité logicielle

De manière générale, un logiciel de qualité s'entend comme un logiciel capable de répondre parfaitement aux attentes du client, le tout sans défaut d'exécution. Ainsi, on détermine la qualité logicielle comme un ensemble de règles et de principes à suivre au cours du développement d'une application afin de concevoir un logiciel répondant à ces attentes **[ABD04].** La NASA par exemple, a déterminé un ensemble de procédures, d'instructions de travail et de règles pour s'assurer que chaque étape du développement s'effectue de manière adéquate **[MAR06].** La qualité d'un logiciel se reflète non seulement dans les processus de développement mais aussi dans la qualité des éléments qui le constituent, la documentation, la présence de tests.....

Mesurer la qualité d'un logiciel consiste alors à déterminer son adéquation par rapport aux objectifs de départ et aux standards de programmation. Il faut donc définir précisément ce que l'application doit faire (caractéristiques fonctionnelles) et comment elle doit le faire (caractéristiques non fonctionnelles). Une fois ces objectifs fixés, on peut alors appliquer un ensemble de règles et de mesures afin de calculer la différence entre objectifs attendus et réalisation obtenue.

Obtenir une mesure de la qualité permet à la fois d'avoir une image précise du logiciel mesuré mais aussi de déterminer son comportement dans le temps : quels sont les risques de bogues, les éventuelles failles sécuritaires, les difficultés de maintenance, les freins à l'évolution, la viabilité à long terme, etc.

Un des objectifs de la mesure de la qualité logicielle consiste à sensibiliser les équipes de développement sur leurs méthodes de programmation. En effet, mesurer la qualité a également pour objectif de fournir des bonnes pratiques de travail et des indicateurs permettant d'augmenter la qualité des futurs développements.

Pour obtenir une image complète de la qualité d'un logiciel on fait appel à un modèle de qualité. Celui-ci regroupe des règles qui décrivent ce que doit être un logiciel de qualité et le répertorie en différents groupes. Le modèle est ensuite évalué à partir de mesures obtenues grâce au code source, à la documentation, aux annexes techniques, aux règles de conception ou toute autre information disponible pour le projet.

Un modèle de qualité comporte souvent plusieurs niveaux de lecture. Il se compose d'une couche de haut niveau qui décrit la qualité selon un point de vue très généraliste. Ce premier niveau est ensuite décrit plus spécifiquement, le tout pour atteindre ensuite le plus bas niveau.

La qualité : la qualité d'un logiciel est définit par la conformité aux exigences. Le sens le plus étroit de qualité du logiciel est : **[GPA00]**

- ✓ Absence de bugs.
- ✓ Bas ratio de défauts.
- ✓ Haute fiabilité.

▪ ***Corollaire :*** un produit est *de qualité* dès qu'il répond aux critères de qualité qu'on lui a attribués. Donc un produit bas de gamme peut être considéré de qualité.

▪ ***La qualité de service :*** un service est le résultat produit par des activités à l'interface entre le fournisseur et le client et par des activités internes des fournisseurs pour satisfaire les besoins des clients. C'est l'ensemble des fonctions fournies et accessibles aux clients.

« *La qualité de service (QoS) est l'effort des performances du service qui détermine le degré de satisfaction de l'utilisateur du service* » **[YAH08]**

2- Importance de la qualité des logiciels [GPA00]

Les défauts du logiciel sont extrêmement coûteux en terme : d'argent, de réputation, et de perte de vie.
Pour illustrer l'importance de la qualité nous allons présenter des exemples de désastres historiques attribués au logiciel :

- Le bug de l'an 2000 : la lutte contre le bug de l'an 2000 a coûté à la France 500 milliards de francs.

Cause : la donnée « année » était codée sur deux caractères.

- Mission Vénus : passage à 5.000.000 de Km de la planète, au lieu de 5 000 Km prévus.

Cause : remplacement d'une virgule par un point.

- DAB : panne totale durant tout un week-end de 1996 des distributeurs automatiques de billets, en région parisienne.

Cause : bug dans le logiciel de compensation inter bancaire.

En général, il existe deux visions pour la qualité : **[RAD04] [YAH08]**

- Vision du développeur.
- Vision d'utilisateur.

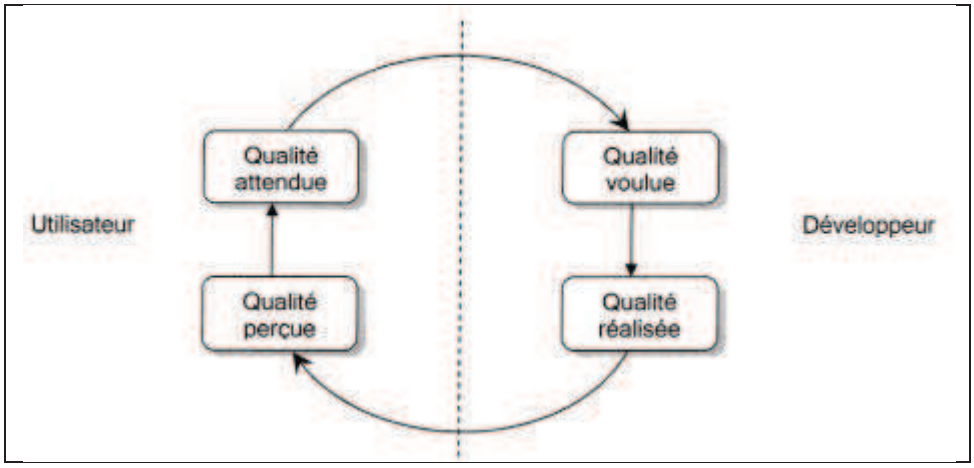

Figure 3.2 : Différents types de qualité.

Du point de vue développeur, nous distinguons :

- ***La qualité voulue :*** caractérise la qualité que l'organisme souhaite atteindre pour répondre à la qualité attendue. C'est la prestation qu'il veut fournir à ses clients

- ***La qualité réalisée :*** caractérise la qualité réellement réalisée (mesureé).

Du point de vue client nous distinguons :

- ***La qualité attendue :*** caractérise la qualité souhaitée par les clients, c'est-à-dire la réponse à leurs besoins et leurs attentes. Cala signifie que les conditions d'obtention de ce niveau de qualité ont été prédéfinies.

- ***La qualité perçue :*** caractérise la qualité ressentie, de façon plus ou moins confuse par le client à partir de ses propres besoins et attente. Elle est l'expression de sa satisfaction.

Le développeur évalue le degré de satisfaction de ses clients en mettant en place des outils de mesure de la satisfaction des clients et de la performance du service rendu.

3- les facteurs, critères et métriques de qualité

➢ **Définitions : [MRO06] [YAH14]**

- ***Facteur de qualité (F):*** caractéristique du logiciel qui contribue à sa qualité. Ils concernent les caractéristiques d'utilisation liées à :
  - L'environnement d'exploitation.
  - L'environnement de suivi et de maintenance.

Les facteurs traduisent la vision externe que peut avoir l'utilisateur, ils sont estimés par les critères.

- ***Critères de qualité (C):*** attribut du logiciel par l'intermédiaire duquel un facteur peut être évalué. Ils sont :
  - Orientés réalisateur.

- Ils sont les composantes des facteurs de qualité, ils sont mesurés par les métriques.

Les critères de qualités concernent les caractéristiques d'utilisation en fonction d'une vision interne (structure de logiciel).

Un facteur F_i peut être évalué de la façon suivante :

$F_i = a_0C_0 + a_1C_1+a_2C_2+.....+a_nC_n$
Tel que :

$\{C_0, C_1, C_2,.....,C_n\}$ l'ensemble des critères du facteur Fi.
$a_0, a_1, a_2,.....a_n$ sont des coefficients de pondération associés aux critères.

- ***Métriques de qualité(m) :*** les critères de qualité sont estimés par des métriques qui sont des mesures à posteriori.

Un critère C_j peut être mesuré de la façon suivante :
$C_j = b_0m_0+b_1m_1+b_2m_2+....+b_mm_m$
Tel que :

$\{ m_0, m_1, m_2,.....,m_m \}$ l'ensemble des métriques du critères C_j.
$b_0, b_1, b_2,.....b_m$ sont des coefficients de pondération associés aux métriques.

Il existe 03 types de métriques pour estimer les critères.

1. ***Présence :*** cette métrique identifie, qu'un attribut soit présent dans un composant ou non, elle est décrite par une variable de type booléen.

2. ***Dénombrement :*** cette métrique est employée pour indiquer des valeurs exactes. Elle est décrite par une variable de type entier.

3. ***Ratio :*** cette métrique est employée pour décrire des rapports (pourcentage).

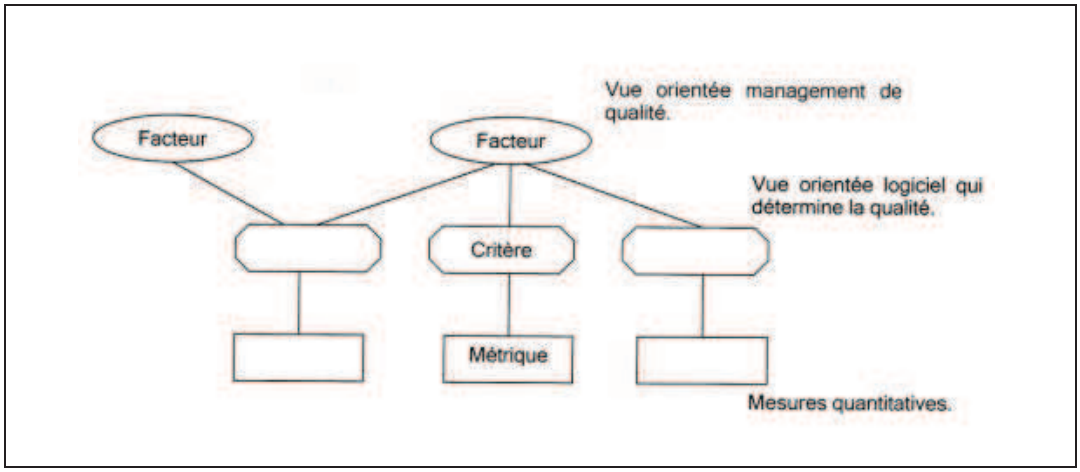

Figure 3.3 : Relation entre facteurs, critères et métriques de qualité.

4- Normes et modèles de qualité logicielle

Un modèle de qualité se compose d'un certain nombre de règles et de principes, réparties en différentes catégories. Appliquer un modèle de qualité consiste à mesurer ces règles. Pour y parvenir, le modèle collecte un certain nombre d'informations issues du projet dont des métriques de code.

Il existe un nombre considérable de métriques et certaines d'entre elles ne sont pas calculées de la même manière selon l'outil utilisé. C'est pourquoi il convient de déterminer quelles métriques seront utiles et comment elles doivent précisément être calculées.
[FLP96] [LMA06] [JON08] [BBD09]

On peut toutefois classer les métriques en deux grandes catégories :

- **Les métriques primitives.** Il s'agit des métriques qui mesurent des propriétés de base du code source telles que le nombre de lignes de code, la complexité cyclomatique ou encore les metriques de Chidamber et Kemerer **[ACC94]** (profondeur d'héritage, nombre de sousclasses, etc.) ou les métriques de Lorenz et Kidd **[LKI94]** (nombre de méthodes, nombre de méthodes héritées, etc.)
- **Les métriques de design**. Elles déterminent si le code source respecte les principes de conception préalablement définies. Il s'agit de métriques telles que les métriques de couplage ou de cohésion de classes, ou encore les métriques d'architecture de packages comme les métriques de couplage afférent de Martin **[MAR97].**

De même, une application peut fournir d'autres sources de renseignements qu'il faut savoir interpréter dans le cadre d'une démarche qualité. Les modèles de qualité doivent donc fournir des indications précises sur l'utilisation et l'évaluation de ces informations. Il s'agit de donner du sens en termes de qualité à des informations brutes.

Les modèles les plus connus actuellement sont des modèles hiérarchiques qui recensent les principes de qualité en partant des exigences globales et des principes les plus généraux pour descendre vers les métriques qui permettent de les mesurer. Ceci implique que la mesure de la qualité ne peut débuter qu'une fois le modèle totalement spécifié et que les premiers résultats ne peuvent être obtenus qu'une fois la collecte des données suffisante.

ISO 9126 **[ISO01]** est une norme standard internationale visant à évaluer la qualité logicielle. Elle normalise et classifie un certain nombre de principes qualité. Réalisée par le comité technique JTC 1 de l'ISO/CEI, cette norme évolue pour être enrichie et intégrée dans la norme SquaRE (Software product Quality Requirement and Evaluation : exigences et évaluation de la qualité du logiciel). Elle est composée de six caractéristiques générales qui définissent la qualité globale d'une application : la capacité fonctionnelle, la fiabilité, l'efficacité, la maintenabilité, la facilité d'usage, la portabilité. Chacune de ces caractéristiques est décomposée en sous caractéristiques.

Cette norme semble être une bonne approche pour déterminer la qualité d'un logiciel dans son ensemble et fournir une vue globale satisfaisante. Cependant, la norme ne précise pas de manière explicite comment mesurer les caractéristiques qualités définies et comment les relier

aux métriques de bas niveau. Il n'y a aucun continu entre ces deux niveaux. Ainsi, le modèle reste trop abstrait : bien que constituant une base théorique solide, le fait de devoir l'adapter à chaque cas de figure sans avoir de guide précis pour le faire augmente d'autant sa difficulté de mise en œuvre.

SquaRE. La norme SquaRE **[ISO05]** définie depuis 2005 est la norme qui succède au standard ISO 9126. Elle a été définie à partir d'ISO 9126 et de la partie évaluation de la norme ISO 14598.
Le système SquaRE décrit deux modèles distincts. Un modèle de qualité lié à l'utilisation du logiciel et un modèle de qualité propre à la production logicielle. Nous nous intéresserons ici uniquement à ce dernier.

Suivant le même principe que la norme ISO 9126, le modèle est constitué de huit caractéristiques, décomposées en sous caractéristiques :

Adéquation fonctionnelle : degré à partir duquel un produit offre les fonctions répondant aux besoins exprimés et implicites dans des conditions d'utilisation spécifiées.

Performances : performances par rapport aux ressources utilisées dans des conditions déterminées.

Compatibilité : degré à partir duquel un produit peut échanger des informations avec d'autres produits et/ou remplir ses fonctions, tout en partageant les mêmes environnements matériel et logiciel.

Facilité d'utilisation : degré à partir duquel un produit peut être utilisé pour atteindre les buts identifiés, avec efficacité, efficience et satisfaction, dans un contexte spécifié.

Fiabilité : degré à partir duquel un produit exécute les fonctions spécifiées dans des conditions spécifiées pour une période de temps spécifiée.

Sécurité : degré à partir duquel un produit protège les informations et données de manière à ce que les personnes ou autres produits aient un accès à ces derniers qui corresponde à leur niveau d'autorisation.

Maintenabilité : degré d'efficacité et d'efficience à partir duquel un produit peut être modifié par les personnes adéquates.

Portabilité : degré d'efficacité et d'efficience à partir duquel un produit peut être transféré depuis un environnement matériel ou logiciel vers un autre, ou d'un usage à un autre.

La norme SquaRE redéfinit de manière beaucoup plus judicieuse les caractéristiques qualités d'un logiciel. Le fait, par exemple, d'avoir distingué la partie sécurité comme étant désormais une caractéristique à part entière, ou encore d'avoir fait une distinction entre la portabilité et la compatibilité rendent le modèle plus pertinent. Cependant, là encore, ce modèle généraliste est difficile à mettre en œuvre ; il n'existe aucun lien explicite entre les caractéristiques (au sens facteurs et critères) et les métriques. De plus, ces normes définissent des modèles de qualité généraux qui visent à qualifier un logiciel dans son ensemble.

Le Modèle de McCall: Facteurs Critères Métriques (FCM). McCall **[MRW76]** a défini un modèle appelé facteurs-critères-métriques pour évaluer la qualité d'un système. Il a identifié cinquante facteurs et a sélectionné les onze principaux représentant une vision externe globale de la qualité. Ces facteurs sont caractérisés par vingt-trois critères qui représentent la vision interne de la qualité : le point de vue du programmeur. Bien que complet, ce modèle est difficile à mettre en œuvre du fait des 300 métriques qui le composent. Il est toutefois implémenté dans quelques outils commerciaux mais la correspondance entre les métriques et les critères manque de clarté. Ce modèle présente également une faiblesse importante : le manque de lisibilité. En effet, lorsqu'un critère obtient une faible note, il est difficile, voire impossible de relier cette note directement au problème qu'elle pointe, surtout lorsque le critère est composé de plusieurs métriques. Dans ce cas il devient difficile de trouver comment remédier au problème existant.

GQM (Goal-Questions-Metrics). Il s'agit d'une approche de la qualité logicielle qui a été promu par Basili **[BCR94]**. Il définit la mesure de la qualité sur trois niveaux : le niveau conceptuel (the ***goal*** level), le niveau opérationnel (the ***question*** level), et le niveau quantitatif (the ***metric*** level).

- Le niveau conceptuel fixe les objectifs de mesure à savoir, ce qui doit être mesuré, le niveau à atteindre en terme de qualité, les objectifs visés par l'entreprise.
- Le niveau opérationnel définit pour sa part quelles sont les questions à se poser pour déterminer si les objectifs définis au niveau précédent sont atteints.
- Le niveau quantitatif détermine quelles métriques doivent être utilisées pour mesurer le niveau précédent.

Même si ce modèle est largement diffusé dans l'industrie, il pose tout de même le problème de ne pas expliciter clairement comment intégrer les stratégies et les buts spécifiques aux entreprises dans le modèle. De plus, ce modèle ne sépare pas clairement les différents points de vue entre les managers et les développeurs et ne permet donc pas une lecture claire systématique et aisée des résultats obtenus.

QMOOD. Le modèle Quality Model for Object-Oriented Design (QMOOD) est également un modèle hiérarchique basé sur la norme ISO 9126. Il est composé de quatre niveaux : les attributs de la qualité du design, les propriétés du design orienté objet, les métriques du design orienté objet et les composants du design orienté objet. Ces attributs de haut niveau sont évalués en utilisant un ensemble de propriétés empiriquement identifiés et pondérés [**BDA02**]. Ce modèle est conçu pour des applications orientées objet et ne peut s'appliquer pour d'autres paradigmes. Plus encore, il ne qualifie que la conception des programmes : il ne prend pas en compte la qualité de l'implémentation ou le respect des règles de programmation par exemple.

Facteur-Stratégie. Marinescu et Ra¸tiu **[MRA04]** sont partis de la question suivante Comment pouvons-nous travailler en partant des résultats ? Leur idée est de relier clairement les facteurs de qualité au code source en utilisant une stratégie de détection. Ils définissent cette stratégie de détection comme un mécanisme générique permettant d'analyser le code source en utilisant des métriques. Les métriques sont soumises à des mécanismes de filtrage et de composition. Ils présentent un nouveau modèle de qualité appelé facteur-Stratégie à partir de cette stratégie. Ce modèle est pertinent pour mesurer les concepts orientées objets mais tout comme le modèle précédent il ne peut s'appliquer pour d'autres concepts et ne couvre pas l'intégralité des propriétés d'une application.

SourceInventory. Bakota et Guymóthy ont présenté un modèle appelé SourceInventory **[BBF08]** qui collecte des mesures telles que des métriques et des informations de couverture de tests. Ce modèle fournit une aide pour interpréter les données collectées mais demeure toutefois un modèle de bas niveau qui ne fournit pas une vue globale et de haut niveau de la qualité.

The Sqale Quality Model. Ce modèle est un modèle hiérarchique qui en suit les principes : trois niveaux allant du plus général—les caractéristiques— au plus détaillé — les mesures des points de contrôle. Les caractéristiques de Sqale ont été déterminées à partir d'un modèle générique du cycle de vie d'un logiciel. Elles sont représentées selon un modèle en couche qui implique que chaque caractéristique doit être validée pour passer à la suivante, tout comme chaque étape de développement d'un logiciel doit être validée pour continuer. Les évaluations des caractéristiques sont basées sur des index de remédiation calculant la distance entre le code analysé et l'objectif qualité à atteindre. Calculé pour chaque composant du code, cet index représente l'effort de remédiation nécessaire pour corriger le composant mesuré. L'index d'un composant se calcule par addition des index de ses constituants. De même, l'index d'une caractéristique se calcule à partir de la somme de ses sous caractéristiques.

Ce modèle a été créé pour mesurer la qualité du code produit et l'évalue en terme d'effort de remédiation uniquement. Il est particulièrement adapté aux développeurs pour lesquels il a été conçu. En revanche, il ne tient pas compte de la qualité fonctionnelle du logiciel.

Conclusion

Mesurer la qualité d'un logiciel est plus complexe qu'il n'y parait. Il faut tout d'abord définir clairement ce que sous-entend le mot qualité pour chaque entreprise, chaque équipe et chaque logiciel mesuré. Une fois les objectifs qualités clairement établis, encore faut-il savoir comment les mesurer. Il ne suffit pas simplement de collecter des résultats de métriques mais surtout de donner du sens à ces différentes mesures.

Le modèle ISO 9126 est un standard de la qualité du produit grandement utilisé depuis sa création au début des années 80. Il est inspiré du modèle de McCall. En plus de l'organisation hiérarchique des différentes caractéristiques de la qualité du produit, le modèle ISO 9126 offre un ensemble de définitions et de termes usuels pour l'évaluation de la qualité du produit logiciel.

Les modèles de type ISO 9126 servent de cadre en établissant différentes catégories, en classant différents concepts, mais restent toutefois trop imprécis ou trop difficiles à mettre concrètement en œuvre pour donner une totale satisfaction.

Un autre défi à relever consiste à déterminer une note globale pour une caractéristique donnée à partir de métriques brutes. Il faut pouvoir combiner ces métriques de manière à traduire les informations qu'elles délivrent pour fournir une lecture aisée de la qualité. De simples transpositions ou de simples moyennes ne permettent pas d'y parvenir de manière totalement satisfaisante.

Dans le chapitre suivant, constituant notre contribution, nous reviendrons sur le modèle de qualité pour les composants, et l'approche retenue et la qualité d'une composition.

Chapitre 4

Contribution

Introduction

L'objectif est de choisir le meilleur assemblage (solution) en terme de qualité parmi plusieurs composants possibles, ce type de problématiques préconise en général l'usage de techniques dites de prise de décision multicritère, pour faire le choix, il faut que chaque assemblage (solution) présente une valeur numérique, ceci permettra à l'utilisateur de comparer les assemblages et de choisir celui qui lui convient, il est possible aussi de prendre en compte des aspects de gestion tels que le temps ou les coûts lors de choix.

Le composant est l'entité de base d'une composition, il serait intéressant de s'attarder sur le modèle de qualité pour les composants, de développer la démarche adoptée pour enfin présenter le processus d'évaluation de la qualité.

1- Modèle de qualité pour les composants logiciels

Le modèle de qualité pour les composants logiciels illustré dans le tableau 4.1 est basé sur le modèle de qualité d'ISO 9126 avec quelques adaptations pour les composants. **[YAH08]**

Il existe 02 types de sous caractéristiques :

1. Les caractéristiques de qualité qui sont observables à l'exécution (comportement) : sont sensibles au temps d'exécution des composants.
2. Les caractéristiques de qualité qui sont observables pendant le cycle de vie : sont sensibles au composant et au développement de systèmes à base de composants.

Caractéristiques	**Sous caractéristiques (Comportement)**	**Sous caractéristiques (Cycle de vie)**
Fonctionnalité	- Exactitude - Sécurité	- Convenance - Interopérabilité - Conformité - Contenu individuel
Fiabilité	- Tolérance aux fautes - Récupération	- Maturité
Utilisabilité	- Configurabilité	- Compréhension - Opérabilité
Efficacité	- Comportement de temps - Comportement de ressources - Scalabilité	
Maintenabilité	- Stabilité	- Testabilité
Portabilité	- Déployabilité	- Adaptabilité - Réutilisabilité

Tableau 4.1 : Modèle de qualité pour les composants logiciels.

- ***Fonctionnalité :*** elle exprime la capacité d'un composant de fournir les services exigés.

- ***Fiabilité :*** elle exprime la capacité d'un composant de maintenir un niveau indiqué d'exécution, pendant une quantité de temps donnée.

- ***Utilisabilité :*** elle exprime la capacité d'un composant d'être compris, appris, employé, configuré, et exécuté.

- ***Efficacité :*** elle exprime la capacité d'un composant de fournir l'exécution appropriée, relativement à la qualité de ressources utilisées.

- ***Maintenabilité :*** elle décrit la capacité d'un composant d'être maintenu.

- ***Portabilité :*** elle définie la capacité d'un composant d'être exploité à partir d'un environnement à l'autre, c'est-à-dire d'être opérationnel dans des environnements différents.

Dans le tableau 4.2 nous décrirons les attributs de qualité pour mesurer les sous-caractéristiques observables à l'exécution présentées dans le tableau 4.1

Sous caractéristiques	**Attributs**
Exactitude	- Exactitude
Sécurité	- Chiffrement de données - Contrôlabilité - Vérification
Récupération	- Gestion d'erreur
Tolérance aux fautes	- Disponibilité du mécanisme
	- Efficacité du mécanisme
Configurabilité	- Effort de configuration
Comportement de temps	- Temps de réponse
	- Latence : ▪ Sortie ▪ Capacité de traitement
Comportement de ressource	- Utilisation de mémoire
	- Utilisation de disque
Scalabilité	- Capacité de traitement
Stabilité	- Modifiabilité
Déployabilité	- Niveau de complexité

Tableau 4.2 : Les attributs de qualité relatifs au sous caractéristiques observables à l'exécution.

- ***Exactitude :*** évalue le pourcentage des résultats obtenus avec la précision spécifiée par les exigences de l'utilisateur.

- ***Chiffrement de données :*** exprime les capacités de chiffrement d'un composant afin de protéger les données.

- ***Contrôlabilité :*** indique comment le composant peut contrôler l'accès à ses interfaces fournies.

- ***Vérification :*** montre si un composant implémente n'importe quel mécanisme de vérification, avec les possibilités pour enregistrer l'accès d'utilisateurs au système et à ses données.

- ***Gestion d'erreur :*** indique si le composant peut manipuler des situations d'erreur, et le mécanisme implémenté.

- ***Disponibilité du mécanisme :*** indique le mécanisme de tolérance aux fautes implémenté dans le composant.

- ***Efficacité du mécanisme :*** mesure l'efficacité réelle du mécanisme de tolérance aux fautes disponible dans le composant.

- ***Effort de configuration :*** mesure l'effort pour que le composant soit configuré.

- ***Temps de réponse :*** mesure le temps nécessaire à l'exécution d'une demande.

- ***Latence :***

- ***Sortie :*** mesure le rendement qui peut être produit avec succès dans une période de temps.
- ***Capacité de traitement :*** mesure la quantité de l'information d'entrée qui peut être traitée avec succès par le composant dans une période de temps indiquée.

- ***Utilisation de mémoire :*** indique la quantité de mémoire nécessaire pour le fonctionnement d'un composant.

- ***Utilisation de disque :*** indique l'espace disque employé par un composant.

- ***Capacité de traitement :*** indique la capacité et le comportement d'un composant pour un volume plus important de données avec la même implémentation.

- ***Modifiabilité :*** indique le comportement d'un composant après une modification.

- ***Niveau de complexité :*** spécifie l'effort pour déployer un composant dans un environnement indiqué.

Les attributs de qualité observables pendant le cycle de vie sont récapitulés dans le tableau 4.3.

Sous caractéristiques	Attributs
Convenance	- Assurance
	- Perfection
	- Pré condition et post condition
	- Preuve des pré et post condition
Interopérabilité	- Compatibilité des données
Conformité	- Standarisation
	- Certification
Contenu individuel	- Dépendabilité
Maturité	- Volatilité
	- Déplacement d'échec
Compréhension	- Disponibilité de documentation
	- Qualité de documentation
Opérabilité	- Niveau de complexité
	- Interfaces fournies
	- Interfaces requises
	- Effort pour fonctionner
Testabilité	- Test de composant dans un environnement spécifique
	- Preuves de composants
Adaptabilité	- Mobilité
	- Capacité de configuration
Réutilisation	- Niveau d'abstraction du domaine
	- Compatibilité des architectures
	- Modularité

Tableau 4.3 : Les attributs de qualité relatifs au sous caractéristiques observables pendant le cycle de vie.

- ***Assurance :*** essaye de mesurer quelle quantité de fonctionnalité exigée est couverte par l'implémentation d'un composant.

- ***Perfection :*** il est possible que quelques réalisations ne couvrent pas complètement les services indiqués. Cet attribut mesure le nombre d'opération implémentées comparées au nombre d'opérations indiquées.

- ***Pré condition et post condition :*** indique si le composant a des pré et post conditions afin de déterminer plus exactement ce que le composant exige et fournit.

- ***Preuve des prés et post condition :*** indique si les prés et les post conditions sont formellement prouvés afin de garantir l'exactitude des fonctionnalités composantes.

- ***Compatibilité des données :*** indique si le format des données manipulées par le composant est conforme avec n'importe quelle norme ou convention internationale.

- ***Standarisation :*** indique si le composant est conforme aux normes internationales.

- ***Certification :*** indique si le composant est certifié par n'importe quelle organisation interne ou externe.

- ***Dépendabilité :*** indique si le composant dépend d'un autre composant.
- ***Volatilité :*** mesure le temps moyen entre les versions commerciales.
- ***Déplacement d'échec :*** indique le nombre de bugs fixés dans une version composante donnée.
- ***Disponibilité de documentation :*** cet attribut est en relation avec la documentation, les descriptions les API, qui ont un impact direct sur la compréhension du composant.
- ***Qualité de documentation :*** indique la qualité des documents fournis avec le composant.
- ***Niveau de complexité :*** indique la capacité de l'utilisation d'un composant.
- ***Interfaces fournies :*** compte le nombre d'interfaces fournies par le composant comme mesure indirecte de sa complexité.
- ***Interfaces requises :*** compte le nombre d'interfaces que le composant exige d'autres composants pour fonctionner.
- ***Effort pour fonctionner :*** montre le nombre moyen d'opérations par interface fournie.
- ***Test de composant dans un environnement spécifique :*** indique dans quels environnements ou plateformes un composant a été examiné.
- ***Preuves de composants :*** indique si le composant était formellement testé.
- ***Mobilité :*** indique dans quels conteneurs ces composants ont été déployés et dans quels conteneurs ces composants ont été transférés.
- **Capacité de configuration :** indique le pourcentage des changements requis pour transférer un composant à d'autres environnements.
- **Niveau d'abstraction du domaine :** mesure le niveau d'abstraction d'un composant lié à son domaine d'affaires.
- **Compatibilité des architectures :** indique le niveau de la compatibilité d'une architecture donnée.
- **Modularité :** indique le niveau de modularité du composant, s'il a des modules, des paquets ou tous les fichiers source sont groupés.

2- Aide multicritère à la décision

L'approche Multicritère de l'aide à la décision, qui a connu un essor important au cours de ces dernières années, apporte des éléments de réponse au problème de l'évaluation selon de multiples points de vue contradictoires. Elle consiste en la définition et l'étude approfondie de

critères supposés refléter les différents points de vue entrant en considération dans un processus de décision. Nous parlerons à cet effet d'Aide Multicritère à la Décision.

Dans ce domaine l'effort a principalement porté sur les problèmes de choix ou de rangement, c'est-à-dire comment sélectionner les meilleures solutions ou comment ordonner un ensemble de solutions de la meilleure à la moins bonne.

Nous décrivons par la suite les démarches d'une méthode d'analyse multicritères, afin de faire le choix d'un meilleur assemblage de composants en termes de qualité selon des valeurs des attributs de qualité qui sont des critères sur lesquels sera basée l'analyse. Les valeurs des attributs de qualité sont définies auparavant.

3- Méthodologie- démarche

Nous définissons dans cette section les démarches de la méthode d'analyse multi-critères « **le tableau multi-critères** » :

Le principe de la méthode est de compléter un tableau à double entrée, comportant les solutions, les critères et leur poids.

- En colonnes : les solutions, actions ou problèmes à traiter
- En lignes : les critères de sélection en indiquant le coefficient de pondération

1. Quels sont les indicateurs (critères) qui permettent d'évaluer la solution la mieux adaptée ?

2. Quel est le poids de chaque indicateur (critères) ? :

Définir avec le groupe les critères de choix et établir en groupe un barème de pondération :
Par exemple :

- Critère très important : 3
- Critère important : 2
- Critère moyennement important : 1

3. Dans quelle mesure chaque solution satisfait-elle les exigences de chaque critère ?

- Inscrire dans chaque case, les notes données par chaque membre du groupe. Toutes les actions, solutions ou problèmes doivent tous être notés pour l'ensemble des critères.
- Totaliser les notes dans chaque case et multiplier le résultat par le coefficient de pondération
- Effectuer le total de chaque colonne

La solution, le problème ou l'action qui totalise le plus grand nombre de points sera retenu.

L'idée d'avoir un processus systématique d'évaluation et de sélection de composants a été introduit en 1982 par I. Brownstein et N. Lerner pour les paquetages logiciels **[BLE82].**
Brownstein et Lerner incluaient dans leur processus des phases de spécification des besoins, d'identification des activités nécessaires à la sélection des paquetages candidats, et d'estimation des ressources nécessaires pour cette évaluation. L'idée a été reprise par d'autres travaux, avant d'être améliorée en 1991 par G. Subramanian et M. Gershon**.** Ceux-ci ont introduit les phases de définition des critères et d'évaluation des candidats sur la base de ces critères. F.Williams a introduit des phases de hiérarchisation des critères et d'identification des candidats potentiels. D'autres travaux ont mis l'accent sur les critères de qualité **[GEO07].**

Deux travaux se distinguent en particulier, ce sont les processus de sélection de composants COTS (Commercial Off-The-Shelf) les plus complets. OTSO (Off-The-Shelf-Option) est un travail pionnier dans la sélection de COTS en 1995 **[KCL95].** Il a innové aussi bien dans la définition des différentes étapes de sélection, que dans la préconisation de critères d'évaluation particuliers, ou dans le choix des méthodes de prise de décision multicritères pour l'évaluation des candidats. PORE (Procurement-Oriented Requirements Engineering) **[MNM97],** a traité simultanément la spécification des besoins et la progressivité de la sélection. D'autres processus, certains s'attachent à redéfinir les différentes étapes de la sélection, on peut citer : CEP**,** PECA**,** Six Sigma**,** CISD**,** IIDA**,** IusWare. Et d'autres innovent dans la définition des critères, soit en proposant de nouvelles manières de les définir, soit en proposant un ensemble de critères particuliers, on peut citer : Delta**,** BAREMO**,** SSEF**,** CAP**,** CBCPS**,** STACE**,** RCPEP**,** DesCOTS **[GEO07].**

4- Le processus d'évaluation de la qualité d'un assemblage de composant

4.1- Etape des exigences de qualité

L'objectif de cette première étape est de spécifier les exigences en termes de facteurs de qualité. Ces exigences peuvent varier d'un système logiciel à un autre selon le contexte d'utilisation de système.

4.2- Préparation de l'évaluation

À ce niveau, l'objectif est d'initier l'évaluation et de mettre au point ses bases. Ceci est fait en trois sous-étapes.

- **La sélection des critères d'appréciation avec leurs poids:** les critères d'appréciation doivent correspondre aux facteurs, en utilisant le modèle de qualité ISO9126 adapté aux composants logiciels, l'assembleur attribue à chaque critère un poids selon son importance. Il est possible aussi de prendre en compte dans ce processus des aspects de gestion tels que le temps ou les coûts.
  Exemple : Soient les deux critères d'évaluation du facteur adaptabilité.
  1- Mobilité avec un poids (coefficient de pondération) égal à 3
  2- Capacité de configuration avec un poids, moins important, égal à 1.
- **La définition des taux de satisfaction :** Pour chaque valeur de qualité des critères, on définit un taux de satisfaction correspondant à la valeur qualité attribuée par le concepteur du composant. Les taux sont mesurés à partir des tests sur le composant pour chaque critère de qualité, Les données de tests sont construites à partir des spécifications du comportement de composant.

- **La détermination des valeurs qualité des critères :** on détermine des valeurs qualité des critères pour chaque composant, à partir des valeurs qualité de chaque critère pour chaque composant et leur taux de satisfaction correspondant.

4.3- La qualité des attributs

a) La qualité des attributs d'un composant

Nous avons indiqué dans les étapes du processus d'évaluation que les valeurs de qualité des critères des composants sont données par le concepteur de ces derniers, mais les taux de satisfaction sont attribués à ses valeurs sur la base des tests effectués sur le composant.
Les critères dont les métriques sont de type présence ou dénombrement sont transformés en valeurs de type ratio par l'assembleur
Soit :
- $\mathbf{VQA_{pj}}$ la valeur de qualité du critère **j** du composant **p**,
-$\mathbf{TAUX_{pj}}$ le taux de satisfaction attribué à la valeur $\mathbf{VQA_{pj}}$ à partir des tests effectués.
Alors : **VQApj ← VQApj* TAUXpj**

Nous utilisons deux principes, en fonction de la nature du critère, pour définir les valeurs **TAUXpj** selon la valeur $\frac{K}{T}$.
Tels que : **T** le nombre de tests effectué et **K** le nombre de tests vérifiant la valeur donné par le concepteur. $\mathbf{K \leq T}, \boldsymbol{0 \leq \frac{K}{T} \leq 1}$

Le principe 1 est utilisé dans le cas où la valeur $\frac{K}{T}$ affecte positivement la qualité du critère (**Tableau 1)**,
Le principe 2 est utilisé dans le cas où la valeur $\frac{K}{T}$ affecte négativement la qualité du critère (**Tableau 2)**,

La valeur $\frac{K}{T}$	[0,0.15 [	[0.15, 0.35 [	[0.35, 0.65 [	[0.65, 0.85 [	[0.85,1]
La valeur TAUXpj	0	0.25	0.50	0.75	1

Tableau 1.Application du principe 1.

La valeur $\frac{K}{T}$	[0,0.15 [	[0.15, 0.35 [	[0.35, 0.65 [	[0.65, 0.85 [	[0.85, 1]
La valeur TAUXpj	1	0.75	0.50	0.25	0

Tableau 2. Application du principe 2.

b) La qualité des attributs d'un assemblage

Pour produire la qualité d'un critère **j** d'un assemblage **Ai**, le concept ROC (Rank Order Centroids) **[36]** est utilisé. Les centroides de classement constituent un moyen de convertir des rangs (1^{ier}, 2^{eme}, 3^{eme}) en notes ou pondérations qui sont des valeurs numériques.
Si n est le nombre de composants, la pondération du composant k est :

$$(\sum_{i=k}^{n}(1/i))/n$$

La production de la qualité du critère **j** d'un assemblage **Ai** implique :

1. **Le classement de l'ensemble des composants selon leur importance.**
2. **La conversion des rangs en pondération à l'aide de ROC.**

Exemple : Soit

- Un assemblage **A1** composé de trois composants C_{11}, C_{12}, C_{13},
- V_{11}, V_{12}, V_{13} des valeurs de qualité du critère j des composants C_{11}, C_{12}, C_{13},
- On considère que le composant C_{11} est le composant le plus important, suivi de composant C_{12} puis le composant C_{13}.

Les pondérations des composants sont calculées comme suit :

Le composant C11 :**W1= (1+1/2+1/3)/3=0.6111.**
Le composant C12 :**W1= (0+1/2+1/3)/3=0.2778.**
Le composant C13 :**W1= (0+0+1/3)/3=0.1111.**

Alor la qualité de l'attribut j de l'assemblage A1 :

$\mathbf{VQA_{j1}}$= (V_{11}*0.6111) + (V_{12}*0.2778) + (V_{13}*0.1111).

4.4- Spécification de la qualité des composants logiciels

Actuellement, il n'y a pas de travaux concernant la spécification de la qualité des composants logiciels.
L'existant concerne uniquement la QoS. Le langage QML (QoS Modelling Langage) permet de décrire la contrainte de QoS des composants logiciels ; mais il n'est pas possible de spécifier que le service pourra réellement fournir.

Dans notre cas nous utilisons le langage XML pour la spécification de la qualité des composants (La figure 4.1), nous proposons que chaque composant possède un fichier XML qui définit les valeurs qualité des attributs.

```
- <COMPONENTS>
  - <CHARACTERISTIC>
      <INTITULE_CHARACTERISTIC>Portabilite</INTITULE_CHARACTERISTIC>
    - <SOUS_CHARACTERISTIC>
        <INTITULE_SOUS_CHARCTS>Reutilisabilite</INTITULE_SOUS_CHARCTS>
      - <ATTRIBUTS>
          <INTITULE_ATTRIBUT>Niveau d'abstraction domaine</INTITULE_ATTRIBUT>
          <QUALITY_ATTRIBUT>0,35</QUALITY_ATTRIBUT>
        </ATTRIBUTS>
      - <ATTRIBUTS>
          <INTITULE_ATTRIBUT>Compatibilite des architecture</INTITULE_ATTRIBUT>
          <QUALITY_ATTRIBUT>0,55</QUALITY_ATTRIBUT>
        </ATTRIBUTS>
      - <ATTRIBUTS>
          <INTITULE_ATTRIBUT>Modularite</INTITULE_ATTRIBUT>
          <QUALITY_ATTRIBUT>0,75</QUALITY_ATTRIBUT>
        </ATTRIBUTS>
      </SOUS_CHARACTERISTIC>
    </CHARACTERISTIC>
  - <CHARACTERISTIC>
      <INTITULE_CHARACTERISTIC>Utilisabilite</INTITULE_CHARACTERISTIC>
    - <SOUS_CHARACTERISTIC>
        <INTITULE_SOUS_CHARCTS>Comprehension</INTITULE_SOUS_CHARCTS>
      - <ATTRIBUTS>
          <INTITULE_ATTRIBUT>Disponibilite de documentation</INTITULE_ATTRIBUT>
          <QUALITY_ATTRIBUT>1,00</QUALITY_ATTRIBUT>
        </ATTRIBUTS>
      - <ATTRIBUTS>
          <INTITULE_ATTRIBUT>Qualite de documentation</INTITULE_ATTRIBUT>
          <QUALITY_ATTRIBUT>0,5</QUALITY_ATTRIBUT>
        </ATTRIBUTS>
      </SOUS_CHARACTERISTIC>
    </CHARACTERISTIC>
  </COMPONENTS>
```

Figure 4.1 : Spécification de la qualité d'un composant par un fichier XML.

4.5- Illustration

Pour illustrer l'approche pour la sélection du meilleur assemblage, nous considérons.

- deux solutions ou deux assemblages (d'un seul composant), l'assemblage A1(le composant C11) et l'assemblage A2 (le composant C21).
- La compréhension et la réutilisabilité sont les facteurs retenus de A1 et A2,
- On considère que la réutilisabilité est le facteur le plus important (pondération égale à 3), et la compréhension, le facteur moyennement important (pondération égale à 1),

Maintenant, nous donnons les pondérations pour chaque ensemble d'attributs concernant les facteurs : réutilisabilité et compréhension.

- La réutilisabilité est estimée par :
  1. Niveau d'abstraction du domaine
  2. Compatibilité des architectures
  3. Modularité

La modularité a la plus grande importance (pondération égale à 3), suivi de la compatibilité des architecteurs (pondération égale à 2) et de niveau d'abstraction du domaine (pondération égale à 1).

- La compréhension est évaluée par :
  1. Disponibilité de la documentation
  2. Qualité de la documentation

Nous retenons que la disponibilité de documentation est la plus importante (pondération égale à 3), suivie de la qualité de documentation (pondération égale à 2).

Nous notons que l'attribut « Disponibilité de documentation » est estimé par des métriques de type présence, et l'attribut « Qualité de documentation » est estimé par des métriques de type niveau (très bas(0), bas(1), moyen(2), haut(3), très haut(4)). L'assembleur transforme ces deux métriques en métriques de type ratio.

L'application de la méthode d'analyse multi-critères donne le tableau suivant :

		Assemblage A1			Assemblage A2		
Facteurs	**Critères**	Critères fournis par le concepteur	Critères recalculés		Critères fournis par le concepteur	Critères recalculés	
			Métriques*			Métriques*	
la réutilisabilité	Niveau d'abstraction du domaine : **Pondération : 1**	**0.35** $\frac{K}{T} \in [0.85, 1]$	M11 : 0.18 M12 : 0.21	0.18+0.21 = **0.39**	**0.40** $\frac{K}{T} \in [0.65,0.85[$	M11 :0.20 M12 :0.10	0.20+0.10= **0.30**
	Compatibilité des architectures **Pondération : 2**	**0.55** $\frac{K}{T} \in [0.65,0.85[$	M21 :0.15 M22 :0.08 M23 :0.17	0.15+0.08 +0.17= **0.40**	**0.60** $\frac{K}{T} \in [0.85, 1]$	M21 :0.11 M22 :0.35 M23 :0.19	0.11+0.35+0. 19= **0.65**
	Modularité **Pondération : 3**	0.75 $\frac{K}{T} \in [0.85, 1]$	M31 :0.22 M32 :0.58	0.22+0.52 = **0.80**	**0.80** $\frac{K}{T} \in [0.35,0.65[$	M31 :0.20 M32 :0.40	0.20+0.40 = **0.60**
la compréhension	Disponibilité de documentation **Pondération : 3**	**1.00** $\frac{K}{T} \in [0.85, 1]$	M41 :1.00	**1.00**	**1.00** $\frac{K}{T} \in [0.85, 1]$	M41 :1.00	**1.00**
	Qualité de documentation **Pondération : 2**	0.50 $\frac{K}{T} \in [0.35,0.65[$	M51 :0.45	**0.45**	0.75 $\frac{K}{T} \in [0.35,0.65[$	M51 :0.50	**0.50**
	Valeurs des facteurs						
	la réutilisabilité **Pondération : 3**	(0.35***1***1)+(0.55***0.75***2)+(0.75***1***3)/6 =0.50			(0.40***0.75***1)+(0.60***1***2)+(0.80***0.5***3)/6 =0.45		
	la compréhension **Pondération : 1**	(1***1***3)+(0.50***0.50***2)/5=0.70			(1***1***3)+(0.75***0.50***2)/5=0.75		
	La qualité globale	(0.50*3) + (0.70*1)/4 =0.55			(0.45*3) + (0.75*1)/4 =0.52		

Dans le cas dont l'assemblage est constitué de plusieurs composants, les valeurs qualité des critères sont calculées par le concept ROC.

4.6- Architecture de logiciel

Cette partie montre l'implémentation constituée de deux parties : la première partie illustre le schéma de la base de données et la deuxième partie, montre le calcul de la valeur globale de la qualité des assemblages,

a) Schéma de la base de données

La base de données utilisé dans l'implémentation est sous ORACLE dont la version 10G, transactionnelle. les tables de la base contient les données pour les valeurs qualité des attributs des composants, et les valeurs qualité des facteurs, ainsi la valeur de la qualité globale des assemblages, on peut à tout moment, la consultation de ses valeurs pour faire l'analyse afin de choisir le meilleur assemblage en terme de qualité.

Le schéma de la base est la suivante :

* sont des métriques permettent de calculer les critères, en utilisant la fonction linéaire Cj, tel que b1=b2=b3=1.

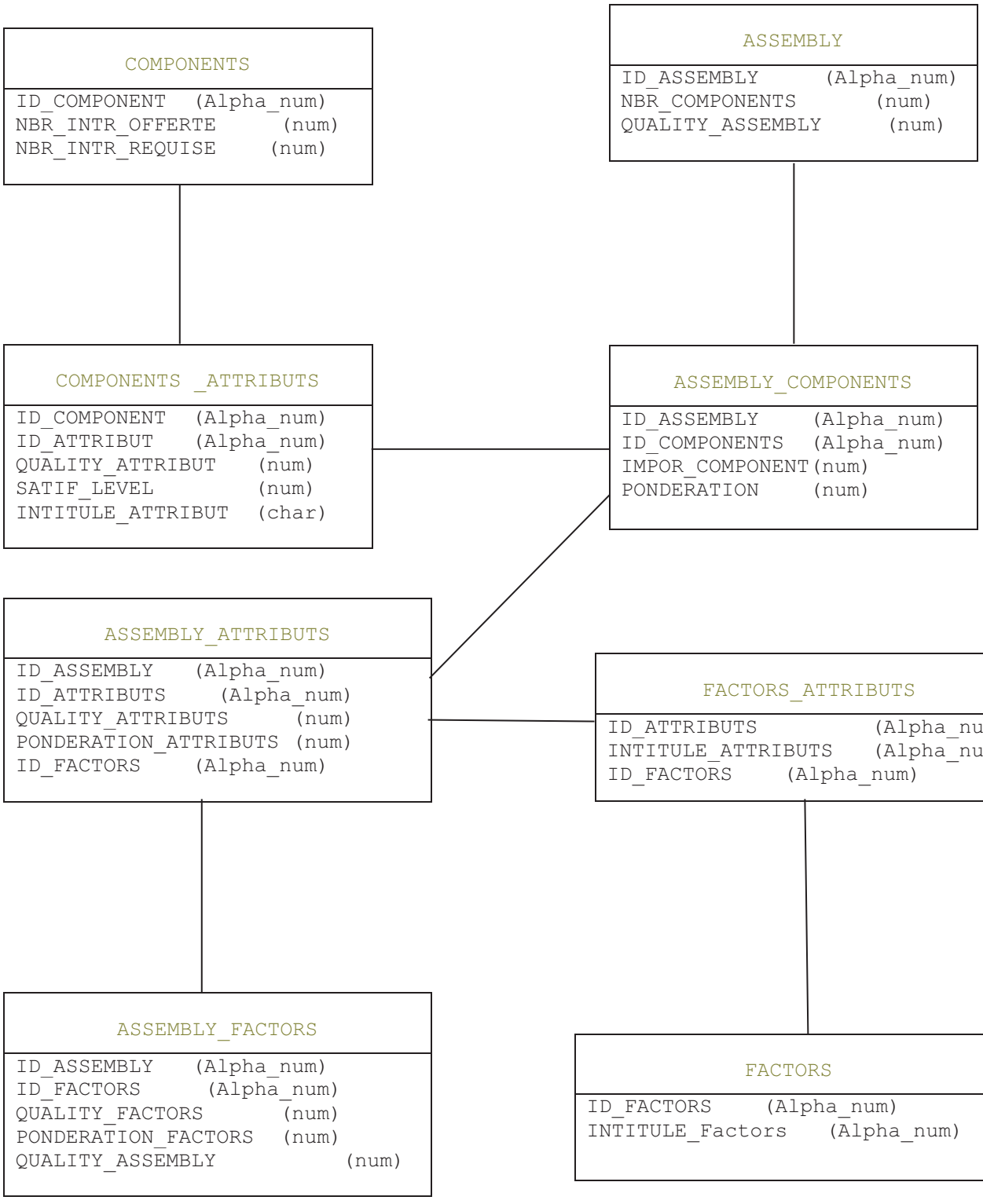

Figure 4.2 : Le schéma de la base de données.

b) L'interface principale

A travers de l'interface de logiciel on peut lancer les différente traitements.

Figure 4.3 : L'interface principale.

1) Le menu principal

Le menu principal est un menu contextuel présentant l'ensemble des options qui orienté vers l'utilisation des différents écrans.

2) Le menu

Afin d'assurer la convivialité dans le logiciel, chaque écran comporte un menu standard offrant des options d'accès à l'information ainsi que les options standards de Windows. Les mêmes options sont prévues dans la barre d'outils sous forme de boutons comportant des icônes représentatives des fonctions qui leurs sont associés.

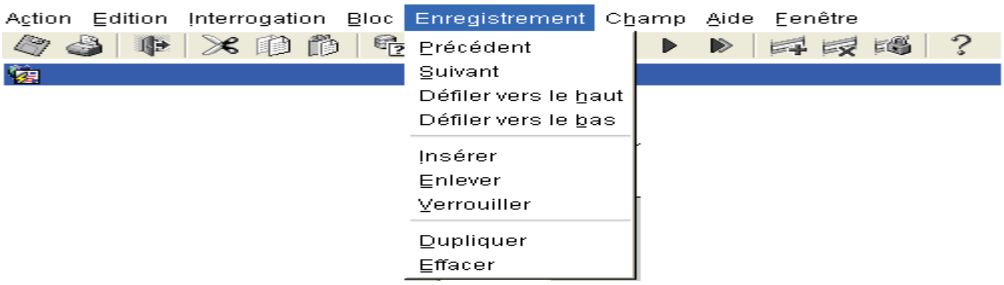

Figure 4.4 : Le menu contextuel.

3) Fonctionnalité

Le système permet :

- la saisie des valeurs qualité des attributs pour chaque composant, les donné sont stocké dans la base.
- La construction des assemblages, le choix sur fait sur les composants saisis avant,
- La définition de l'importance de chaque composant dans l'assemblage, le calcul des pondérations des composants se fait automatique par le système après validation.
- Le calcul des valeurs qualité des attributs de l'assemblage, se fait automatique (l'application de principe ROC) ;
- Après la détermination des pondérations et les taux de satisfaction pour les attributs de l'assemblage, le système fait le calcul des valeurs qualité des facteurs,
- Le calcul de la qualité globale de l'assemblage après la détermination des pondérations des facteurs de qualité.

Toutes les valeurs sont stocké dans la base de données, on peut les consultés. Et après l'analyse de ses valeurs, on peut faire le choix sur le meilleur assemblage.

Dans le logiciel trois 03 type d'écran :

- Type Formulaire
- Type Maitre détail
- Type tabulaire

a) Ecran de saisie mise à jour les attributs des composants

Cet écran permet de définir les attributs des composants avec des valeurs qualité et leurs taux de satisfaction. Les attributs et leurs facteurs sont stockés dans la base (Menu principal → Informations de bases)

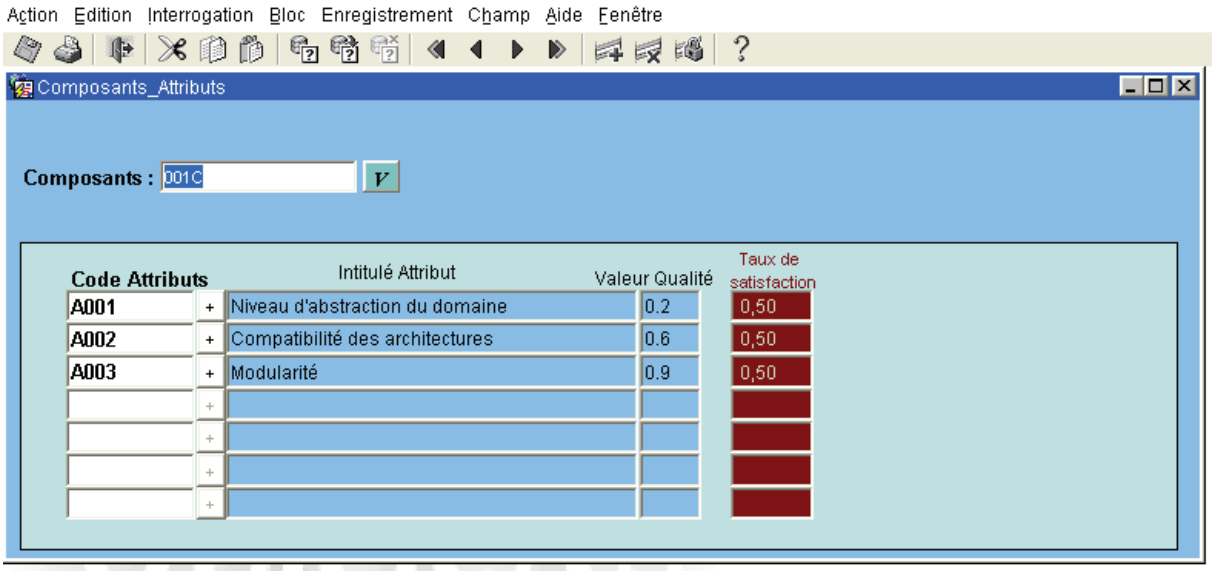

Figure 4.5 : Définition des attributs de composant.

b) Ecran de saisie mise à jour des composants de l'assemblage

Cet écran permet de définir les composants de l'assemblage afin de calculer leurs pondérations (par le concept ROC), ainsi le calcul de la qualité des attributs pour l'assemblage.

La qualité globale de l'assemblage s'affiche après le calcul de la qualité des facteurs correspondant aux attributs des composants constitue l'assemblage.

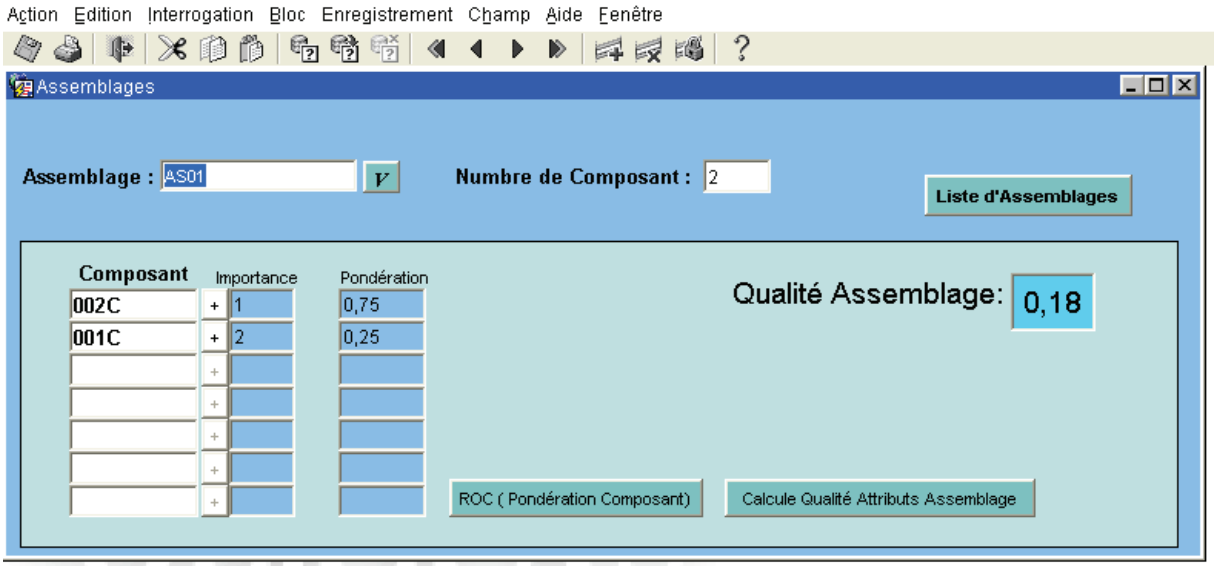

Figure 4.6 : Définition des composants d'assemblage.

On cliquant sur le bouton « Calcul Qualité Attributs Assemblage » le masque suivant s'affiche :

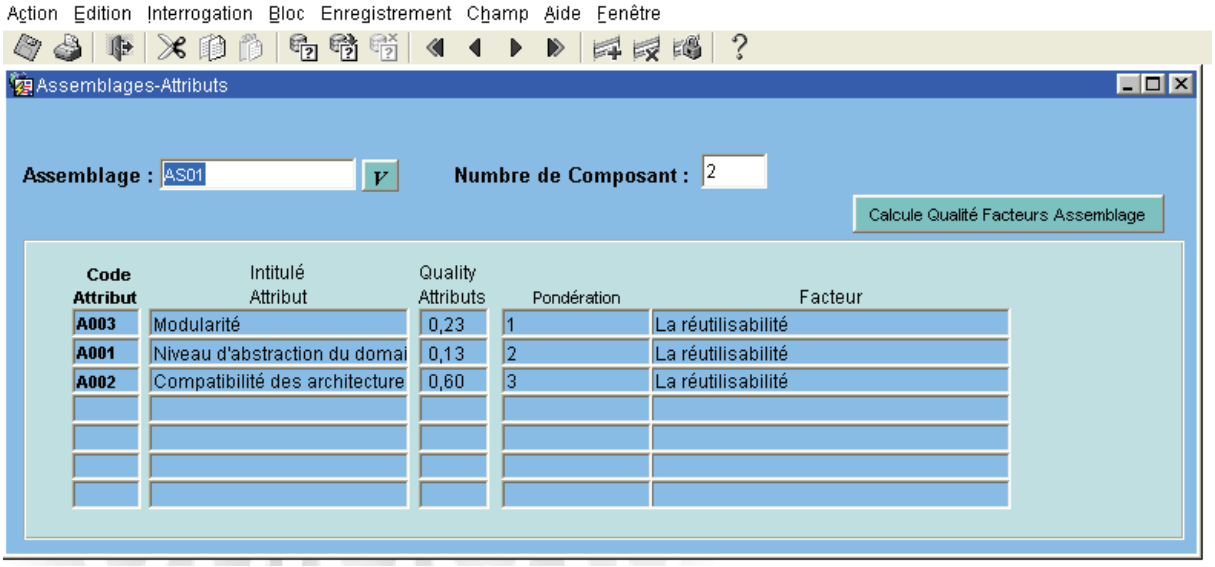

Figure 4.7 : Définition des attributs d'assemblage.

Apres la définition des pondérations des attributs de l'assemblage, on lance le calcul de la qualité des facteurs de l'assemblage (Figure 4.8).

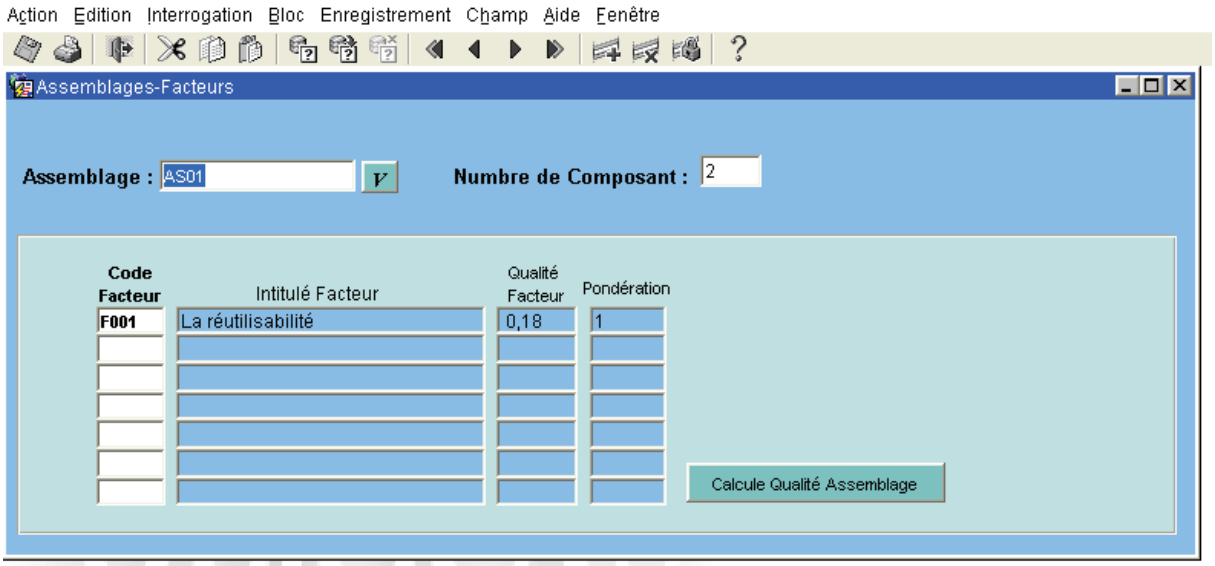

Figure 4.8 : Calcul de la valeur globale de la qualité.

Le calcul de la qualité globale de l'assemblage (Figure 4.6), se fait après la définition des pondérations des facteurs on cliquant sur le bouton « Calcule Qualité Assemblage ».

Apres le calcul de la qualité de chaque assemblage, on fait le choix sur le meilleur assemblage on utilisant l'écran liste d'assemblages avec leur qualité.

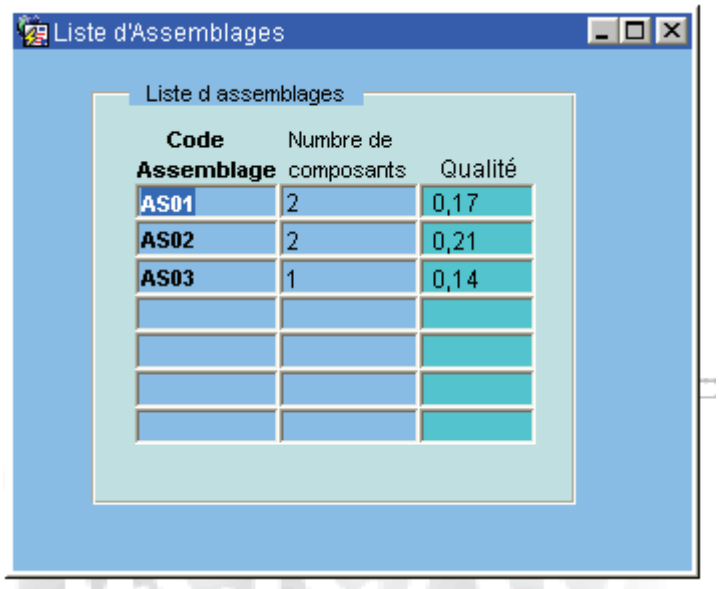

Figure 4.9 : Liste d'assemblages avec leur qualité.

CONCLUSION GENERALE

Conclusion et perspectives

Nous concluons que les composants logiciels sont devenus des éléments architecturaux incontournables dans le monde du logiciel. Ils doivent permettre d'assimiler à terme le développement du logiciel à un simple processus d'assemblage. Cet objectif ne pourra être atteint qu'à la condition de disposer de composants logiciels conçus comme de véritables unités de déploiement fiables, performantes et parfaitement décrites tant du point de vue de leurs propriétés fonctionnelles que de leurs propriétés non-fonctionnelles.

Le paradigme composant a introduit la méthode CBSE, comme une nouvelle méthode pour la conception des applications logicielles. Cette méthode a pour objectif le développement de gros logiciels par l'intégration de composant existants.

Nous concluons également que dans un processus de développement à base de composant, l'assemblage et la sélection d'une composition devant un choix multiple dont la qualité est le plus optimale en fonction des besoins de l'utilisateur, restent des problèmes complexes.

Notre objectif étant de trouver une solution qui permettra de choisir la meilleure composition devant un choix multiple, et pour atteindre cet objectif, nous avons présenté un processus qui permet de faire la sélection, non seulement, d'un meilleur composant, mais aussi d'une meilleure composition. Ce processus est basé sur les valeurs des éléments de qualité (facteurs, critères et attributs) de chaque composant. Pour faire, il était nécessaire d'apposer sur une technique dite de prise de décision multicritère.

Dans cette étude nous nous sommes donnés certaines limites : utilisation des valeurs qualité des attributs données par le concepteur de composant. En perspective, il serait intéressant de baser sur le code source des composants de type boite blanche ou grise pour mesurer des attributs de qualité. On envisage également d'utiliser les documents de conception (cas d'utilisation) ou encore des informations supplémentaires fournies par l'utilisateur (par exemple recommandation sur le nombre de composants) ou par le concepteur du système (style architectural par exemple).

BIBLIOGRAPHIE

Bibliographie

[ABD04]: A. Abran, P. Bourque, R. Dupuis, and L.L. Tripp. Guide to the software engineering body of knowledge (ironman version). Technical report, IEEE Computer Society, 2004.

[ACC01]: Projet ACCORD, « Assemblage de composants par contrats Etat de l'art et de la standardisation », Livrable 1.1-1, France. Juin 2002.

[ACC02]: Projet ACCORD, « Etat de l'art sur les Langages de Description d'Architecture ADLs », Livrable 1.1-2, Juin 2002, France.

[ACC03] : Projet ACCORD, « Modèle abstrait d'assemblage de composant par contrats», Livrable 1.4, Juin 2003, France.

[ACC94]: Shyam R. Chidamber and Chris F. Kemerer. A metrics suite for object oriented design. IEEE Transactions on Software Engineering, 20(6):476–493, June 1994.

[BBD09]: F.Balmas, A. Bergel, S.Denier, S. Ducasse, J. Laval, K.Mordal-Manet, H.Abdeen, and F. Bellingard. Software metric for java and c++ practices (squale deliverable 1.). Technical report, INRIA Lille Nord Europe, 2009.

[BBF08]: T. Bakota, A. Beszédes, R. Ferenc, and T. Gyimóthy. Continuous software quality supervision using sourceinventory and columbus. In Companion of the 30th international conference on Software engineering, ICSE Companion '08, pages 931–932, New York, NY, USA, 2008. ACM.

[BCR94]: V. Basili, G. Caldiera, and H. Rombach Dieter. The goal question metric approach. In Encyclopedia of Software Engineering. Wiley, 1994.

[BDA02]: J. Bansiya and C. Davis. A hierarchical model for object-oriented design quality assessment. IEEE Transactions on Software Engineering, 28(1):4–17, January 2002.

[BEU05]: A.Beugnard, « Contribution à l'étude de l'assemblage de logiciels », thèse de doctorat, l'université de Bretagne Sud. Décembre 2005.

[BLE82]: I. Brownstein, N. Lerner. Guidelines for Evaluating and Selecting Software Packages. Elsevier Science Publishing Co., Inc., 1982.

[CER04]: H. Cervantes, « Vers un modèle à composants orienté services pour supporter la disponibilité dynamique », thèse de doctorat, Mars 2004, Université Joseph Fourier, Grenoble 1.

[CHO09]: Abdellah CHOUARFIA, Mebarka YAHLALI «Software Components Assembly with an Appreciated QoS » in journal of software, Vol 4, N° 6, pp 516-520,2009.

[DES08]: N. Desnos, « Ports composites pour l'assemblage autonomique de composants logiciels, Application à la construction dynamique et à l'évolution non anticipée », THÈSE pour obtenir le grade de DOCTEUR DE L'UNIVERSITE MONTPELLIER II, Discipline : informatique, Ecole Doctorale : I2S, 23 juin 2008.

[DLR00]: M.Dalmau, S.Laplace, P.Roose, « Méthode de choix d'une configuration de composants logiciels optimale guidée par l'évaluation de la qualité de service » LIUPPA - Département Informatique de l'I.U.T. de Bayonne.

[FLP96]: N. Fenton, S. Lawrence Pfleeger. Software Metrics: A Rigorous and Practical Approach. International Thomson Computer Press, London, UK, second edition, 1996.

[GAR00]: D. Garlan. Software architecture: a roadmap. ICSE - Future of SE Track, 2000.

[GEO07]: B.George, «Un processus de sélection de composants logiciels multi-niveaux», thèse pour obtenir le grade de docteur de l'université Bretagne Sud, Laboratoire valoria, 2007.

[GPA00]: D.Galin, P.Patton, «Introduction à l'assurance de Qualité Logicielle ».

[HAD03]: S. HADJAB, Assemblage de Composants Logiciels : Les Composants-Composites, Mémoire de stage pour l'obtention du Diplôme d'Études Approfondies, Université de Technologie de Troyes – UTT, 11 juillet 2003.

[HME09]: Les composants logiciels.pdf, cours pour post-graduation proposé par Mme R. HARIR MEKKI, 2009.

[HME08]: HARIR MEKKI Rachida, « Mobilité et interopérabilité des composants logiciels », THÈSE pour obtenir le diplôme de DOCTORAT EN SCIENCES, Discipline : informatique, USTOran, 2008.

[HPR11]: C.Hernán Prada Rojas, « Une approche à base de composants logiciels pour l'observation de systèmes embarqués » THÈSE Pour obtenir le grade de docteur de l'université de Grenoble, Spécialité : Informatique, juin 2011.

[ISO01]: ISO/IEC. Iso/iec 9126-1 software engineering -product quality- part 1: Quality model, 2001.

[ISO05]: ISO/IEC. Iso/iec 25000:2005 software engineering -software product quality requirement and evaluation, 2005.

[JON08]: C. Jones. Applied Software Measurement: Global Analysis of Productivity and Quality. McGraw-Hill, Inc., Hightstown, NJ, USA, 2008.

[KBC98]: R. Kazman, L. Bass, P. Clements. Software Architecture In Practice. Book, Addison Wesley,1998.

[KCL95]: J. Kontio, S. Chen, K. Limperos, R. Tesoriero, G. Caldiera, and M. Deutsch. A cots selection method and experiences of its use. In Proceedings of the 20th Annual Software Engineering Workshop (NASA), Greenbelt, Maryland, USA, 1995.

[LKI94]: M. Lorenz and J. Kidd. Object-Oriented Software Metrics: A Practical Guide. Prentice-Hall, 1994.

[LMA06]: M. Lanza, R. Marinescu. Object-Oriented Metrics in Practice. Springer-Verlag, 2006.

[MAR06]: B. Marvray, NASA Goddard Space Flight Center. Procedure for developing and implementing software quality programs, last update 2006.

[MAR97]: R.C. Martin. Stability, 1997. www.objectmentor.com.

[MEN04]: V. Mencl. Specifying Component Behavior with Port State Machines. Electr. Notes Theor. Comput. Sci., vol. 101, pages 129-153, 2004.

[MNM97]: N.Maiden, C.Ncube, A.Moore. Lessons learned during requirements acquisition for cots systems. Communications of the ACM, 1997.

[MPA05]: V. Mencl, F. Plasil, J. Adamek. Behavior Assembly and Composition of Use Cases - UML 2.0 Perspective, in Proceedings of the Software Engineering (SE) 2005 conference, Feb. 15-17, 2005, Innsbruck, Austria, isbn : 0-88986-466-7, ISSN 1027-2666, pages 193-201, ACTA Press, Feb 2005.

[MPR07]: H. Muccini, A. Polini, F. Ricci, A. Bertolino. Monitoring Architectural Properties in Dynamic Component-Based Systems.CBSE : pages 124-139, 2007.

[MRA04]: R. Marinescu and D. Ra¸tiu. Quantifying the quality of objectoriented design: the factor-strategy model. In Proceedings 11th Working Conference on Reverse Engineering (WCRE'04), pages 192–201, Los Alamitos CA, 2004. IEEE Computer Society Press.

[MRO06]: C.Million-Rousseau, « Test, qualité et maintenance », support de cours, 2006, Ecole Polytechnique Université de savoie.

[MRW76]: J. McCall, P. Richards, and G.Walters. Factors in Software Quality. NTIS Springfield, 1976.

[MTA00] : N. Medidovic, R.N.Taylor, A Classification and Comparaison Framework for software Architecture Description Languages, IEEE Transactions on Software Engineering, vol. 26, no. 1, Janvier 2000.

[RAD04] : B.Radu , « Evaluation de la qualité de service par l'utilisateur final dans les système mobiles », thèse de doctorat, Mars 2004,Université de Marne la Vallée, France.

[RIM06]: M. RIVEILL et P. MERLE, « Programmation par composants », support de cours, l'université de Nice-Sophia-Antipolis et l'université des sciences et technologies de Lille Laboratoire d'informatique fondamentale de Lille (LIFL).

[YAH08]: YAHLALI Mebarka, « Qualité d'un assemblage de composant logiciels », Mémoire en vue de l'obtention du diplôme de magistère, Département Informatique, USTOran, 2008.

[YAH14]: YAHLALI Mebarka, CHOUARFIA Abdellah «SCAE Software Component Assembly Evaluation » in International journal of software engineering and its applications, Vol 8, N° 3,pp 255-268, 2014.

Résumé

L'ingénierie logicielle à base de composants repose sur le principe d'assemblage de composants interagissant entre eux. Devant le choix de plusieurs compositions (assemblages) fonctionnellement identiques, se pose le problème de choix de la meilleure composition en termes de qualité. Bien évidement chaque composant a privilégié certains facteurs de qualité dans son contexte qui peuvent être en contradiction avec ceux d'un autre composant. Notre travail consiste à étudier les paramètres de qualité en présentant un processus d'évaluation qui permet de choisir, parmi plusieurs compositions, celle dont la qualité est la plus optimale ou de proposer la meilleure composition en fonction des besoins de qualité de l'utilisateur. En appliquant une technique de prise de décision multicritère, un résultat globale de l'évaluation est d'obtenu, non seulement, pour chaque composant, mais aussi pour chaque composition, ces résultats sont utilisés pour la sélection d'une meilleur composition.

Mots-clés : *Composant logiciel, Assemblage, Qualité, Modèle de qualité, Facteurs et métriques, Décision multicritères.*

Abstract

The component-based software engineering based on the assembly of components interacting between them. In front of the choice of several compositions (assemblies) functionally identical, the difficulty of choosing the best composition in terms of quality arises. Obviously each component has privileged certain quality factors in its context which can be in contradiction with those of another component. Our work is to study the quality parameters by presenting an evaluation process to choose among several compositions, from a multiple composition, one whose quality is the best or to propose the best composition according to the needs of user quality. By applying a technique of multicriteria decision, a global evaluation result is obtained, not only, for each component, but also for each composition; these results are used to select a best composition.

Keywords: *Software component, Composition, Quality, Model of quality, Factors and metrics, multicriteria Decision*