UNIVERSITE SULTAN MOULAY SLIMANE
Faculté des Sciences et Techniques
Béni-Mellal

N° d'ordre : 91/2016

Centre d'Etudes Doctorales : Sciences et Techniques
Formation Doctorales : Mathématiques et Physique Appliquées

THESE

Présentée par

Abdellatif HAIR

Pour l'obtention du grade de
Docteur
Spécialité : Informatique
Option : Génie logiciel

Développement à base de composants centré utilisateur

Soutenue le **19 Mars 2016** devant la commission d'examen

FAKIR Mohamed	**Professeur à FST -USMS- Béni Mellal**	**Président**
CHADLI Lalla Saadia	**Professeur à FST -USMS- Béni Mellal**	**Rapporteur**
EL-KIRAM My Ahmed	**Professeur à FSSM -UCA- Marrakech**	**Rapporteur**
IDRISSI Najlae	**Professeur à FST -USMS- Béni Mellal**	**Invitée**
MELLIANI Saïd	**Professeur à FST -USMS- Béni Mellal**	**Directeur de Thèse**
MINAOUI Brahim	**Professeur à FST -USMS- Béni Mellal**	**Examinateur**

Remerciements

Je souhaite remercier en premier lieu mon directeur de thèse, ***MELLIANI Saïd***, Professeur au département de Mathématiques de la Faculté des Sciences et Techniques de Béni Mellal et dirigeant du laboratoire ***LMACS*** pour m'avoir fait l'honneur de m'encadrer durant la réalisation de cette thèse. Je lui suis également reconnaissant pour le temps conséquent qu'il m'a accordé, sa franchise et sa sympathie. Je lui adresse ma gratitude pour tout cela.

Je voudrais remercier les membres de jury : ***Pr. CHADLI Lalla Saadia, Pr. EL-KIRAM My Ahmed, Pr. FAKIR Mohamed, IDRISSI Najlae,*** et ***Pr. MINAOUI Brahim*** pour l'intérêt qu'ils ont porté à mon travail.

Mes remerciements qui ne sont pas les moindres sont pour les personnes les plus chères dans ma vie, mes parents qui m'ont toujours soutenu et encouragé dans tout ce que j'entreprenais durant toute ma vie. Ce travail est le fruit de leurs patiences et sacrifices.

Je ne pourrais pas oublier ma femme qui a subi tous les désagréments durant cette période de préparation de thèse et m'a énormément soutenu et motivé.

Je pense aussi aux membres du département d'Informatique que je ne pourrais sûrement pas citer sur cette page, je pense entre autre à **EL AYYACHI, NACHAOUI, ERRITALI** et **BASLAM**. Ensuite, je pourrais ajouter beaucoup de personnes qui au bout de ces années au sein du laboratoire et au gré des rencontres tiennent forcément une place importante dans ma vie.

Ce travail n'a pu atteindre ses objectifs sans la contribution de près ou de loin de plusieurs personnes auxquelles j'adresse mes chaleureux remerciements.

Résumé

La réutilisation des parties de logiciel déjà développées pour construire de nouvelles applications présente de nombreux intérêts. La construction d'une application est donc vue, non plus comme un développement intégral et complet, mais comme un assemblage de briques de bases réutilisables (dits Composants logiciels). Dans ce domaine, les technologies supportant la construction et l'assemblage de composants ont atteint un premier niveau de maturité, notamment avec des standards tels que EJB, CCM, .NET, etc. Malgré cela, les techniques d'ingénierie logicielle spécifiques à ce domaine (dite ingénierie logicielle basée composant, appelée également la CBSE) sont encore insuffisantes et demandent plus d'efforts de recherche notamment dans les phases de modélisation des applications à composants.

Cette thèse introduit deux nouvelles approches d'assemblage de composants logiciels basées sur le concept point de vue de l'utilisateur. Le principal ajout à CBSE est celui du concept de composant multi-vues. Un composant multi-vues est un composant logiciel qui permet de stocker et restituer un sous-composant en fonction du profil de l'utilisateur. Dans le premier assemblage, un nouveau connecteur entre les composants logiciels, appelé connecteur de visibilité, a été proposé. Quant au deuxième, un patron d'assemblage centré utilisateur a été développé. Les stéréotypes, les règles et les contraintes, ajoutés dans les deux approches, ont été regroupés sous forme d'un profil UML.

Sur le plan méthodologique, cette thèse propose une démarche d'analyse et de conception qui permet d'intégrer de façon logique et consistante la notion de point de vue dans le contexte de développement à base de composant.

Le prototype support à la démarche a été réalisé en adaptant l'atelier Objecteering/UML par la technique des profils. Ce prototype permet de mener une modélisation à base de composants multi-vues en appliquant l'une des deux approches d'assemblage.

Mots clés : Composant multi-vues, point de vue, connecteur de visibilité, assemblage, patron, profil UML, CBSE.

Abstract

The reuse of already developed software parts to build new applications has many interests. The construction of an application is seen, not as a full and complete development, but as a collection of reusable bases bricks (called Software Components). In this area, the technologies supporting the construction and assembly of components have reached the first level of maturity, especially with standards such as EJB, CCM, .NET, etc. Despite this, specific software engineering techniques for this area (called component-based software engineering, also called the CBSE) is still insufficient and require more research efforts particularly in the modeling phase components applications.

This thesis introduces two new assembling approaches of software components based on the *viewpoint* concept of the user. The main addition to CBSE is that of *multi-view component* concept. A multi-view component is a software component that can store and remove a sub-component according to the user profile. In the first assembly, a new connector between software components called *visibility connector*, has been proposed. The second, a user centered an assembly pattern was developed. Stereotypes, rules and constraints, added to the two approaches have been combined in the form of a UML profile.

In terms of methodology, this thesis proposes an analysis and design approach that integrates logically and consistently the viewpoint concept in the component-based development context.

The support prototype in the process was achieved by adapting Objecteering/UML tool by technical profiles. This prototype allows to lead a modeling based on multi-views components using one of the two assembly approaches.

Keywords : Composant multi-views Component, viewpoint, visibility connector connecteur, assembling, pattern , UML profile, CBSE.

Table des matières

Introduction générale 11

Chapitre 1 : Composants logiciels 16

1.1. Introduction 16

1.2. Les concepts principaux des composants logiciels 17

1.2.1. Différentes définitions du composant 17

1.2.1. Représentation d'un composant 18

1.2.1.1. Interface 19

1.2.1.2. Propriétés 20

1.2.1.3. Contraintes 20

1.2.2. Connecteurs 20

1.2.3. Contrats 21

1.2.4. Cycle de vie 22

1.3. Quelques modèles de composants 23

1.3.1. JavaBeans EJB 24

1.3.1.1. Les Spécifications 24

1.3.1.2. Les composants 24

1.3.1.3. Déploiement et assemblage 26

1.3.2. CCM 28

1.3.2.1 Les spécifications 28

1.3.2.2. Les composants 28

1.3.2.3. Déploiement et assemblage 30

1.3.3. Microsoft .NET 30

1.3.3.1. Les spécifications 31

1.3.3.2. Les composants 31

1.3.3.3. Déploiement et assemblage 32

1.3.4. Fractal 32

1.3.4.1. Les spécifications 33

1.3.4.2. Les Composants ... 34
1.3.4.3. L'Assemblage ... 34
1.4. Ingénierie dirigée par les modèles ... 35
1.4.1. Apport de l'ingénierie dirigée par les modèles ... 36
2.4.2. Modèle de composants UML 2.0 ... 36
2.4.2.1. Interface ... 37
2.4.2.2. Port ... 37
1.4.2.3. Composant ... 38
1.4.2.4. Connecteurs ... 38
1.4.2.5. Structure composite ... 38
2.4.3. Architectures logicielles et UML ... 40
1.4.3.1. Utilisation directe d'UML ... 40
1.4.3.2. Utilisation des mécanismes d'extensibilité d'UML ... 41
1.4.3.3. Augmentation du métamodèle UML ... 42
1.5. Conclusion ... 42

Chapitre 2 : Composants logiciels multi-vues ... 43

2.1. Introduction ... 43
2.2. Modélisation multi-vues ... 44
2.2.1. Projet VBOOM ... 44
2.2.2. Relation de visibilité : Définition et propriétés ... 45
2.2.2.1. Définition ... 45
2.2.2.2. Propriétés ... 46
2.3. Modèle de composants logiciels multi-vues ... 47
2.3.1. Relation de visibilité pour les composants logiciels ... 47
2.3.2. Présentation du modèle de composants logiciels multi-vues ... 48
2.3.3. Processus de conception de CLM ... 49
2.4. Processus de transformation et mise en œuvre des CLM ... 49
2.4.1. Présentation du processus de transformation ... 49
2.4.2. Mise en œuvre des CLM en CCM ... 50
2.4.2.1 Processus de conception de CLM en CCM ... 50
2.4.2.1.1. Description des CLM : le langage VIEW-IDL3 ... 51
2.4.2.1.2. Projection de la description d'un CLM de VIEW-IDL3 vers IDL3 ... 53
2.4.2.1.3. L'interconnexion des composants ... 56

2.5. Conclusion 56

Chapitre 3 : Une approche à base de patron pour l'assemblage de Composants logiciels 57

3.1. Introduction 57
2.2. Patrons d'ingénierie 58
2.2.1. Historique et Définition 58
2.2.2. Patrons de différents niveaux d'ingénierie 59
2.2.3. Patron pour la génération de code multi-cibles 62
3.3. Relation de Composition comme support à la composition logicielle 64
3.3.1. Composition horizontale 65
3.3.2. Composition verticale 65
3.4. Patron d'assemblage pour les composants logiciels 66
3.4.1. Motivation 66
3.4.2. Adaptation des relations de liaisons 66
3.4.3. Représentation du patron pour les composants logiciels multi-vues 67
3.4.3.1. Canevas de présentation d'un patron 67
3.4.3.2. Patron pour les composants logiciels multi-vues 68
3.5. Vers une représentation complète du patron CLV 70
3.5.1. Représentation complète du patron CLV 70
3.5.2. Mise en œuvre du patron CLV amélioré 71
3.6. Profile UML pour les composants logiciels multi-vues 72
3.6.1. Traduction du diagramme de composants multi-vues 73
3.6.2. Profile UML 75
3.7. Conclusion 77

Chapitre 4 : Vers une démarche à base de composants logiciels 78

4.1. Introduction 78
4.2. Ingénierie logicielle basée composant (CBSE) 79
4.2.1. Catalysis 79
2.4.2. UML Component 80
4.2.3. Component Unified Process : CUP 81
4.3. Démarche à base de composants multi-vues 83
4.3.1. Modèles de la démarche MCM 83

4.3.2. Description générale de la démarche MCM 84
4.3.2. Phase 1: Analyse des besoins 86
4.3.3. Phase 2 : Elaboration des sous-modèles 90
4.3.3.1. Elaboration de diagramme de collaboration et description des composants 93
4.3.3.2. Définition de diagramme partiel de composants 96
4.3.3.3. Elaboration de diagramme de composants du sous-modèle 96
4.3.4. Phase 3 : Conception du modèle global 97
4.3.4.1. Fusion des composants partiels 97
4.3.4.2. Etablissement du diagramme de composants du modèle général 98
4.4. MCM une démarche dirigée par les modèles et outil support 98
4.4.1. MCM sous MDA 98
4.4.2. Outil support à MCM 100
4.5. Conclusion 102

Conclusion générale et Perspectives 104

Bibliographie 107

Figures et Tableaux

Figure 1.1. Présentation des éléments principaux d'un composant logiciel....................................... 19
Figure 1.2. Création d'une entité bean.. 26
Figure 1.3. Architecture de l'environnement .Net 31
Figure 1.4. Représentation d'un composant Fractal 33
Figure 1.5. Un exemple de définition d'une ADL à l'aide de Fractal ADL.. 34
Figure 1.6. Assemblage Fractal correspondant à l'ADL de la figure 1.5 34
Figure 1.7. Port Sans Interfaces.. 37
Figure 1.8. Port Avec Interfaces 37
Figure 1.9. Représentation des composants en UML 2.0 38
Figure 1.10. Vue externe d'un composant 39
Figure 1.11. Vue interne d'un composite.. 39
Figure 2.1. Graphe de visibilité... 46
Figure 2.2. Modèle de composant du CLM 48
Figure 2.3. Processus de transformation... 50
Figure 2.4. Processus de projection des fichiers VIEW-IDL3 vers IDL3 53
Figure 3.1. : Patron d'implémentation pour la génération de code multi-cibles.................................. 63
Figure 3.2. : Graphe d'implémentation multi-cibles 64
Figure 3.5. Connecteur d'assemblage entre deux ports 65
Figure 3.6. Connecteur d'assemblage entre deux interfaces... 65
Figure 3.7 : Connecteur de délégation entre un port externe et un port interne 66
Figure 3.8. Représentation Complète du patron CLV 70
Figure 3.9. Diagramme d'activité d'instanciation du patron amélioré CLV 72
Figure 3.10. Extrait du Diagramme à base de composant multi-vues Formation................................. 74
Figure 3.11. Diagramme traditionnel du diagramme à base de composant multi-vues Formation..... 75
Figure 3.12. Fragment du métamodèle associé au profil par un développement à base de CMV....... 76
Figure 4.1. Représentation du modèle de développement 84
Figure 4.2. Représentation du modèle de développement de la démarche MCM 85
Figure 4.3. Diagramme de cas d'utilisation du système conférence.. 87
Figure 4.4. Diagramme de composant multi-vues 90
Figure 4.5. Développement en parallèle des modèles de cas d'utilisation.. 91
Figure 4.6. Développement incrémental des modèles de cas d'utilisation 92
Figure 4.7. Description textuelle du scénario Définir le programme de la conférence 94
Figure 4.8. Diagramme de séquence du scénario : Définir le programme de la conférence............... 94

Figure 4.9. Diagramme de collaboration entre composants du scénario : Définir le programme de la conférence 95
Figure 4.10. Composants partiels de Session et Conference_Scientifique associés sous-modèle PROGRAMMATION 95
Figure 4.11. Diagramme partiel de composants du modèle de cas d'utilisation
Définir le programme de la conférence 96
Figure 4.12. Diagramme partiel de composants du sous-modèle PROGRAMMATION 97
Figure 4.13. Diagramme de composants de l'exemple conférence scientifique 99
Figure 4.14. Démarche MCM dirigée par les modèles MDA 100
Figure 4.15. Adaptation d'Objecteering/UML Modeler par des Profils UML réalisés
sous Objecteering/UML Profile Builder 101

Tableau 2.1. Le fichier de déclaration du CLM Médiathèque en VIEW-IDL3 52
Table 2.2. Ensemble des définitions introduits au langage IDL3 53
Tabeau 2.3. Un extrait de projection de la description de la Médiathèque en IDL3 56
Tableau 3.1. Exemple d'un patron d'analyse générale (patron Rôle) 60
Tableau 3.2. Exemple d'un patron de conception générale (patron Etat) 61
Tableau 3.3. Exemple d'un patron d'architecture (MVC 2) 62
Tableau 3.4. Tableau d'adaptation 67
Tableau 3.5. Canevas de présentation d'un patron 68
Tableau 3.6. Représentation générale du patron CLV 69
Tableau 3.7. Stéréotypes introduits dans le développement à base de composants multi-vues 76
Tableau 3.8. Eléments ajoutés dans le développement à base de composants multi-vues 76
Tableau 4.1. Décomposition des différents cas d'utilisation en cas d'utilisations élémentaires 89

INTRODUCTION GENERALE

Cadre

Le travail présenté dans ce rapport a été réalisé au sein du Laboratoire Modélisation Mathématiques et Calculs Scientifiques (LAMCS) de la Faculté des Sciences et Techniques de Béni-Mellal. Ce travail constitue une suite des travaux commencés depuis 2000 au laboratoire de Génie Logiciel (LGI) de l'ENSIAS de Rabat.

Contexte

Le monde a connu une véritable explosion en matière de demande de logiciels ces quinze dernières années. Cette explosion a engendré de sérieux impacts tactiques, stratégiques et organisationnels sur les organismes de développement de logiciels.

Le développement des applications utilisant les concepts traditionnels reste toujours une tâche coûteuse et complexe, malgré l'avantage d'être spécifiques et bien adaptées. La vision s'est alors radicalement changée en passant de la perspective de développement d'un système **monobloc** vers un système constitué en assemblant des pièces provenant de l'organisme lui-même ou d'un constructeur différent. Cette nouvelle vision a été basée principalement sur l'acquisition de ce qui est disponible pour développer ce qui est spécifique pour réduire la complexité de développement des logiciels. Un tel logiciel peut être ainsi construit par assemblage de briques logiciels réutilisables appelées **composants logiciels**.

Il existe actuellement en ingénierie logicielle un intérêt grandissant pour les techniques et les outils permettant de développer des applications par assemblage de composants logiciels. Par conséquent, une sous-discipline s'est nouvellement reconnue : Génie Logiciel Basé sur les Composants (GLBC). Cette sous-discipline consiste à proposer des techniques, des méthodologies et des procédés relatifs à ce nouveau genre de développement [Crnkovic et al. 2002]. Cet intérêt pour les composants résulte aussi bien de la volonté de réduire les coûts de développement en augmentant la réutilisation que de la nécessité d'inventer de nouvelles formes de développement. Aussi, pour prendre en compte de la complexité structurelle croissante des applications liée à de nouveaux besoins comme la répartition, la fiabilité ou l'évolution.

Cette nouvelle ère de l'orienté **composants** commence à peine à se développer alors que l'idée fut proposée pour la première fois en 1968 par McIlroy [McIlroy 1968]. Originellement, cette approche se fonde sur une analogie entre un électronicien qui fabrique un circuit par assemblage de composants électroniques via des connexions supportant le passage des électrons et un informaticien qui pourrait construire une application informatique par assemblage de composants logiciels réutilisables via des «connexions» supportant les communications et les échanges de données entre les composants. Actuellement, cette approche nécessite encore d'être précisée, outillée et expérimentée afin de réellement prendre son essor.

Le développement basé sur les composants (Component-Based Software Engineering : CBSE) a été récemment et largement adopté grâce à l'avenue d'une pléthore d'environnements RAD (Rapid Application Development) et de technologies de support [Collet et al. 2005]. Ce genre de développement a aussi favorisé la conception de gros systèmes d'entreprise basés sur des composants distribués, interopérables et encapsulant des opérations réutilisables.

Motivations et objectifs de la thèse

Actuellement, de nombreuses propositions se réclament du mode de développement par assemblage de composants logiciels. Cette abondance provient d'une part des industriels qui proposent de plus en plus de technologies comme le J2EE ou le .NET s'inscrivant dans l'approche composants et d'autre part des recherches académiques qui s'investissent à proposer des méthodes d'analyse et de conception à base de composants. Ces propositions sont souvent disparates et diversifiées de par leurs origines, leurs objectifs, leurs concepts, leurs mécanismes ou encore leurs utilisations.

De plus, dans la société de l'information qui se développe à grande vitesse notamment à travers l'Internet, l'accès aux informations devra être ouvert au plus grand nombre de citoyens. Pour cela, cet accès devra être différencié de façon à respecter les cultures, les niveaux de sensibilisation, les différences de vitesse d'apprentissage, les droits du citoyen et la protection des informations personnelles. Dans cette perspective, nous pensons que le développement de systèmes complexes "centrés sur les points de vue des acteurs" jouera un rôle stratégique dans le futur. A l'instar des chercheurs travaillant dans l'ingénierie des exigences, nous préconisons de prendre en compte les besoins des acteurs d'un système d'information (développeurs, utilisateurs finals, exploitants...) le plus tôt possible dans le processus de développement.

Le concept de vue/point de vue - que l'on retrouve également sous les termes voisins de rôle, sujet, perspective, aspect - a été étudié dans plusieurs domaines de l'informatique : bases de données, représentation des connaissances, analyse et conception, langages de programmation, outils de Génie logiciel, etc. [Krumeich et al. 2014]. Parmi les premiers travaux qui ont été menés pour intégrer ce concept dans la modélisation des systèmes complexes, on peut citer : la définition du langage VBOOL (View Based Oriented Object Language) [Marcaillou et al. 1994], la méthode associée VBOOM (View Based Oriented object Method) [Kriouile 1995] [Coulette et al. 1995] et son outil support VB_TOOL [Hair 1997] [Marzak 1997]. Encore, on trouve le travail le plus abouti est la standardisation de VBOOM sous le standard UML. En d'autre terme, il s'agit de l'intégration du concept de vue/point de vue dans le langage de modélisation unifié UML [Hair 2004b] [Hair 2005]. La génération de code multi-cibles des langages objets du marché comme C++, Java, etc. et non uniquement le langage VBOOL de la méthode standardisée U_VBOOM (Unified_ View Based Oriented object Method) représente un travail très intéressant dans le domaine et il a bien favorisé l'utilisation de la méthode U_VBOOM [Hair 2004c]. Cette génération est basée sur une nouvelle implantation de la relation de visibilité et les concepts liés en s'appuyant sur le patron d'analyse Rôle.

Nous nous sommes intéressés, dans cette thèse à intégrer le concept vue/point de vue dans le développement du logiciel basé sur les composants en général. Pour mener ce travail, nous avons fixé les deux objectifs suivants :

- proposer une (ou plusieurs) démarche(s) d'assemblage de composants logiciels centrée utilisateur (point de vue). Du fait que les composants disponibles ne sont pas toujours compatibles, alors la technologie des connecteurs logiciels est proposée pour adapter les composants hétérogènes afin de faciliter leur assemblage [Nikunj et al. 2000]. Certains types d'assemblage n'ont pas encore été étudiés. Parmi ces types, l'assemblage des composants via la ***relation de visibilité*** pour pouvoir concevoir des composants dits ***composants multi-vues***. Pour ce faire, nous avons développé deux approches d'assemblage. La première consiste à proposer un nouveau type de connecteur entre les composants logiciels appelé ***connecteur de visibilité*** et la deuxième propose une nouvelle implémentation de la relation via un nouveau patron nommé ***Patron Composant Logiciel multi-Vues (CLV)***.

- proposer une méthodologie d'analyse/conception basée composants et centrée sur les points de vue des acteurs interagissant avec le système. L'effort des travaux de recherche s'est d'abord porté sur les plates-formes technologiques, afin de répondre à des besoins d'utilisateurs. Dans la réalité, on se retrouve avec plusieurs composants logiciels développés séparément et coexistent avec les risques d'incohérence

associés, et avec un modèle global, représentant le système tout entier, constitué par plusieurs composants fréquemment remis en cause, et parfois profondément, quand les besoins des utilisateurs évoluent. Pour remédier à ces problèmes, nous avons reconsidérer les grandes lignes de l'approche développée dans U_VBOOM en conservant son objective d'une modélisation multi-vues et s'inspirer des grandes lignes de la méthode à base de composants CUP (Component Unified Process) [D'Souza D. et al 1998] pour proposer une nouvelle méthode de développement à base de composants logiciels. Cette méthode permet la construction d'un modèle global représentant tout le système à développer et qui prend en compte simultanément tous les besoins des acteurs et qui permet une évolution facile de de ce dernier quand les besoins des utilisateurs évoluent. Tenir compte de l'effet UML et sa généralisation comme standard dans la modélisation de systèmes logiciels, la nouvelle méthode d'analyse/conception basée composants logiciels présente un succès dans le CBSE.

L'élaboration des profils UML pour réaliser un assemblage orienté point de vue (utilisateur) des composants logiciels ou faire une analyse/conception d'un tel système à base de composants logiciels s'appuie sur les mécanismes d'extension du formalisme UML, notamment les stéréotypes, sans remettre en cause le méta-modèle UML existant, ce qui ouvre la voie au support des démarches proposées par les principaux outils UML du marché et donc d'être utilisable par tous les concepteurs.

Organisation de ce mémoire

Outre ce chapitre d'introduction, qui a défini le cadre dans lequel se situe ce travail et a présenté la problématique abordée dans notre travail, ce mémoire de thèse est organisé en quatre autres chapitres.

Le chapitre 1 dresse en premier lieu les notions de base liées aux composants logiciels suivies par un état de l'art de quelques modèles de composants les plus utilisés, à savoir EJB [Edwards et al. 2002], CCM [OMG 2002a], .NET et Fractal en abordant certains points essentiels de ces modèles à savoir la spécification, les composants et l'assemblage. Nous terminons ce chapitre par l'ingénierie dirigée par les modèles en présentant le modèle de composants UML 2.0.

Le chapitre 2 présente notre première proposition d'assemblage de composants logiciels. Après avoir présenté une brève description de la modélisation multi-vues, en insistant sur la définition de la relation de visibilité et ses principales propriétés, le modèle d'assemblage associé à cette relation pour les composants logiciels multi-

vues (CLM) est alors proposé. La transformation du modèle et sa mise en œuvre pour le modèle CCM concluent ce chapitre.

Le chapitre 3 propose notre deuxième approche d'assemblage de composants logiciels. Il rappelle la notion de patron et présente quelques patrons et leurs réutilisations. Ensuite, la nouvelle implémentation de la relation de visibilité à l'aide de patron génération de code multi-cibles est présentée. Puis, il présente le patron CLV que nous avons développé pour réaliser l'assemblage des composants logiciels multi-vues. Enfin, il se conclu par la présentation du profile UML associé au développement à base de composants multi-vues pour valider l'approche proposée.

Le chapitre 4 présente une nouvelle méthode de développement à base de composants logiciels. Ce chapitre aborde tout d'abord l'ingénierie logicielle basée composants (CBSE) par une présentation de quelques méthodes : Catalysis, UML Component et CUP. Ensuite, il présente la démarche sous les différents modèles (cas d'utilisation, interaction, conception et fusion). Puis, les différentes phases de développement de la démarche sont alors proposées. Enfin, le chapitre situe la démarche dans le contexte MDA et conclu par le prototype support.

Enfin, la conclusion de ce mémoire reprend les contributions de cette thèse. Elle dresse également les principales perspectives de ce travail que nous envisageons comme suite à ce sujet de thèse.

Chapitre I

Composants logiciels

1.1. Introduction

Depuis des années le monde a connu une énorme utilisation des logiciels dans tous les domaines tel que l'industrie, l'enseignement, l'administration et même dans plusieurs aspects de la vie quotidienne. Cela se traduit en une augmentation des exigences sur la production du logiciel. On demande de plus en plus des logiciels fiables, flexibles, robustes, adaptables, mieux exploitables et que l'on peut facilement installer et déployer. Cette demande grandissante du logiciel ainsi que l'augmentation sur le plan des exigences de qualités ont provoqué une grande complexité à la fois au niveau des logiciels mêmes qu'aux niveaux des processus de développement de ces derniers. Comment vaincre cette complexité ou au minimum la réduire ? Comment adapter rapidement des logiciels face aux changements ? Et finalement, comment prendre en compte l'évolution du logiciel dès son développement ? Ces points présentent des défis très importants pour les développeurs de logiciels et des systèmes informatiques de manière générale.

La réutilisation présente une clé de solutions pour ces problèmes. L'idée de réutiliser des logiciels en tout ou en partie est très ancienne. L'approche de développement par composants rétablie l'idée de réutilisation en introduisant de nouveaux éléments. Dans cette approche, le logiciel est construit par assemblage de composants développés auparavant et préparés pour être intégrés.

Les expériences ont montré que le développement à base de composants exige une approche systématique pour se concentrer sur les aspects du composant pendant le développement de logiciels [Crnkovic et al. 2002]. Les techniques du génie logiciel traditionnel doivent être ajustées à la nouvelle approche et de nouvelles procédures doivent être développées. Le génie logiciel à base de composants (*Component-based software engineering* (CBSE)) est reconnu comme nouvelle sous-discipline du génie logiciel où principalement on s'intéresse à fournir des supports pour le développement de composants autant qu'entités réutilisables, des supports pour

l'assemblage de composants ainsi que des supports pour la maintenance et la mise à niveau des applications à base de composants par remplacement ou personnalisation de composants logiciels [Crnkovic et al. 2002].

Dans ce premier chapitre nous présentons, dans la section 1.2, les concepts principaux des composants logiciels en donnant les différentes définitions proposées d'un composant logiciel et les éléments de base pour présenter un composant logiciel. La section 1.3 expose quelques modèles technologiques, à base de composants, les plus utilisés. Puis, la section 1.4 présente l'initiative de l'architecture dirigée par les modèles MDA (Model Driven Architecture) de l'OMG [Schmidt, 2006] [Oussalah 2005] qui permet de qualifier une démarche en référence au Processus Unifié avec une présentation du modèle à base de composants UML 2.0 [Booch et al. 2005] [Miles et al. 2013].

1.2. Les concepts principaux des composants logiciels

1.2.1. Différentes définitions du composant

La notion de composant vise un développement modulaire et une maintenance plus aisée des applications. Les composants (comme les objets) favorisent la réutilisation et la programmation à grande échelle. Ils cherchent à maîtriser le couplage entre les entités et à mettre en avant la notion d'architecture. Pour cela, ils s'appuient sur les notions de composition et d'assemblage.

Il existe de nombreuses définitions de composant dans la littérature. Nous présentons quelques définitions qui nous semblent pouvoir couvrir toutes les manières selon lesquelles ce concept peut être aperçu.

Szyperski dans [Szyperski 1998] définit un composant comme «*Une unité de composition avec des interfaces spécifiées contractuellement et seulement des dépendances explicites vis à vis de son contexte. Un composant logiciel doit pouvoir être déployé indépendamment et fait l'objet de composition par des tiers. »*

En analysant cette définition, on constate que : le composant communique avec son environnement à travers des interfaces. Par conséquent, celles-ci doivent être spécifiées clairement tout en encapsulant l'implémentation à l'intérieur du composant.

Selon D'Souza et Wills [D'Souza D. et al. 1998] *«un composant est une partie réutilisable d'une application. Il est développé indépendamment et peut être combiné avec d'autres composants pour construire des unités plus grandes. Un composant peut être adapté mais il ne peut pas subir des modifications».* La spécificité de cette

définition réside dans le fait qu'il est noté en clair qu'un composant ne peut pas être modifié.

Brown [Brown 2000] propose la définition suivante : *«Un composant logiciel est un élément logiciel conforme à un modèle de composants et peut être indépendamment déployé et composé sans modification selon un standard de composition.»* En plus des aspects qui apparaissent des autres définitions, cette définition mentionne qu'un composant doit être conforme à un modèle de composants. Cette définition est très proche de la vision de l'industrie qui est différente de la manière avec laquelle les composants sont vus dans les milieux académiques.

Pour conclure, nous revenons au point de départ. Il existe plusieurs définitions pour le concept composant. Dans chacune, l'accent est mis sur un ou plusieurs aspects. Tout le monde est d'accord sur quelques points, le composant est une unité de composition. Il doit être spécifié pour être composé avec d'autres composants ou intégré dans des applications ; les composants sont réutilisables, la réutilisation dans le génie logiciel à base de composant est différente de celle que nous pouvons trouver dans la technologie orientée objet ou d'autre technologies liées au génie logiciel traditionnel [Gröne et al. 2005]. Cette différence réside dans le fait qu'un composant peut être utilisé au moment de l'exécution «*run time*» sans le besoin de recompilation. Le deuxième argument de cette différence apparait parce que le composant détache ses interfaces de son implémentation ce qui permet la composition sans être obligé de savoir aucun détail sur l'implémentation du composant.

1.2.1. Représentation d'un composant

Parmi les nombreux modèles de composant existants, nous avons choisi de présenter celui décrit dans [Oussalah 2005] pour illustrer juste les principaux concepts qui nous intéressent dans les composants. La figure 1.1 représente un méta-modèle décrivant la notion de composant et de ses principaux éléments. Un composant est généralement représenté par les éléments suivants [Oussalah 2005] :

- son type qui représente la définition abstraite du composant caractérisé par ses interfaces avec leurs modes de connexion et ses propriétés configurables ou non.
- son implantation qui contient la mise en œuvre des aspects fonctionnels (code métier) et non fonctionnels (gestion des connexions, persistance, transactions, etc.) de son type.

- ses instances qui s'exécutent dans le système. Chaque instance est définie par une référence unique, à savoir un type de composant et une implantation particulière de ce type.

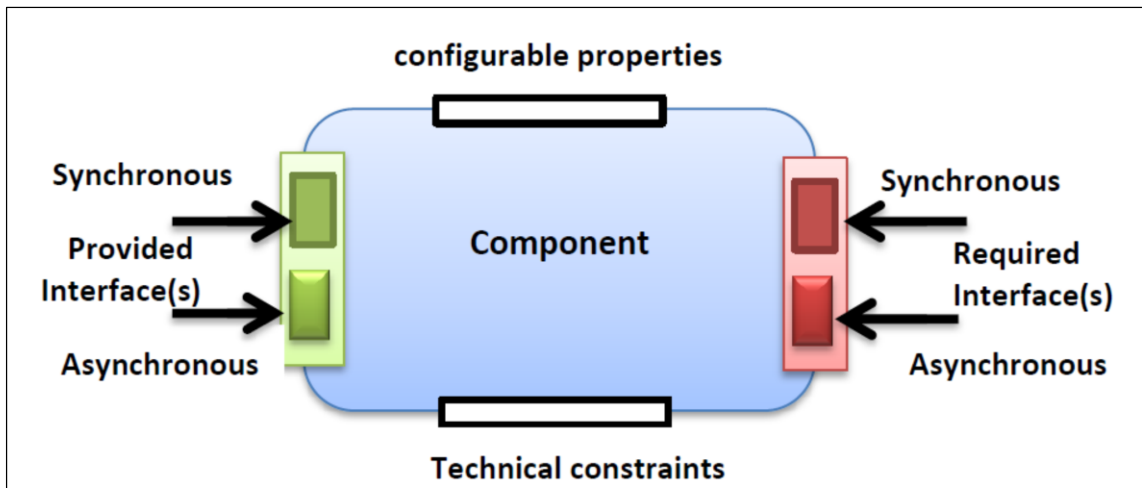


Figure 1.1. *Présentation des éléments principaux d'un composant logiciel*

1.2.1.1. Interface

Les interfaces de composants définissent les services requis et fournis par un type de composant. L'interface décrit entièrement ce que fait le composant et ce qu'il a besoin de connaître. De ce fait une interface définit des contrats qui doivent être respectés pour la réalisation d'un composant. Pour un objet, l'interface correspond à la liste des signatures de ses méthodes. Cette notion de contrat est assez pauvre puisqu'il est principalement syntaxique et ne concerne que ce que fournit l'objet. Dans les modèles de composant, les contrats peuvent prendre différentes formes souvent beaucoup plus riches [Beugnard et al. 99]. La forme la plus commune de contrats est la spécification des services fournis et requis par un composant. Un autre type de contrat consiste en la définition de *pré* et *post-conditions* sur l'exécution des méthodes. D'autres contrats plus complexes permettent la spécification de protocoles d'interactions spécifiant les séquences de services qu'un composant accepte. Une interface est composée de deux éléments (cf. figure 1.1) :

Les points de connexion sont les points d'interaction (des besoins ou des services) entre le composant et son environnement. Ils sont appelés aussi port. Le port qui exprime le besoin du composant en termes de service est un *port requis*. Le port qui fournit un service aux autres composants est un *port fourni*. Un port qui fournit un service et qui en requiert en même temps, est un *port mixte*.

L***es services*** décrivent le comportement fonctionnel du composant en exprimant la sémantique des fonctionnalités fournies (ce que fait le composant) et requises par le composant (ses besoins pour fonctionner).

D'après [Oussalah 2005], un port peut avoir plusieurs modes de connexions possibles: synchrone, asynchrone et continu.

1.2.1.2. Propriétés

Les propriétés d'un composant peuvent concerner aussi bien sa structure, son comportement ou ses fonctionnalités. Elles peuvent être paramétrées et configurées selon le contexte d'exécution. Il existe également des propriétés non fonctionnelles qui ne font pas partie des services applicatifs et sont plutôt fournies par l'architecture à travers le serveur et les conteneurs (sécurité, réseau, traçage, etc.).

1.2.1.3. Contraintes

Les contraintes représentées dans le méta-modèle de la figure 1.1 représentent les besoins des composants et les conditions de leur bon fonctionnement. Ces exigences portent sur l'environnement de déploiement et sur l'utilisateur. Elles peuvent également être un type spécial de propriétés mais elles ont une syntaxe spéciale vu leur rôle dans la conception d'architecture à base de composant.

1.2.2. Connecteurs

En se basant sur les définitions proposées dans la littérature pour les connecteurs, nous proposons la définition ci-après qui combine les points essentiels de celles-ci. Les connecteurs sont des entités architecturales de communication qui modélisent de manière explicite les interactions entre les composants [Traverson et al. 2002]. Ils contiennent des informations concernant les règles d'interaction entre les composants. Ainsi, l'objectif des connecteurs est d'atteindre une meilleure réutilisabilité lors de l'assemblage des composants. En effet, la raison de l'existence des connecteurs est de faciliter le développement d'applications à base de composants logiciels. Les composants s'occupent du calcul et du stockage tandis que les connecteurs s'occupent de gérer les interactions (communication/coordination) entre les composants. 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. Medvidovic [Medvidovic et al. 2000] a classé les services d'interaction offerts par les connecteurs

en quatre types. Chaque type de connecteur offre un ou plusieurs services d'interaction. Ces services sont les suivants :

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

• **Le service de coordination :** supporte le transfert de contrôle entre composants. Les appels de fonctions sont un exemple de cette catégorie de connecteurs.

• **Le service de conversion :** convertit les interactions inter-composant si nécessaire. Il permet aux composants hétérogènes d'interagir. L'inadéquation d'interaction est un obstacle majeur dans la composition des grands systèmes. Les services de conversion permettent aux composants qui n'ont pas été spécialement conçus pour fonctionner les uns avec les autres, d'établir et de mener des interactions.

• **Le service de facilitation :** négocie et améliore l'interaction entre composants.

Les composants et les connecteurs sont assemblés à partir de leurs interfaces (ports) pour former une configuration particulière. Une configuration est un agencement, une topologie. En d'autres termes, il s'agit d'un graphe de composants et de connecteurs qui décrit une structure architecturale permettant de déterminer si les composants et les connecteurs sont correctement composés.

1.2.3. Contrats

Les contrats [Meyer 1992] trouvent leurs raisons d'exister par le besoin de décrire le comportement global d'un composant et par la limitation de la majorité des techniques de descriptions d'interfaces comme les IDL [Gudgin 2001] qui ne traitent que les signatures selon une vision purement syntaxique. Les contrats [Meyer 1992] permettent d'avoir des spécifications plus précises pour les comportements des composants. Bertrand Meyer est parmi les premiers qui ont introduit les contrats dans le développement des systèmes. Il mentionne dans [Meyer 1992] qu'un contrat précise les contraintes globales (invariantes) qu'un composant doit maintenir.

Pour chaque opération le contrat précise les pré-conditions que les clients doivent satisfaire ainsi que les post-conditions promises en retour par le composant. L'ensemble des pré-conditions, post-conditions et les invariants à maintenir constituent la spécification du comportement d'un composant.

En plus de leurs utilisations pour spécifier les comportements des composants individuellement, les contrats sont utilisés aussi pour spécifier les interactions entre un ensemble de composants. Cette spécification concerne : l'ensemble de composants ; les rôles de chaque composant à travers un ensemble d'obligations

ainsi que leurs types ; et un ensemble d'invariants à maintenir par les composants. L'utilisation des contrats dans ce cas favorise la réalisation ainsi que le raffinement d'unités logicielles à base de composants de grande granularité. Cela est dû aux facteurs suivants :

- Les contrats permettent aux développeurs du logiciel d'isoler et de spécifier explicitement avec un niveau élevé d'abstraction les rôles de tous les composants dans des contextes particuliers.
- La présence de plusieurs contrats permet de modifier indépendamment les rôles de chaque composant, ou de faire des extensions des rôles.
- De nouveaux contrats peuvent être obtenus par l'association de différents participants avec différents rôles.

Puisqu'un composant peut participer dans plusieurs contrats, son comportement global peut être assez complexe. Plus encore, les contrats spécifient les conditions selon lesquelles les composants interagissent avec d'autres composants en termes de *pré* et *post* conditions.

1.2.4. Cycle de vie

Un modèle de composants regroupe un ensemble de conventions à respecter lors de la construction et l'utilisation des composants. Ces conventions permettent de définir et de gérer d'une manière uniforme les composants. Elles couvrent toutes les phases du cycle de vie d'un logiciel à base de composants [Farias 2003] [Lau et al. 05]. Les phases du cycle de vie le plus usuel d'un composant sont les suivantes :

1. Création : durant cette phase, les services offerts et requis, les pré et post-conditions, etc. sont définis en respectant les contraintes spécifiées dans les interfaces. L'emballage du composant (s'il contient d'autres composants par exemple) a lieu également pendant cette phase.

2. Assemblage : cette opération consiste à assembler tous les composants d'une application. Il reflète l'architecture d'une application. Les composants de l'application sont reliés les uns aux autres suivant l'architecture d'assemblage. Un assemblage doit être empaqueté pour qu'il puisse être diffusable et utilisable par des tiers. L'assemblage peut être réalisé pendant l'exécution comme dans le modèle de composant Fractal [Bruneton et al. 2003][Bruneton 2004].

3. Déploiement : le déploiement d'un composant consiste à extraire le composant de son paquet, à l'installer dans son site d'utilisation et à valider les contraintes qui existent sur son déploiement [Bálek et al. 2001]. Le composant est défini comme une

entité logicielle conçue et construite indépendamment d'une application spécifique. Elle est intégrée plus tard dans une ou plusieurs applications. Cette entité se présente souvent sous la forme d'un fichier archive, contenant principalement le code du composant, et un descripteur de déploiement indiquant comment effectuer ce déploiement [Carrez 2003]. Ce descripteur du composant contenu dans le paquet est utilisé pour instancier le composant. Souvent, ces descripteurs de déploiement sont écrits en XML eXtended Markup Language [Bray 2004]. Ils contiennent toutes les caractéristiques nécessaires au déploiement du composant (catégorie de composant, interfaces, services requis, etc.).

4. Exécution : elle représente l'étape d'utilisation du composant par le client, après son installation.

1.3. Quelques modèles de composants

Le concept du modèle de composants présente un appui essentiel pour le développement, la composition, la communication, le déploiement et l'évolution des composants. Le modèle de composants est un facteur important pour traiter les aspects de l'interopérabilité, l'évolutivité, la maintenabilité, ainsi que de nombreux autres attributs de qualité. La majorité des modèles de composants existants offrent des possibilités d'ajouter de nouveaux services ou de nouvelles fonctionnalités aux systèmes logiciels d'une manière transparente. Cela est dû aux possibilités offertes par la majorité des modèles de composants entre autres le pouvoir de définir explicitement les composants et les connexions entre ces derniers ; le pouvoir de définir explicitement les implémentations de composants à partir de codes natifs; et finalement définir explicitement les propriétés extra fonctionnelles des composants. La majorité des modèles de composants industriels visent certains objectifs communs comme l'augmentation de l'indépendance des codes ; augmenter la transparence des services ; la mise en œuvre de la distribution et la généralisation de service.

Il existe plusieurs modèles et de technologies de composants logiciels comme COM [Microsoft 2005], CCM [OMG 2002a][Marvie et al. 2001], JAVA BEANS [SUN 2010], .NET [Microsoft 2010], FRACTAL [Bruneton et al. 2004] et OSGI [OSGI 2010]. Ces modèles de composants se diffèrent les uns des autres principalement selon les points suivants : les concepts clés de chaque modèle ; la manière d'implémenter les composants ; la manière dont sont réalisés les assemblages et finalement le cycle de vie du composant ainsi que celui de l'application à base de composants.

Dans cette section, nous présentons brièvement quelques modèles de composants les plus connus : JavaBeans EJB**,** CCM, .NET et Fractal. Vu l'objectif de cette thèse,

nous nous concentrons sur leurs spécifications, la description de leurs architectures ainsi le principe de leur déploiement.

1.3.1. JavaBeans EJB

Le modèle Java Beans [SUN 2010] [Wetherbeeet al. 2015] présente une première tentative d'intégrer la notion de composant dans le langage Java. Le modèle met l'accent sur la réutilisation et l'augmentation des capacités de composition statique et dynamique. L'objectif principal de Java Beans est de définir un modèle de composants pour java. Ce modèle est utilisé pour créer des composants que l'on peut déplacer et composer ensemble dans une application.

1.3.1.1. Les Spécifications

Les composants EJB héritent des propriétés du JavaBeans. L'architecture d'EJB est une extension du modèle de composants JavaBeans au schéma client-serveur [Wetherbee et al. 2015]. L'architecture EJB a pour but de :

- Fournir une architecture standard de composants pour construire les applications réparties orientées objets en langage Java en combinant des composants développés avec des outils de différents vendeurs.

- Développer les composants EJB une seule fois et les déployer dans plusieurs plate-formes sans recompilation ou modification de code (Write Once, Run Anywhere).

- Se concentrer sur les aspects de développement, de déploiement, et d'exécution du cycle de vie d'une application métier.

- Être compatible avec les plateformes existantes, ayant une API Java.

- Fournir l'interopérabilité entre les composants EJB et les applications dans un langage différent de Java.

- etc.

1.3.1.2. Les composants

La technologie EJB supporte 3 types de composants : les composants temporaires appelés « session bean», les composants persistants appelés « entity bean » et les composants réactifs activés par message <<message-driven beans>> [Rubinger et al. 2010].

Les composants sessions : les sessions modélisent les processus métier. Une session est créée par le client et existe seulement pendant une seule interaction client/serveur. Au nom d'un client, elle réalise les accès aux bases de données, ou les calculs, ou même les mises à jour dans une base de données partagée. Elle n'est pas partagée entre plusieurs utilisateurs. Elle peut être transactionnelle, mais elle disparaît après une panne du serveur EJB ou une erreur système. Le conteneur peut aussi la supprimer. Le conteneur gère l'état d'une session bean. La session elle-même doit gérer ses propres données.

Une session bean peut être sans état *(stateless) ;* il ne garde pas de trace des informations échangées d'un appel de méthode à une autre. C'est un bean sans variable d'instance et deux de ses instances sont équivalentes. En effet, il ne possède pas de variable d'instance et n'est créé que pour une requête d'un client. Un conteneur peut facilement gérer ce type de bean car il peut être créé ou détruit à tout moment sans se soucier de son état.

Une session bean peut être avec état (*stateful*) ; il possède une variable d'instance et donc change d'état au cours d'une transaction. L'exemple classique est le caddy dans un site de vente sur Internet où le client enregistre ses achats et à tout moment peut décider d'en payer le contenu. Le cycle de vie de ce type de session bean est plus compliqué à gérer pour le conteneur, car s'il veut le suspendre temporairement, il doit stocker son état afin de le restaurer.

Les composants entités (Entity beans) : Les beans entités ont un cycle de vie long. Ils existent au travers de plusieurs sessions clientes et sont partagés par plusieurs clients. Ils survivent aux pannes diverses du serveur. Ils représentent aussi un ensemble de données dans un système de stockage persistant comme une base de données. Les informations stockées dans les EJB entités sont liées aux informations de la base de données sous- jacente. Hormis le fait que les données sont conservées entre plusieurs sessions, les spécifications EJB prévoient deux types de synchronisation entre l'EJB et la base de données qu'il représente.

La persistance gérée par bean BMP ((Bean-Managed Persistence) spécifie que le bean doit lui-même contenir les primitives de sauvegarde et de restauration vers une base de données persistante. Les méthodes ejbLoad() et ejbStoreO sont implémentées dans le bean. Le conteneur appelle l'une pour restaurer l'état de l'EJB depuis une base de données, et l'autre pour le sauvegarder. Ainsi, le programmeur possède la main sur la gestion de la persistance, mais l'utilisation de la persistance gérée par bean ne permet pas un portage sur d'autres environnements que celui pour lequel il a été créé. La persistance gérée par un conteneur évite d'avoir à implémenter les méthodes ejbLoad() et ejbStore().

Tandis que pour le CMP (Container-Managed Persistence), le conteneur réalise automatiquement toutes les opérations de sauvegarde, de chargement et de recherche. La tâche du développeur est donc allégée.

Le cycle de vie des composants entités est similaire à celui d'une session avec état, sauf qu'il y a des méthodes de sauvegarde/restauration d'état ou de recherche par clé (cf. figure 1.2).

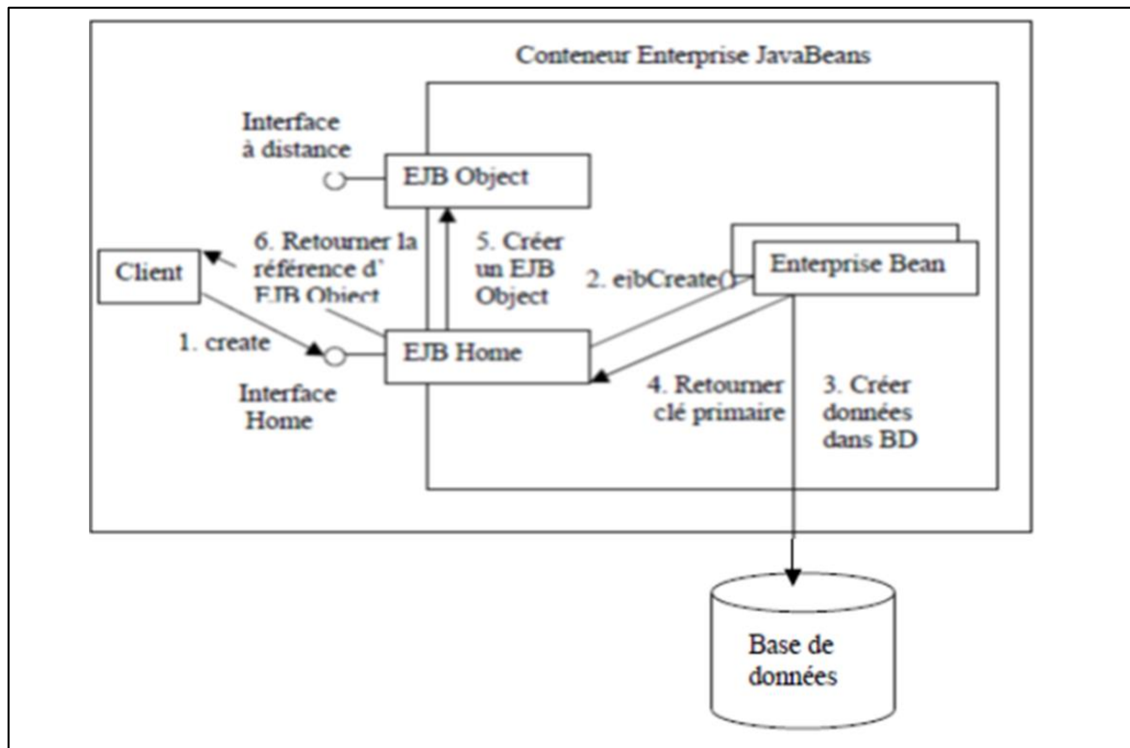


Figure .1.2. *Création d'une entité bean*

1.3.1.3. Déploiement et assemblage

Dans cette partie, nous allons décrire les étapes de développement d'une application à base de composants EJB exécutée sur la plate-forme JonAS (une implémentation de J2EE). Nous commençons par l'écriture du composant EJB et ses descripteurs de déploiement et d'empaquetage, puis l'écriture du composant Client et enfin sa mise en œuvre.

Etape 1 : Développement du composant EJB

Cette étape définit quatre phases à suivre pour réaliser un EJB.

1- ***L'écriture du composant EJB :*** Un composant EJB se compose de trois entités (trois classes du point de vue programmeur) :

- **L'interface distante** : cette interface définit les méthodes métier d'un EJB ;

- **L'interface maison** : elle définit les méthodes qu'un client peut invoquer pour créer, trouver ou supprimer un EJB.

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

- **L'Enterprise Java Bean** : qui est l'implémentation de notre composant.

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

- **Descripteur de déploiement standard :** le programmeur de l'EJB doit fournir avec son composant un descripteur de déploiement. Ce fichier décrit, par exemple, quelle classe est l'implémentation, l'interface locale et l'interface distante de l'EJB.
- **Descripteur de déploiement spécifique à JOnAS :** certaines informations nécessaires au déploiement de l'EJB dans JOnAS ne sont pas dans le descripteur précédent. Pour résoudre ce problème, il faut créer un autre descripteur de déploiement spécifique à JOnAS. Ce document devra s'appeler jonasejb-jar.xml.

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

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

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

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

Etape 2 : l'implémentation du Client de l'EJB

Dans cette phase, nous allons implémenter le code du client de notre EJB. Il suffit d'écrire les lignes de code qui permettent de trouver les interfaces locales, invoquer des méthodes, etc.

Etape 3 : La mise en œuvre

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

1.3.2. CCM

Le modèle CCM (Corba Component Model) de l'OMG (Object Management Group) [OMG 2002a][OMG 2002b][OMG 2002c] vise à améliorer la réutilisation des composants logiciels. Il favorise la description plutôt que la programmation (seuls les aspects fonctionnels doivent être programmés, les autres aspects doivent être décrits et/ou générés). La description des spécifications, des composants et du déploiement est présentée ci-dessous.

1.3.2.1 Les spécifications

Dans la spécification CCM, il existe quatre modèles de spécification ainsi qu'un méta-modèle pour les composants CCM. Ils sont définis comme suit :

- *le modèle abstrait* permet de décrire la spécification du composant en IDL (Interface Definition Language) [OMG 2002b] ;

- *le modèle de programmation* permet de définir la coopération entre le composant et la plate-forme d'exécution ; il définit les relations entre l'implantation du composant et les spécifications du modèle abstrait.

- *le modèle de déploiement* définit un processus d'installation de composants sur différents sites ;

- le *modèle d'exécution* définit l'environnement d'exécution des composants (avec les services associés, comme la transaction ou la persistance).

1.3.2.2. Les composants

La norme CCM définit un modèle abstrait de composant qui impose une représentation précise des relations entre le composant et son environnement (notamment les services que fournit ou requiert le composant). Deux modes d'interaction entrante sont fournis :

- *Les facettes* sont des ports fournis à invocation synchrone, qui représentent les différentes interfaces fonctionnelles.

- *Les puits d'évènement* sont des ports fournis permettant les notifications asynchrones. Ce sont les points d'entrée dans lesquels des évènements peuvent être déposés.

De plus, un composant peut définir ses interfaces requises, qui définissent comment le composant utilise les services d'autres composants :

- *Les réceptacles* sont des ports requis à invocation synchrone. Ce sont les points d'entrée qui décrivent la capacité d'un composant à pouvoir exploiter des références d'objets fournies par une entité tierce (ils représentent les dépendances vers d'autres composants).

- *Les sources d'évènement* sont des ports requis pour les notifications asynchrones. Elles émettent des évènements vers plusieurs consommateurs ou vers un canal d'évènement.

L'interface est décrite en IDL et comporte plusieurs types de ports que l'on peut interconnecter entre eux. Chaque port fourni du composant correspond à l'implantation d'une interface et a sa propre référence (référence de facette/de puits). Trois interfaces (facettes) sont implantées par tous les composants, et permettent d'accéder aux différents ports proposés par le composant :

- Components::Navigation permet d'accéder aux facettes ;
- Components::Receptacles permet de (dé)connecter les composants pendant leur exécution ;
- Components::Events permet d'accéder aux sources et puits d'événements (avec notamment des opérations de (dés)abonnement à une source d'événements).

Quatre catégories d'instance de composants existent : *Service*, *Session*, *Processus* et *Entité* (selon que l'on considère le mode d'accès et l'existence d'une clé primaire liée à la notion de persistance[3]). Les deux premières catégories sont non persistantes ; donc elles n'ont pas de clés primaires. De plus, la première catégorie est avec état et la seconde est sans état. La persistance de la catégorie *processus* est transparente pour le client, ce qui n'est pas le cas avec la catégorie *entité*. Le type du composant est décrit dans le descripteur de déploiement correspondant qui va être abordé par la suite dans le déploiement.

1.3.2.3. Déploiement et assemblage

CCM spécifie explicitement un scénario de déploiement pour les composants. Ainsi, les composants sont empaquetés dans des paquets auto-descriptifs pouvant être assemblés entre eux. Ils forment ainsi un assemblage, qui correspond à l'unité de déploiement de CCM. Pour réaliser le déploiement d'applications, il est nécessaire de définir la liste des composants constituant l'application, leur localisation et leurs connexions. Pour chaque composant, il est nécessaire de spécifier ses éléments (interfaces, implantations), de décrire les *dépendances* qui représentent ses contraintes systèmes (OS, ORB (*Object Request Broker*) et sa configuration initiale. Toutes ces informations permettent d'installer et d'exécuter l'application. Pour décrire ces informations, CCM spécifie des descripteurs décrivant les paquets, les composants et les assemblages. Ces différents descripteurs sont basés sur un langage dérivé de XML qui s'appelle OSD (*Open Software Descriptor*) [Hoff et al. 1997].

Dans ce modèle, le déploiement est réalisé par un ORB qui représente le support d'exécution des applications, il fournit les différents services d'interaction. Il existe également des outils de déploiement qui utilisent les services de l'ORB pour réaliser le transfert, l'installation, l'assemblage, l'instanciation et la configuration des applications sur les machines des clients.

Les différents éléments utilisés pour le déploiement d'un composant CCM sont regroupés dans deux fichiers ZIP correspondant à deux paquets différents : *le paquet de composant* qui donne les informations associées au déploiement d'un composant et *le paquet d'assemblage* qui contient l'ensemble des paquets de composants, un descripteur de propriétés et un descripteur d'assemblage *CAD (Component Assembly Descriptor)* qui donne les caractéristiques de l'assemblage (fabriques, instances de composant et connections à créer). Il représente donc un descripteur d'architecture (ADL).

1.3.3. Microsoft .NET

L'architecture .NET proposée dans [Platt 2002] par Microsoft, représente un regroupement d'applications Microsoft qui vise les deux points suivants :

- une évolution du modèle de composant COM+;
- fournir une plate-forme de développement Windows, qui soit ouverte à tout type de langage.

1.3.3.1. Les spécifications

Deux spécifications de l'organisation de standardisation internationale *ECMA (European Computer Manufacturers Association)* existent pour l'architecture de Microsoft : le langage C# et l'infrastructure CLI (*Common Language Infrastructure*).

La figure 1.3 représente l'architecture générale de .NET. Il s'agit d'un ensemble de services communs, utilisables depuis plusieurs langages. Ces services s'exécutent sous la forme d'un code intermédiaire indépendant de l'architecture sous-jacente. Ces services s'exécutent dans un *CLR (Common Language Runtime)* assurant les fonctions de gestion des ressources et de surveillance du bon fonctionnement des applications. Enfin, pour garantir l'interopérabilité entre les langages, le langage de spécification *CLS (Common Language Specification)* est utilisé pour définir les propriétés qu'un langage doit vérifier pour interopérer avec la plate-forme .NET.

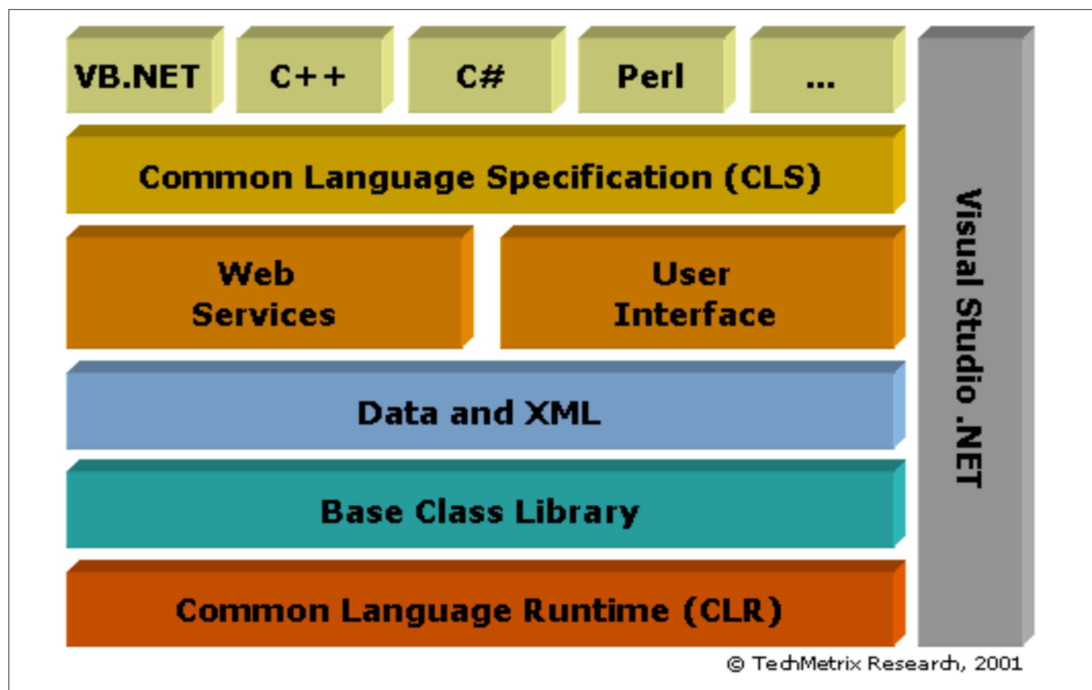


Figure 1.3. *Architecture de l'environnement .Net*

1.3.3.2. Les composants

Les composants .NET sont appelés Objets Distants. Ils peuvent être programmés dans n'importe quel langage et peuvent avoir plusieurs interfaces. Un mécanisme d'introspection permet de naviguer parmi les services proposés par le composant.

1.3.3.3. Déploiement et assemblage

Le *framework* .NET est proposé en partie pour résoudre les problèmes de version et de déploiement. Ces problèmes apparaissent lorsqu'on installe une nouvelle application et qu'elle comporte des composants déjà présents sur la machine, dans une autre version. Ce problème est dû essentiellement aux librairies dynamiques (DLLs). Pour résoudre ce genre de problèmes, l'architecture .NET utilise le fichier de déploiement qui est une librairie Windows (fichier .DLL), avec éventuellement un exécutable (.EXE) et également un nouveau concept appelé assemblage. Il est constituée de :

- Nom du composant et sa version ;
- La spécification des interfaces du composant ;
- Les ressources (textes, images, sons, etc.) ;
- Les composants ou assemblages requis ;
- Les permissions de sécurité ;
- Les méta-données qui donnent des informations sur les types et/ou classes fournies par l'assemblage. Elles sont utilisées pour créer les accès vers les composants (objets proxy). Ces méta-données peuvent être utilisées à la compilation du composant client, ou à partir d'une description WSDL (Web Services Description Language) [Christensen et al. 2001] qui décrit le composant et ses méthodes.

Les informations sur le composant telles que la liste des fichiers, les *dépendances* et d'autres informations utilisées pour exécuter le code sont mises dans un *manifeste*. Il suffit juste de copier les applications dans leur répertoire. Pour concevoir des applications .NET, on doit respecter les contraintes suivantes :

- Les assemblages doivent être assez descriptifs, pour simplifier les activités d'installation et de désinstallation.
- Les assemblages doivent respecter une politique de numérotation des versions.
- L'infrastructure doit permettre à plusieurs versions d'un composant de s'exécuter sur la même machine.

1.3.4. Fractal

Fractal [Bruneton 2004] est un modèle de composants développé par France Télécom R&D et l'INRIA. Fractal est un modèle de composants hiérarchique, modulaire et

extensible pouvant être utilisé pour la conception, l'implantation, le déploiement et la reconfiguration des applications [Bruneton et al. 2003]. Les caractéristiques principales du modèle Fractal sont la *récursivité*, *l'introspection* et le *partage*.

1.3.4.1. Les spécifications

Contrairement à d'autres modèles comme les EJB ou CCM dont les composants sont plutôt de moyenne granularité et destinés aux applications de gestion tournées vers l'Internet, la granularité des composants Fractal est quelconque. Leurs caractéristiques font qu'ils conviennent aussi bien à des composants de bas niveau (par exemple un pool d'objets) que de haut niveau (par exemple une IHM complète). Le but du Fractal est de développer et de gérer des systèmes complexes comme les systèmes distribués. Un composant Fractal est généralement composé de deux parties : une membrane qui possède des interfaces fonctionnelles et des interfaces permettant l'introspection et la configuration (dynamique) du composant, et un contenu qui est constitué d'un ensemble fini de sous-composants (cf. figure 1.4).

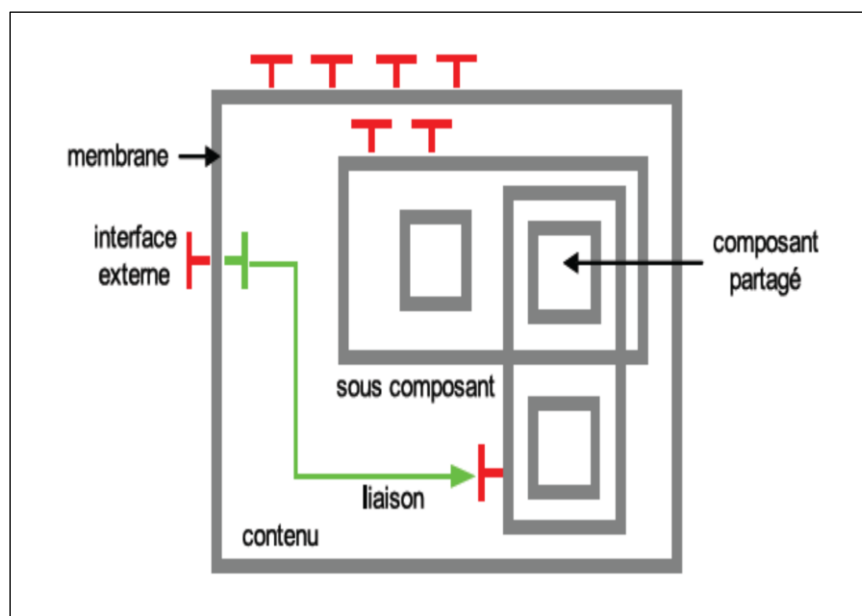


Figure 1.4. *Représentation d'un composant Fractal*

Par analogie avec la biologie, un composant Fractal est une cellule avec un plasma entouré par une membrane. Le plasma peut contenir d'autres cellules. Une membrane contrôle et gère son plasma. En théorie, un composant peut appartenir à deux ou plusieurs plasmas différents. Chaque membrane définit un contexte de nommage et possède des interfaces internes et externes. Les cellules interagissent à l'aide de signaux échangés par les interfaces.

1.3.4.2. Les Composants

Le modèle Fractal fournit deux mécanismes permettant de définir l'architecture d'une application : l'imbrication (à l'aide des composants composites) et la liaison. La liaison est ce qui permet aux composants Fractal de communiquer. Fractal définit deux types de liaisons : primitive et composite. Les liaisons primitives sont établies entre une interface client et une interface serveur de deux composants résidant dans le même espace d'adressage. Par exemple, une liaison primitive dans le langage C (resp. Java) est implantée à l'aide d'un pointeur (resp. référence). Les liaisons composites sont des chemins de communication arbitrairement complexes entre deux interfaces de composants. Les liaisons composites sont constituées d'un ensemble de composants de liaison (e.g. stub, skeleton) reliés par des liaisons primitives.

1.3.4.3. L'Assemblage

Le langage Fractal ADL permet de décrire, à l'aide d'une syntaxe XML, des assemblages de composants Fractal (cf. figure 1.5).

```
<definition name="AbstractClientServer" extends="RootType">
        <component name="client" definition="ClientType"/> <!-- pas d'implémentation -->
        <component name="server" definition="ServerType"/> <!-- pas d'implémentation -->
        <binding client="this.m" server="client.m"/>
        <binding client="client.s" server="server.s"/>
</definition>
```

Figure 1.5. *Un exemple de définition d'une ADL à l'aide de Fractal ADL*

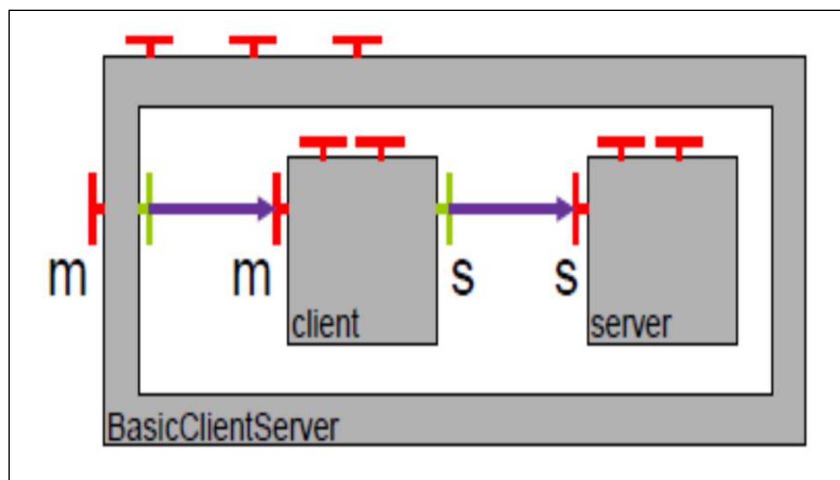


Figure 1.6. *Assemblage Fractal correspondant à l'ADL de la figure 1.5*

La figure 1.6 donne un exemple de définition réalisée à l'aide du Fractal ADL. Le composant décrit est un composite dont le nom est BasicClientServer. Ce composite

possède une interface serveur, de nom m. Par ailleurs, le composite encapsule deux composants : Client et Server. La définition du composant Client est intégrée à celle du composant BasicClientServer : le composant a deux interfaces (m et s), il possède une partie de contrôle de type primitive. La définition du composant Server suit le même principe avec une interface serveur s. Enfin, la description ADL mentionne deux liaisons : entre les interfaces m du composite et du Client et entre les interfaces s du Client et du Server.

1.4. Ingénierie dirigée par les modèles

L'ingénierie dirigée par les modèles et, en particulier, les processus de développement logiciel à base de modèles ont toujours été au centre des préoccupations des chercheurs. Ils correspondent à un paradigme dans lequel le code source n'est plus considéré comme l'élément central d'un logiciel, mais comme un élément dérivé d'éléments de modélisation. Cette approche prend toute son importance dans le cadre des architectures logicielles et matérielles en utilisant des standards tels que les spécifications MDA (Model-Driven Architecture) proposées par l'OMG [Atkinson et al. 2001]. De telles architectures s'intègrent tout naturellement dans un processus de développement à base de modèles s'assurant, à chaque niveau de modélisation, que les modèles obtenus ont les qualités requises. Cette démarche dirigée par les modèles met le modèle au centre des préoccupations des analystes/concepteurs. Leur élaboration devient donc centrale et le choix du formalisme revêt une importance capitale. Actuellement deux tendances se dégagent : la première est plus généraliste comme UML et la seconde répond davantage aux exigences d'un domaine avec les «Domain Specific Languages» (DSL) et les « Domain Specific Model Language » (DSML). Dans le premier cas, le processus de développement est plus long car on part d'un modèle plus abstrait. Le second est plus ciblé car les concepteurs peuvent s'appuyer sur des technologies propres à un domaine. Cependant, en débutant le processus avec UML, les modèles pourront être plus facilement réutilisables, par contre en se basant, dès le départ du processus, sur un langage de modélisation spécifique au domaine, le processus restera l'affaire des experts du domaine. Quelle attitude avoir quand on cherche à mettre en place un processus de développement dirigé par les modèles ? Les recherches menées jusqu'à ce jour dans le cadre de collaborations académiques/industrielles, montrent que ces technologies intéressent de plus en plus les industriels. Elles ont pour effet non négligeable de réduire le temps entre conception, mise au point, production et maintenance des logiciels tout en garantissant toutes les qualités que l'on exige d'eux.

1.4.1. Apport de l'ingénierie dirigée par les modèles

L'objectif principal d'une approche d'ingénierie dirigée par les modèles (IDM) [Bézivin et al. 2004] est de reporter la complexité d'implémentation d'une application au niveau de sa spécification. Cela devient alors un problème d'abstraction de langage de programmation en utilisant un processus de modélisation abstrait fondé sur l'utilisation de plusieurs standards tels que MOF, OCL, UML et XMI.

L'architecture dirigée par les modèles (MDA) [Mellor 2004] [Miller et al. 2003] est un champ spécifique de l'IDM pour spécifier une architecture à 3 niveaux :

- Une représentation d'un modèle indépendant (CIM). Le CIM décrit le contexte dans lequel les systèmes seront utilisés ;
- Une représentation d'un modèle indépendant de toute plateforme (PIM), à partir de la représentation CIM, elle décrit le système lui-même sans aucun détail sur l'utilisation de la plateforme. Une représentation PIM sera traduite par une ou plusieurs architectures de plates-formes réelles ;
- Une représentation d'un modèle spécifique à une plate-forme (PSM) à partir de la représentation PIM.

A ce niveau, l'environnement des plates-formes ou des langages d'implémentation sont connus.

Le MDA permet de concevoir une application de gestion de flux à partir des différentes traductions depuis la représentation CIM jusqu'à la représentation PSM. Ces transformations peuvent être automatisées, notamment quand un méta-modèle spécifie les règles de transformation entre les différents modèles impliqués. Par ce biais, une approche MDA augmente l'interopérabilité dans des environnements hétérogènes et fournit une méthode d'intégration des systèmes en utilisant des moteurs génériques de transformation.

2.4.2. Modèle de composants UML 2.0

UML 2.0 introduit un certain nombre de nouveaux concepts par apport à la précédente version et affine un certain nombre de concepts existants [Booch et al. 2005]. Cette section offre un bref aperçu des concepts les plus pertinents pour les architectures logicielles. Cinq concepts en UML 2.0 méritent notre attention qui seront abordés ci-après :

1. Interfaces,
2. Ports,
3. Composants,
4. Connecteurs,
5. Structure composite.

2.4.2.1. Interface

UML 2.0 étend le concept d'interface pour inclure explicitement les interfaces fournies et requises. Les interfaces requises ont été introduites pour compléter les interfaces fournies et décrire les caractéristiques d'un service qu'un composant utilise pour accomplir ses fonctionnalités.

2.4.2.2. Port

Le port est un nouveau concept dans UML 2.0 semblable à une interface en ce qu'il décrit comment un composant interagit avec son environnement, mais différent en ce sens que chaque port est un point d'interaction distinct du composant. Les ports peuvent avoir des types, et un composant peut spécifier la multiplicité d'un port. Chaque port peut être associé à un certain nombre d'interfaces (fournies et/ou requises), dans une collection qui soit logique du point de vue interaction.

La figure 1.7 montre un port simple (sans interfaces associées). Le port P est représenté par un rectangle sur la frontière du composant Server et dispose d'une cardinalité 1..* ; ce qui indique que chaque instance du serveur aura un ou plusieurs ports P. La figure 1.8 étend cet exemple pour montrer l'association des interfaces spécifiques (fournies et requises) avec le port.

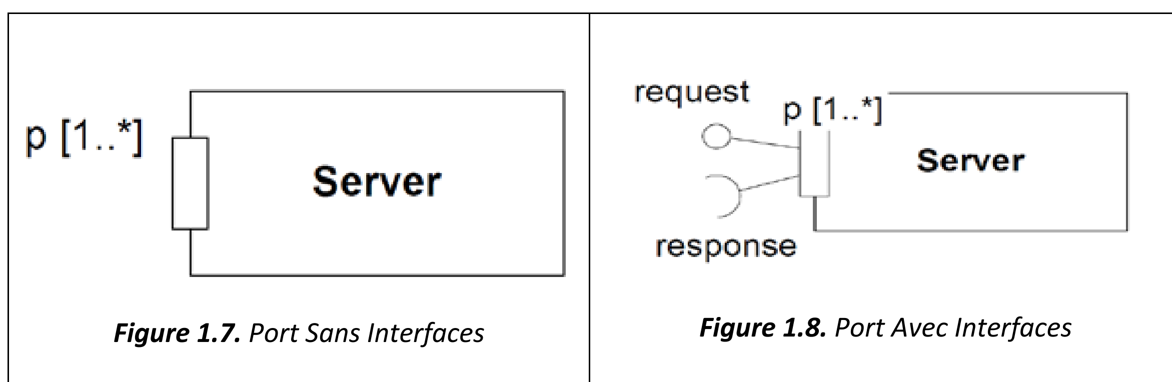


Figure 1.7. *Port Sans Interfaces*

Figure 1.8. *Port Avec Interfaces*

1.4.2.3. Composant

En UML 2.0, un composant est "une partie d'un système modulaire qui encapsule son contenu". Cette généralisation permet au composant d'être utilisé pour décrire la conception ou la mise en œuvre des applications.

Bien que la représentation graphique des éléments en UML 1.4 (comme indiqué sur la droite de la figure 1.9) soit toujours prise en charge pour la compatibilité descendante. Deux autres représentations sont disponibles dans UML 2.0.

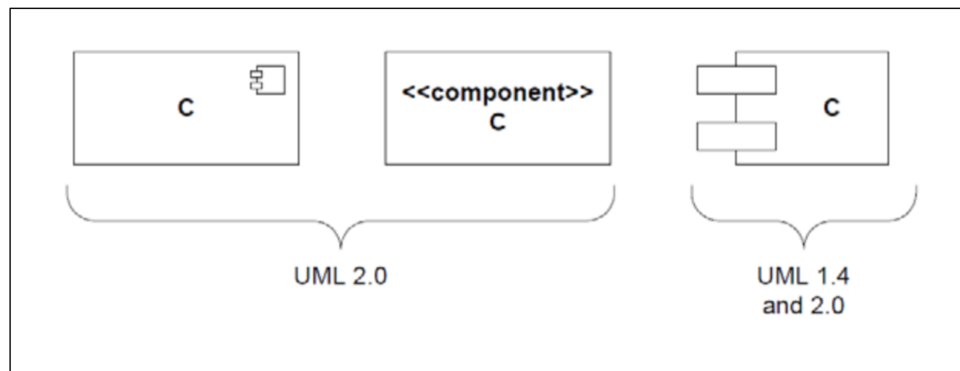


Figure 1.9. *Représentation des composants en UML 2.0*

1.4.2.4. Connecteurs

Le connecteur est un nouveau concept dans UML 2.0. Il représente un lien de communication entre deux ou plusieurs instances de composants. Les connecteurs peuvent être réalisés par une variété de mécanismes, tels que des pointeurs ou des connexions réseau. Formellement, un connecteur est juste un lien entre deux ou plusieurs éléments connectables (par exemple, les ports ou les interfaces), il ne peut pas être associé à une description de comportement ou des attributs qui caractérisent la connexion. Et donc, il ne peut pas avoir une signification proche de celle du composant et il ne peut pas assurer de traitements.

1.4.2.5. Structure composite

Il existe deux types de composants en UML2.0 : le composant atomique et le composant composite. La première catégorie définit le composant comme un élément exécutable du système. La deuxième catégorie étend la première en définissant le composant comme un ensemble cohérent de parties. Chaque partie représente une instance d'un autre composant. La figure 1.10 montre la vue externe dite encore boîte noire d'un composant. Le composant offre deux ports entree1 et entree2 et requiert un port sortie.

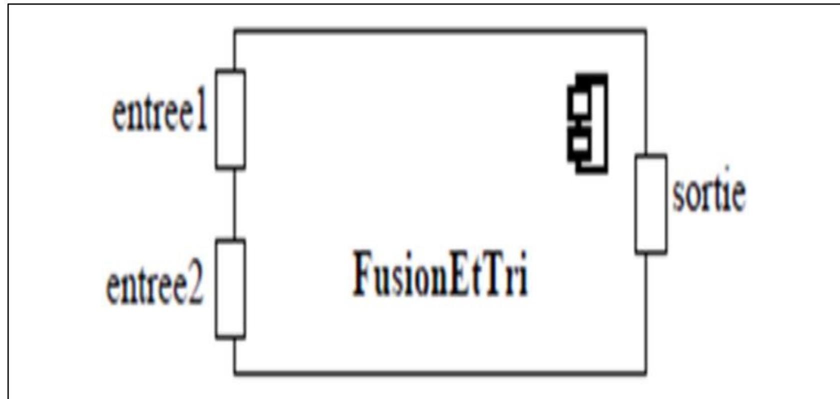


Figure 1.10. *Vue externe d'un composant*

La figure 1.11 montre la vue interne dite encore boîte blanche d'un composant. Ce dernier est composé de deux sous-composants. La direction des connecteurs de délégation indique que les deux ports entree1 et entree2 sont typés avec des interfaces offertes et le port de sortie est typé avec une interface requise. Les sous composants sont connectés par un connecteur d'assemblage non nommé.

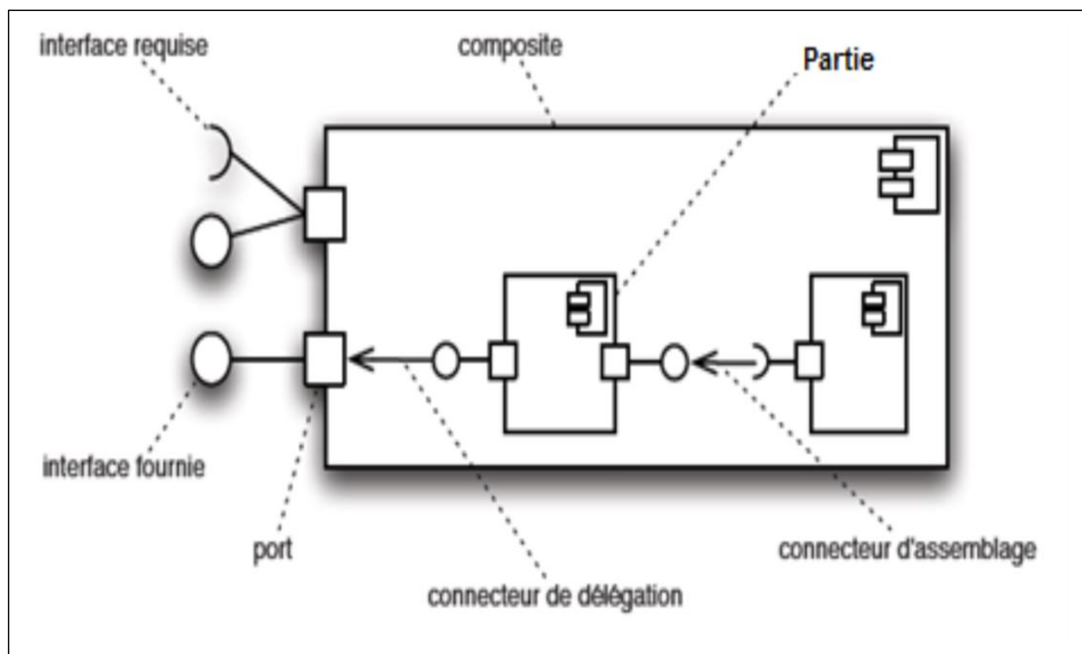


Figure 1.11. *Vue interne d'un composite*

A partir d'UML 2.0, un pas vers la description d'architecture a été franchi par rapport aux versions précédentes. Les concepts de port, de connecteur et de diagramme d'architecture ont été introduits ce qui permet à UML d'être un concurrent des ADL.

UML permet aussi bien les descriptions structurelles des architectures que les descriptions comportementales via les protocoles notamment. La plupart des ADL ont les mêmes objectifs qu'UML à savoir la modélisation des systèmes. Depuis sa

version 2.0, UML est bien mieux adapté pour décrire des architectures logicielles qu'il ne l'était dans ses versions précédentes [Medvidovic et al. 2002]. Actuellement, non seulement il supporte les principales constructions des ADL mais il offre aussi un mécanisme d'extension (sous la forme de profil) puissant qui permet de l'adapter lorsqu'il n'offre pas en standard un concept ou un mécanisme dont on pourrait avoir besoin.

Malgré les avantages apportés par sa dernière version, UML souffre de plusieurs problèmes dont le principal est sa complexité. Il est en effet difficile de maîtriser complètement les mécanismes UML (surtout les plus récents) qui manquent souvent de précision dans leur spécification. C'est ainsi que des interfaces peuvent être attachées directement à un composant alors qu'elles devraient toujours être attachées à un port d'un composant pour donner plus de sens à ce concept.

2.4.3. Architectures logicielles et UML

Les quatre niveaux du méta-modélisation appliqués au langage UML (modèle d'un système, modèle UML, métamodèle UML, méta-méta modèle général) induisent trois stratégies potentielles pour modéliser des architectures logicielles en UML [Medvidovic et al. 2002]. Ces stratégies sont:

- L'utilisation directe d'UML ;

- L'utilisation des mécanismes d'extensibilité d'UML pour contraindre le méta-modèle UML afin de l'adapter aux concepts architecturaux ;

- L'augmentation du métamodèle UML afin de supporter directement et explicitement les concepts architecturaux ;

Chaque stratégie a des avantages et des inconvénients. Dans la suite, nous présentons et nous évaluons ces trois stratégies en s'inspirant des travaux effectués par [Medvidovic et al. 2002] [Zarras et al. 2001] [Roh et al. 2004] [Goulao et al. 2003] [Kandé et al. 2000] [Garlen et al. 2002].

1.4.3.1. Utilisation directe d'UML

Cette stratégie consiste à utiliser les constructions offertes par UML pour représenter les concepts architecturaux des ADL tels que composant, connecteur, rôle, port et configuration. Garlan et al. [Garlen et al. 2002] proposent et évaluent quatre approches permettant de représenter les principaux concepts architecturaux en UML 1.x. Ces approches sont centrées sur la représentation des types des composants et

des instances de composants en UML 1.x. Ainsi, les auteurs proposent les alternatives suivantes :

- Classes et objets : les types des composants sont représentés par des classes UML1.x et leurs instances par des objets ;

- Classes et classes : les types et les instances de composants sont représentés par des classes ;

- Composants UML 1.x : les types des composants sont représentés par des composants UML 1.x et les instances de composants par des instances de composants UML 1.x ;

- Sous-systèmes : les types des composants sont représentés par des sous-systèmes UML et les instances de composants par des instances des sous-systèmes.

L'avantage majeur de l'utilisation d'UML en tant que tel pour la modélisation des architectures logicielles décrites par des ADL est la compréhension de cette modélisation par tout utilisateur d'UML. De plus, une telle modélisation peut être manipulée par des ateliers UML supportant les autres étapes du développement : expression des besoins, conception et implémentation. Quant à l'inconvénient inhérent à cette stratégie, il concerne l'incapacité d'UML en tant que tel –notamment UML1.x- à traduire d'une façon explicite les concepts architecturaux.

1.4.3.2. Utilisation des mécanismes d'extensibilité d'UML

UML est un langage de modélisation "généraliste" pouvant être adapté à chaque domaine grâce aux mécanismes d'extensibilité offerts par ce langage tels que stéréotypes, valeurs marquées (tagged values) et contraintes. Les extensions UML ciblant un domaine particulier forment des profils UML. Les mécanismes d'extensibilité offerts par UML permettent d'étendre UML sans modifier le méta-modèle UML. Plusieurs travaux permettent d'adapter aussi bien UML1.x qu'UML2.0 aux architectures logicielles.

L'utilisation de profils pour adapter UML au domaine des architectures logicielles constitue un avantage important : expliciter la représentation des concepts architecturaux sans modifier le méta-modèle UML. De plus, ces profils peuvent être manipulés par des ateliers UML supportant le concept de profil.

Actuellement, ces ateliers sont de plus en plus nombreux tels que Softeam UML Modeler (http://www.objecteering.com) [Objecteering 2015] et IBM Rational Software Modeler (http://www.rational.com) [Rational 2015]. La contrepartie pour

les utilisateurs est de devoir maîtriser notamment les contraintes -décrites souvent en OCL- associées aux stéréotypes formant ces profils.

1.4.3.3. Augmentation du métamodèle UML

Cette stratégie consiste à ajouter de nouveaux éléments de modélisation (de nouvelles constructions syntaxiques) en modifiant directement le métamodèle UML. Ceci donne un nouveau langage de modélisation supportant de façon native les concepts architecturaux. L'augmentation du métamodèle UML afin de couvrir des besoins architecturaux a les deux inconvénients majeurs suivants :

- La nouvelle version d'UML devient de plus en plus complexe. Ceci a un impact sur la facilité d'utilisation de ce langage ;

- La nouvelle version d'UML perd son caractère standard et par conséquent elle devient incompatible avec les ateliers UML existants.

1.5. Conclusion

Dans ce premier chapitre, nous avons essayé de donner un aperçu plus ou moins détaillé sur les composants logiciels avec ses éléments de base qui les constituent. Dans cet aperçu on a présenté les définitions relatives à la majorité des concepts et des notions en relation avec un composant logiciel. Egalement, on a présenté quelques modèles à base de composants ayant connu des réussites notamment sur le plan industriel. L'architecture dirigée par les modèles (Model Driven Architecture) de l'OMG a été proposée pour situer une telle démarche à base de composants. Et finalement, nous avons conclu par le modèle unifié UML 2.0.

Dans le chapitre suivant, nous discutons notre première approche pour construire un composant logiciel à partir de composants existants. Cette approche est connue sous le nom <<assemblage>>. La spécificité de notre proposition au niveau assemblage est basée sur le paradigme vue/point de vue.

Chapitre 2

Composants logiciels multi-vues

2.1. Introduction

Le développement des applications reste toujours une tâche très coûteuse et complexe. L'idée d'acquérir ce qui est disponible pour développer ce qui est spécifique est une solution évidente pour réduire la complexité du développement des logiciels. En s'inspirant du concept des composants utilisés dans plusieurs domaines (électroniques, mécanique, ...), un logiciel peut aussi être construit par assemblage de briques logiciels réutilisables dits - **composants logiciels** -. Le composant, comme son nom le dit, c'est pour être composé et interagir avec d'autres via des interfaces fournies et requises.

L'utilisation des composants logiciels vise à réduire davantage les coûts de développement d'une application par la réutilisation et l'assemblage de composants préfabriqués comme les pièces d'un puzzle.

L'ensemble des moyens nécessaires pour interconnecter les composants en synchronisant les sorties des uns avec les entrées correspondantes des autres sont appelés les **connecteurs**. Ils peuvent être vus comme un mode de communication de deux ou plusieurs composants basée sur la synchronisation de leurs services fournis et requis. Ces connecteurs représentent les moyens logiciels explicites permettant d'assurer l'assemblage des composants.

Par ailleurs, chacun s'accorde à reconnaître l'intérêt de prendre en compte l'utilisateur -au sens large du terme- dans le développement des systèmes complexes. Le concept de point de vue est un moyen pertinent pour mettre en œuvre cette préoccupation [Carré et al. 1991] [Finkelstein et al. 1990]. Les concepts de vue/point de vue ont été étudiés dans plusieurs domaines de l'informatique, parfois sous d'autres termes tels que sujet [Harrison et al. 1993], séparation multidimensionnelle des préoccupations [Tarr et al. 1999], rôle [Kristensen 1996], aspect [Kiczales et al. 1997], etc. Dans le cadre d'intégration du concept de vue dans la modélisation

orientée objet, le langage de programmation VBOOL [Marcaillou et al. 1995] et la méthode associée VBOOM [Kriouile 1995] ont été définis et plus récemment le profil VUML a été proposé par M. Nasser [Nasser 2005][Anwar et al. 2010]. Mes premières contributions dans ce domaine étaient de proposer la méthode unifiée d'analyse et de conception U_VBOOM fondée sur le concept de vue et basée sur la méthode VBOOM [Hair 2004a][Hair 2004b][Hair 2005], et j'ai implémenté la relation de visibilité introduite par le langage VBOOL afin d'avoir une génération de code multiple [Hair 2004c].

En se basant sur le concept de vue/point de vue et le composant logiciel, nous allons proposer un modèle permettant de faciliter le développement d'applications à base de composants logiciels. Ce modèle que nous présenterons permettra la construction d'un composant logiciel appelé "**Composant Logiciel Multi-vues**" (**CLM**) par assemblage d'un composant logiciel (composant base) et des composants logiciels (composants vues). La spécificité d'un composant logiciel multi-vues est la possibilité d'avoir des interfaces dont l'accessibilité et le comportement changent dynamiquement selon le point de vue de l'utilisateur courant. C'est à dire le client peut activer/désactiver des *composants vues* durant l'exécution. Nous introduisons ainsi la notion de visibilité dans les modèles de composants logiciels. La réification de cette relation est réalisée par le nouveau type de connecteur que nous proposons, ***connecteur de visibilité***.

Nous consacrons ce chapitre à la présentation de notre proposition concernant un modèle de composants logiciels multi-vues (CLM). Nous commençons tout d'abord par la présentation de l'étude que nous avons établie sur la relation de visibilité et de ses propriétés au sein de l'étude de la modélisation multi-vues. Après, nous introduisons la relation de visibilité pour les modèles de composants logiciels en proposant de nouveaux concepts tels que le composant base, le composant vue et le connecteur de visibilité. Et, nous terminons ce chapitre par la présentation du processus de conception des composants logiciels multi-vues.

2.2. Modélisation multi-vues

2.2.1. Projet VBOOM

Le projet VBOOM (View Based Object Oriented Methodology) a été lancé en 1992, il s'est déroulé en partenariat avec le laboratoire mixte ARAMIIHS de Toulouse et le laboratoire d'Informatique de l'ENSIAS de Rabat. Ce projet a donné lieu à plusieurs travaux de recherche [Coulette et al. 1995]. Ces travaux ont visé l'introduction du concept de vue et point de vue dans le cadre d'une modélisation par objets. L'objectif

était d'apporter une plus grande souplesse, flexibilité et rigueur dans la modélisation à objets dans un contexte de Génie Logiciel. Le modélisateur d'un système a priori complexe (par exemple un satellite) doit pouvoir définir des vues multiples sur ce système, puis accéder à l'une ou à plusieurs de ces vues selon ses besoins particuliers. Pour cela, un nouveau concept « la classe flexible », et une nouvelle relation appelée « visibilité », intégrés au sein d'une extension d'Eiffel [Meyer 1992] appelée VBOOL – View Based Object-Oriented Language – [Marcaillou 1995], ont été définis. Dans le même contexte, une méthode d'analyse et de conception par objet VBOOM – View Based Object Oriented Method – [Kriouile 1995] supportant les concepts de vue et point de vue a été élaborée. Cette méthode permet de modéliser un système selon les points de vue de ses différents utilisateurs. Ceci conduit à élaborer un certain nombre de modèles partiels appelés « modèles visuels ». Une fois ces modèles réalisés – éventuellement de façon décentralisée – une étape de fusion de ces modèles est proposée par la méthode VBOOM. Cette fusion porte sur les interfaces des classes composant les modèles partiels. Elle a été formalisée comme une loi de composition interne sur l'ensemble des interfaces de classes. Un prototype d'environnement supportant la méthode, appelé VBTOOL [Marzak 1997] [Hair 1997] [Hair et al. 1998], a été réalisé.

Afin de satisfaire les besoins de modélisation des systèmes complexes dans un contexte de Génie Logiciel, Marcaillou et al ont étendu l'approche orientée objet en intégrant les points de vue dans la modélisation par objet. L'implantation de la notion de vue et de point de vue est réalisée en définissant une nouvelle relation appelée visibilité. Celle-ci, proche de l'héritage, permet de sélectionner les informations contenues dans une classe et de les filtrer vers les classes dérivées, selon un principe analogue à la définition des statuts d'exportation en délégation [Marcaillou et al. 1996]. Pour supporter la visibilité, un langage fortement typé appelé VBOOL (View Based Object Oriented Language) a été défini.

2.2.2. Relation de visibilité : Définition et propriétés

2.2.2.1. Définition

La relation de visibilité s'applique quand on veut exprimer le fait qu'une "entité" peut être considérée sous des "angles" multiples [Coulette et al. 1995]. C'est bien le cas d'une Médiathèque, par exemple, qui est appréhendée sous l'angle des prêts, des exemplaires, des comptes adhérents, etc. Dans le langage de programmation VBOOL [Marcaillou et al. 94] l'entité Médiathèque est représentée par une classe multi-vues

qui déclare des liens de visibilité avec les classes Prêts, Exemplaires, Comptes_Adhérents appelées "vues".

La relation de visibilité est donc une relation qui lie une entité (entité base) avec ses vues (entités vues) par un ensemble de liens de visibilité. Elle assure la sémantique "*est vue comme*" qui caractérise l'assemblage d'une entité base avec ses entités vues. Elle est représentée par un graphe (cf. figure 2.1), dénommé graphe de visibilité, qui peut dériver à un multi-graphe si au moins une de ses vues est une entité base (c'est la propriété de transitivité, qui sera définie par la suite).

Les entités multi-vues sont donc des entités complexes créées par l'assemblage d'une entité base et d'un ensemble d'entités vues à l'aide de la relation de visibilité. En d'autres termes, une entité multi-vues est formée d'une entité base et de plusieurs entités vues qui décrivent chacune un type d'utilisation de l'entité multi-vues [Hair 2004c].

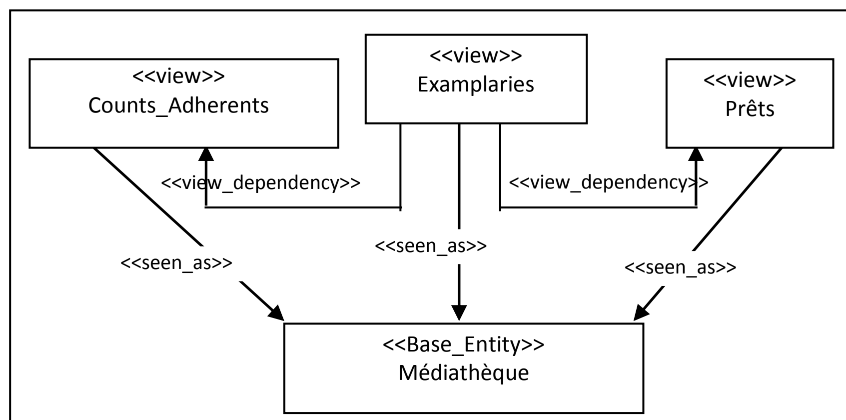


Figure 2.1. *Graphe de visibilité*

2.2.2.2. Propriétés

Parmi les propriétés les plus essentielles de cette relation, on trouve [Hair 2006] :

La transitivité : La relation de visibilité est transitive lorsque les mêmes types de relation de visibilité sont impliqués dans les prémisses.

L'antisymétrique : Une *entité base* est vue comme une *entité vue*, mais l'inverse n'est pas possible. Par exemple : l'*entité base* "Médiathèque" est vue comme une *entité vue* "Prêts" alors que l'entité "Prêts" ne peut pas être vue comme une entité vue "Médiathèque" ;

La multiplicité : Une entité vue ne peut être reliée qu'à une et une seule entité de base (cette propriété est vérifiée pour des liens de visibilité). Une entité vue ne doit pas être partagée entre deux entités de base.

La dépendance : C'est une propriété importante qui permet la propagation des modifications entre les entités vues dépendantes. Le sens de la propagation peut être celui de l'entité vue source vers l'entité vue destinataire (par exemple : le prêt d'un livre effectué par un adhérent modifie le nombre d'exemplaires disponible de ce livre, ainsi les deux entités vues "Prêts" et "Exemplaires" sont dépendantes) ;

L'exclusion mutuelle : Cette propriété consiste à réaliser le droit d'accès sur un modèle multi-vues. L'activation d'une entité vue V1 qui présente l'exclusion mutuelle avec une autre entité vue active V2 ne peut se produire que si l'entité vue V2 est désactivée et inversement.

2.3. Modèle de composants logiciels multi-vues

Nous consacrons cette partie à la présentation de notre proposition concernant le modèle de composants logiciels multi-vues. Nous commençons par l'introduction de la relation de visibilité pour les composants logiciels et les nouveaux concepts liés. Ensuite, nous présentons le modèle de composants logiciels multi-vues [Hair 2016].

2.3.1. Relation de visibilité pour les composants logiciels

Pour pouvoir introduire la relation de visibilité dans les composants logiciels, nous devons définir de manière très précise la notion de composant logiciel multivues ainsi que l'ensemble de propriétés de la relation de visibilité pour les composants logiciels.

Un composant logiciel multi-vues (CLM) est un composant logiciel qui peut être considéré et appréhendé sous des "angles" multiples, dénommés ses composants vues. Un composant logiciel multi-vues a un composant logiciel appelé composant base lié avec des liens de visibilité avec les composants vues.

A partir de cette définition nous pouvons établir les règles à respecter pour pouvoir introduire la notion de composant logiciel multi-vues dans les modèles de composants logiciels muti-vues :

Règle 1 : un CLM est avant tout un composant logiciel qui publie des interfaces fournies et requises.

Règle 2 : un CLM est constitué d'un *composant base* et des *composants vues*. Un *composant base* est lié à ses *composants vues* par des liens de visibilité.

Règle 3 : les liens de visibilité doivent refléter les propriétés de la relation de visibilité telles qu'elles ont été citées dans la section 2.2 (la dépendance, l'exclusion mutuelle, etc.).

Règle 4 : un composant vue ne peut s'appliquer que sur un unique composant base.

Règle 5 : les composants vues sont accessibles directement tant qu'ils ne font pas partie d'un CLM. A partir du moment où ils deviennent des composants vues d'un CLM, ils deviennent inaccessibles qu'à travers leur CLM.

Règle 6 : un CLM doit offrir la possibilité de changer dynamiquement son comportement et son accessibilité selon le type d'utilisation.

2.3.2. Présentation du modèle de composants logiciels multi-vues

Dans l'approche "composant", une application logicielle est construite par une collection de composants logiciels interconnectés à l'aide de connecteurs [Traverson et al. 2002] [Nikunj et al. 2000]. Dans notre modèle, le lien de visibilité est traduit en un connecteur appelé *connecteur de visibilité* qui est un composant logiciel. Il possède ses interfaces et ses propriétés et s'interpose entre le composant base et le composant vue pour réifier la sémantique de la relation de visibilité (cf. figure 2.2). Par conséquent, le connecteur de visibilité doit assurer les propriétés de la relation de visibilité et plus particulièrement les deux propriétés : la dépendance et l'exclusion mutuelle.

La dépendance : Pour assurer la relation de dépendance entre deux composants vues, une variable entière sera stockée dans le connecteur de visibilité correspondant. Cette variable doit être testée par toutes les opérations concernant la modification des informations liées des deux composants vues.

L'exclusion mutuelle : Pour assurer le droit d'accès entre deux composants vues en exclusion mutuelle, une variable entière sera stockée dans les connecteurs de visibilité correspondants. Cette variable doit être testée par toutes opérations d'activation d'un composant vue en exclusion mutuelle.

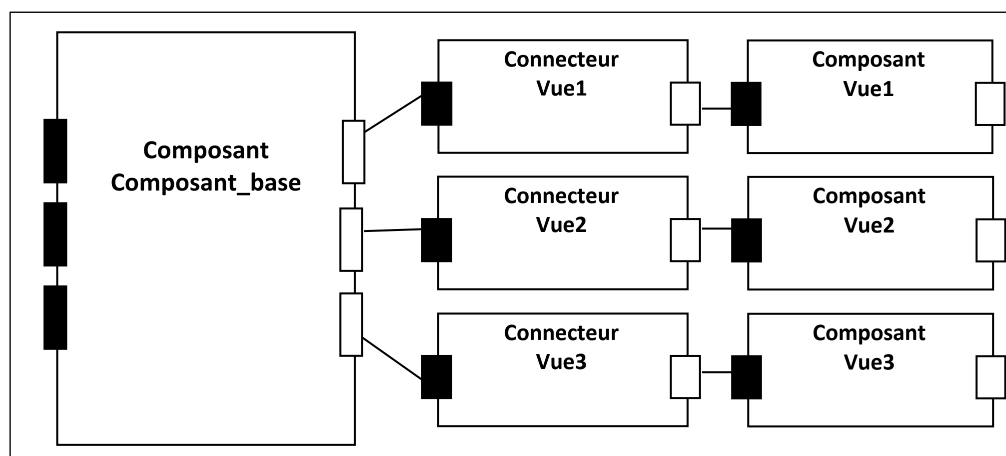


Figure 2.2. *Modèle de composant du CLM*

2.3.3. Processus de conception de CLM

L'utilisation de CLM nécessite la définition d'un ensemble d'opérations permettant d'assurer les services associés à la relation de visibilité. Les opérations de manipulation de CLM sont "activer/désactiver" un composant vue faisant un lien de visibilité avec un composant base et l'invocation de l'une des interfaces du CLM. Il est nécessaire de vérifier la cohérence de la sémantique de la relation de visibilité lors de l'utilisation de chaque opération. Nous soulignons que les interfaces fournies par le CLM incluent les interfaces fournies par ses composants vues. L'utilisation de ces derniers est spécifiée lors de la définition des liens de visibilité.

2.4. Processus de transformation et mise en œuvre des CLM

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

2.4.1. Présentation du processus de transformation

Conformément à la démarche MDA (Model Driven Architecture), notre modèle de CLM est un modèle indépendant de toute plate-forme (PIM) [Pastor et al. 2013]. A partir de ce modèle, un processus de transformation a été défini et utilisé pour les projections CCM [OMG 2002c] [OMG 2007] et EJB [Greenfield 01] [Edwards et al. 2002]. La figure 2.3 illustre le processus de transformation utilisé.

Dans un premier temps, un nouveau modèle est automatiquement généré à partir du modèle indépendant fourni. Ce nouveau modèle correspond à la copie conforme du modèle d'origine à laquelle on ajoute un ensemble de décisions spécifiques à la plateforme cible. Ces décisions spécifiques ne peuvent être précisées dans le modèle indépendant.

Une fois les décisions spécifiques à la plate-forme cible sont ajoutées, une autre transformation délivre le modèle UML spécifique à la plate-forme choisie. Ce modèle est un PSM (Platform Specific Model) dans le contexte MDA. Les règles de transformation sont automatiquement appliquées en fonction des décisions spécifiques à la plate-forme cible.

Finalement, le concepteur peut adapter le modèle spécifique (PSM) obtenu, en particulier en définissant ses choix d'implantation. Ces modèles spécifiques à une plateforme particulière sont exprimés à l'aide de profils UML. Les éléments spécifiques à la plate-forme CCM sont exprimés à l'aide du langage UML [Booch et al. 2005] et du profil CCM (fondé sur le profil CORBA). Dans la plate-forme CCM, l'assemblage est exprimé à l'aide de fichiers XML, appelés "descripteurs d'assemblage". Ceux-ci sont également générés par l'outil sous la forme de notes UML.

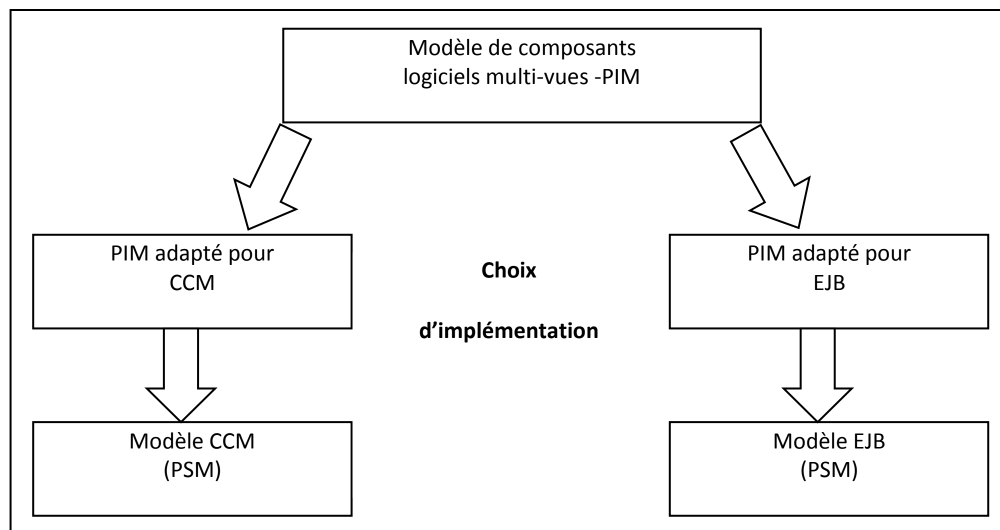


Figure 2.3. *Processus de transformation*

2.4.2. Mise en œuvre des CLM en CCM

2.4.2.1 Processus de conception de CLM en CCM

Dans la partie précédente, nous avons introduit la relation de visibilité pour les modèles de CLM. Dans cette partie, nous allons mettre en œuvre ce modèle pour les composants CCM. Pour pouvoir définir et utiliser des CLM dans le modèle CCM, nous allons procéder comme suit [Hair 2007] :

- La description du CLM ou de son utilisation en VIEW-IDL3 ;
- La projection de cette description ou de cette utilisation vers IDL3 par le compilateur VIEW-IDL3 ToIDL3 ;
- L'interconnexion entre le composant base, les connecteurs de visibilité et les composants vues.

2.4.2.1.1. Description des CLM : le langage VIEW-IDL3

La description d'un type de CLM nécessite l'introduction de nouveaux concepts. Ces derniers sont ajoutés à ceux définis par le langage IDL3 [ACCORD 2002]. La description d'un CLM, à l'aide de VIEW-IDL3 définit principalement les éléments suivants : le module de la classe composant logiciel multi-vues, son composant base, ses composants vues et ses connecteurs de visibilité (liens de visibilité) et les propriétés de ses liens de connexion. La déclaration du composant composant base se fait à l'aide du nouveau mot-clé **base_composant**.

```
base_component nom_Composant_base {..};
```

La déclaration de composants vues se fait par le nouveau mot-clé **view_composant**.

```
view_component nom_composant_vue {..};
```

L'exemple suivant montre la description d'un CLM Médiathèque, son composant base et les composants vues, ainsi que les propriétés des connecteurs de visibilité. La déclaration en VIEW-IDL3 des CLM se fait tout simplement comme celle en IDL3 (Tableau 2.1).

```
module demoMediatheque {
//déclaration des interfaces
interface LoansInterface {
liste opérations….. ;};
interface Counts_AdherentsInterface {
liste opérations….. ;};
interface ExamplariesInterface {
liste opérations….. ; };
================================================================================
//déclaration du composant base_component  du CLM
base_component Mediathèque {
attribute string the_name;
type : asynchrone ;
port_out LoansInterface ;
port_out ExamplariesInterface ;
port_out Counts_AdherentsInterface ;
port_in LoansInterface ;
port_in ExamplariesInterface ;
port_in Counts_AdherentsInterface ;
================================================================================
//déclaration du Composant vue Loans
view_component  Loans {
attribute string the_name;
//déclaration des variables de la relation de visibilité
active : false ;
index: 1 ;
dependency : 0 ; // composant vue indépendant des autres composants vues
mutual_exclusion  : 0 ; // composant vue ne présentant aucune exclusion mutuelle
```

```
};
==========================================================================================
//déclaration du composant vue Examplaries
view_component  Examplaries {
attribute string the_name;
//déclaration des propriétés de la relation de visibilité
active : false ;
index : 2 ;
dependency : 1 ; // indice du composant vue dépendant : Loans
mutual_exclusion  : 3 ; // indice du composant vue qui présente l'exclusion mutuelle avec ce
                                   //composant vue
};
==========================================================================================
//déclaration du composant vue  Counts_Adherents
view_component  Counts_Adherents {
attribute string the_name;
//déclaration des propriétés de la relation de visibilité
active : false;
index :3;
dependency : 1 ; // indice du composant vue dépendant : Loans
mutual_exclusion  : 3 ; // indice du composant vue qui présente l'exclusion mutuelle avec ce
 };                                  // composant vue
==========================================================================================
//La déclaration de la maison "home" du composant base_component Médiathèque
home MédiathèqueHome manages Médiathèque { };
==========================================================================================
// La déclaration du composant Loans
component Loans {
attribute string the_name;
provides LoansInterface        Loans_ViewInterface;
};
//déclaration de la maison "home" du composant Loans_View
home Loans_ViewHome manages Loans { };
==========================================================================================
// La déclaration du composant Counts_Adherents
component Counts_Adherents {
attribute string the_name;
provides Counts_AdherentsInterface     Counts_Adherents_ViewInterface;
};
//déclaration de la maison "home" du composant Counts_Adherents
home Counts_Adherents_View Home manages Counts_Adherents { };
==========================================================================================
// La déclaration du composant Examplaries
component Examplaries {
attribute string the_name;
provides ExamplariesInterface    Examplaries_ViewInterface ;
};
//déclaration de la maison "home" du composant Examplaries
home Examplaries_ViewHome manages Examplaries { };
} ;
```

Tableau 2.1. *Le fichier de déclaration du CLM Médiathèque en VIEW-IDL3*

L'ensemble des définitions que nous avons introduit au langage IDL3 sont résumées dans le tableau ci-dessous (Tableau 2.2) :

CONCEPT	**LA DESCRIPTION DU CONCEPT**
base_component	Pour spécifier un type de *composant base.*
view_component	Pour spécifier un type de composant vue.
index	Chaque composant vue a son propre indice et l'identifie des autres composants vues.
dependency	S'il est différent de zéro alors il existe une dépendance entre deux composants vues. Ce nombre représente l'indice du composant vue dépendant avec le composant vue courant. S'il est égal à zéro alors il n'y a pas de dépendance entre le composant vue courant et un autre composant vue.
mutual_exclusion	S'il est différent de zéro alors il existe une exclusion mutuelle entre deux composants vues. Ce nombre représente l'indice du composant vue en exclusion mutuelle avec le composant vue courant. S'il est égal à zéro alors il n'y a pas d'exclusion mutuelle entre le composant vue courant et un autre composant vue.
active	Prend les deux valeurs : 1 : le composant vue est actif, 0 : le composant vue est inactif
type	Prend les deux valeurs : synchrone et asynchrone.
port_in	Représente le type de l'interface requise (un réceptacle ou un puits d'événements)
port_out	Représente le type de l'interface fournie (une facette ou source d'événements)

Table 2.2. *Ensemble des définitions introduits au langage IDL3*

2.4.2.1.2. Projection de la description d'un CLM de VIEW-IDL3 vers IDL3

La projection de la description d'un CLM, de VIEW-IDL3 en IDL3 par le compilateur **View-IDL3ToIDL3** (cf. figure 2.4.), suit des règles bien définies. Toutefois, la règle principale est qu'un type de CLM est transformé en un ensemble de composants représentant le composant base, ses composants vues et les connecteurs de visibilité.

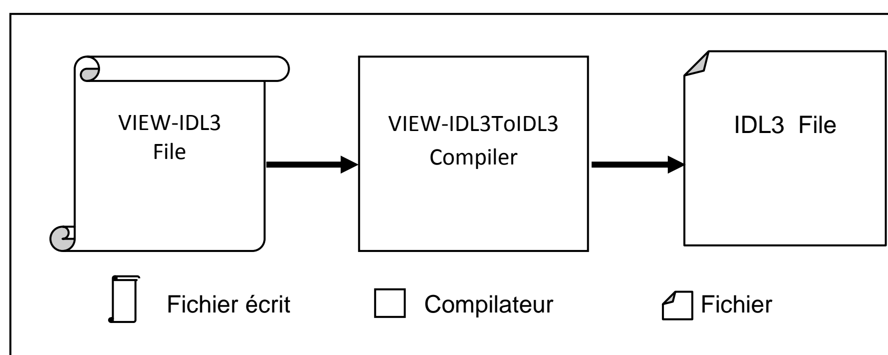


Figure 2.4. *Processus de projection des fichiers VIEW-IDL3 vers IDL3*

La projection de la description d'un CLM de VIEW-IDL3 vers IDL3 produit :

1. La définition d'un composant qui prend le nom : "nom_composant" avec sa maison "home" qui possède le nom "nom_composantHome";

2. Pour chaque description d'un lien, il définit un composant qui dispose du nom du composant suffixé par *Connector.* Le même suffixe sera ajouté à sa maison ;

3. La description des *composant vues* et le *composant base* reste la même en gardant aussi, la déclaration de la maison ;

4. En fonction de la valeur de type : si synchrone, il crée soit une facette pour le ***composant base*** et un réceptacle pour le **composant vue**. Si asynchrone, il crée une source d'événements pour le ***composant base*** et un puits d'événement pour le ***composant vue*** ;

5. La déclaration des attributs reste telle qu'elle est ;

6. Les ports d'entrée/sortie gardent le même nombre mais change la façon de déclaration.

Pour illustrer le résultat de la projection de VIEW-IDL3 en IDL3, nous présentons la projection de l'exemple donné dans la section précédente (exemple d'un CLM Médiathèque) dans le tableau suivant (Tableau 2.3) :

```
module demoM {
//Définition des interfaces
interface LoansInterface { liste opérations….. ; };
interface Counts_AdherentsInterface { liste opérations….. ; };
interface ExamplariesInterface { liste opérations….. ; };
==============================================================================
//La description du composant composant_base MediathèqueComposant_Base
component MediathèqueComposant_Base {
attribute string the_name;
provides LoansInterface ;
provides ExamplariesInterface ;
provides Counts_AdherentsInterface ;
uses LoansInterface To_LoansViews;
uses ExamplariesInterface To_ExamplariesViews ;
uses Counts_AdherentsInterface To_Counts_AdherentsViews ;
//La déclaration de la maison du composant composant_base MediathèqueComposant_Base
home MediathèqueComposant_BaseHome manages MediathèqueComposant_Base { };
==============================================================================
// La déclaration du composant Loans
component Loans {
attribute string the_name;
provides LoansInterface Loans_ViewInterface;
};
//déclaration de la maison "home" du composant Loans
home Loans_ViewHome manages Loans { };
```

```
================================================================================
// La description du connecteur Loans_ViewConnector
component Loans_ViewConnector {
attribute string the_name;
//La déclaration des variables de la relation du connecteur Loans_ViewConnector
attribute boolean active;
attribute int index;
attribute int dependency;
attribute mutual_exclusion;
//La déclaration des interfaces du connecteur Loans_ViewConnector
provides LoansInterface for_MediaLoansCon ;
uses LoansInterface to_LoansCon ;
};
//La déclaration de la maison du connecteur Loans_ViewConnector
home Loans_ViewConnectorHome manages Loans_ViewConnector { };
================================================================================
// La déclaration du composant Examplaries
component Examplaries {
attribute string the_name;
provides ExamplariesInterface Examplaries_ViewInterface;
};
//déclaration de la maison "home" du composant Examplaries
home Examplaries_ViewHome manages Examplaries { };
================================================================================
// La description du connecteur Examplaries_ViewConnector
component Examplaries_ViewConnector {
attribute string the_name;
//La déclaration des variables de la relation du connecteur Examplaries_ViewConnector
attribute boolean active;
attribute int index;
attribute int dependency;
attribute int mutuel_exclusion;
//La déclaration des interfaces du connecteur Examplaries_ViewConnector
provides ExamplariesInterface for_MediaExamplariesCon ;
uses ExamplariesInterface to_ExamplariesCon ;
};
//La déclaration de la maison du connecteur Examplaries_ViewConnector
home Examplaries_ViewConnectorHome manages Examplaries_ViewConnector { };
================================================================================
// La déclaration du composant Counts_Adherents
component Counts_Adherents {
attribute string the_name;
provides Counts_AdherentsInterface Counts_Adherents_ViewInterface;
};
//déclaration de la maison "home" du composant Counts_Adherents
home Counts_Adherents_ViewHome manages Counts_Adherents { };
================================================================================
// La description du connecteur Counts_Adherents_ViewConnector
component Counts_Adherents_ViewConnector {
attribute string the_name;
//La déclaration des variables de la relation du connecteur Counts_Adherents_ViewConnector
```

```
attribute boolean activie;
attribute int index;
attribute int dependency;
attribute int mutual_exclusion;
//La déclaration des interfaces du connecteur Counts_Adherents_ViewConnector
provides Counts_AdherentsInterface for_MediaCountsCon ;
uses Counts_AdherentsInterface to_CountsCon;
};
//La déclaration de la maison du connecteur Counts_Adherents_ViewConnector
home Counts_Adherents_ViewConnectorHome manages Counts_Adherents_ViewConnector { };
```

Tabeau 2.3. *Un extrait de projection de la description de la Médiathèque en IDL3*

2.4.2.1.3. L'interconnexion des composants

Après avoir implémenté le code métier du CLM, son composant base, ses composants vues et ses connecteurs de visibilité, nous avons procédé à l'interconnexion de ces composants. Cette opération doit être automatisée dans notre modèle. En effet, un fichier JAVA contenant l'ensemble de connexions entre connecteurs et composants est automatiquement généré. Un outil dédié à cette tâche est en cours implémentation.

2.5. Conclusion

Dans ce chapitre, nous avons présenté un modèle à base de composants logiciels multi-vues. Ce modèle de CLM combine les avantages de l'approche par point de vue et les avantages du développement de logiciels à base de composants.

Le modèle à base de CLM permet la construction d'un composant logiciel par assemblage d'un composant base et des composants vues. Le composant vue est lié avec le composant base par un nouveau type de connecteur appelé connecteur de visibilité.

Le modèle proposé est un modèle indépendant de toute plate-forme ce qui conforme à la démarche de développement dirigé par les modèles MDA.

Nous avons également présenté la mise en œuvre de la conception d'un composant logiciel multi-vues en CCM ce qui a permis de valider le modèle proposé et les étapes à suivre pour concevoir un composant logiciel multi-vues, en commençant de la description jusqu'au lancement d'exécution sur la plateforme OpenCCM. Ainsi, il est à noter que l'interconnexion entre composant base avec les composants vues, selon notre modèle, se fait automatiquement. La description du type du CLM a nécessité l'introduction de nouveaux concepts qui sont ajoutés à ceux définis par le langage IDL3.

Chapitre 3

Une approche à base de patron pour l'assemblage de Composants logiciels

3.1. Introduction

Dans l'ingénierie des logiciels basée composant le développement d'un logiciel se réduit à un assemblage de composants prédéfinis. Chaque composant joue un rôle spécifique dans le système. Il définit clairement les services qu'il offre et les services requis pour accomplir sa fonction. Le développement de logiciel basé composant augmente considérablement la fiabilité des systèmes puisque leur construction est basée sur des composants testés et certifiés. De plus, les coûts du développement sont généralement réduits car une partie des composants sont simplement réutilisés. L'étape de maintenance se réduit, quant à elle, à un remplacement de composants, ce qui facilite sa tâche et favorise l'évolution du système.

Habituellement, en utilisant un modèle de composants logiciels, un assemblage de composants reflète le raisonnement qui décrit la combinaison et la communication des composants logiciels. Il en résulte que l'évolution globale d'un système à base de composants logiciels est basée sur l'implémentation et l'évolution de ses composants. Un utilisateur d'un tel système est incité à coordonner par lui-même l'utilisation dans le temps de ces composants afin de gérer la structure dynamique de son application (création/suppression, connexion/déconnexion de composants).

Pour représenter une politique d'assemblage, nous avons introduit le concept de patron. Le rôle de ce patron est de représenter la combinaison et le mécanisme de communication, entre les composants logiciels. Son objectif est de capitaliser le savoir-faire afin de proposer une solution générique et correcte pour faciliter et automatiser l'assemblage de composants. Ce patron décrit alors comment les

composants logiciels sont impliqués et doivent être composés dans la construction d'un système logiciel à base de composants.

Dans ce chapitre, nous présentons les patrons dans l'ingénierie des systèmes en donnant des exemples des patrons les plus utilisés ainsi que la manière de représenter un patron. Nous passons après à l'élaboration du patron d'assemblage des composants logiciels orienté point de vue. Et, nous concluons notre chapitre par la présentation du profile UML associé au développement à base de composants multi-vues pour valider notre travail.

2.2. Patrons d'ingénierie

Cette section introduit le concept de patron d'ingénierie et le définit. Elle discute aussi l'utilité des patrons dans l'ingénierie des systèmes d'information et enfin, elle propose deux exemples de patrons. Le premier est le patron Rôle pour la phase d'analyse et le deuxième est le patron Etat pour la phase de conception. Nous concluons cette section par proposer le patron pour la génération de code multiple dans le cadre de la programmation orientée point de vue. Il s'agit du patron sur lequel notre proposition d'assemblage de composants logiciels a été bâtie.

2.2.1. Historique et Définition

La notion de patron a été généralisée par C. Alexander, dans le domaine de l'architecture des bâtiments [Alexander et al. 1977][Alexander 1979]. C. Alexander constatait que les grands bâtisseurs, au fil des siècles, n'avaient pas suivi un modèle préétabli avec des règles rigoureuses, mais que les architectures avaient été adaptées les unes après les autres et à leurs environnements. Il a remarqué avec ses collègues que certaines solutions de conception, considérées comme efficaces et bien acceptées, étaient récurrentes et s'appliquaient dans différentes situations de construction. Cette idée est concrétisée au travers de 253 patrons propres aux problèmes récurrents en architecture décrits d'une manière uniforme pour des raisons de commodité, de clarté et de diffusion [Alexander et al. 1977].

Dans le domaine de l'informatique, les premiers patrons ont été présentés par K. Beck et W. Cunninghan [Beck et al. 1987] lors de la conférence OOPSLA de 1987. Il s'agissait d'une première adaptation du langage de patrons d'Alexander à la conception et à la programmation Objet. Un ensemble de patrons plus spécifiquement dédiés à la conception par objets a été élaboré, un peu plus tard, par E. Gamma [Gamma 1991] en 1991 lors des travaux de sa thèse. Ces travaux ont été suivis en 1995 d'un premier ouvrage collectif [Gamma et al. 1995]. À la même

période, dans le cadre de l'ingénierie de systèmes d'information P. Coad [Coad 92] proposait de faciliter l'analyse d'un système en identifiant les besoins selon 7 patrons pré-établis. Depuis, de nombreux ouvrages et articles ont été publiés sur les patrons [Gamma et al. 1995], [Coad 1995], [Coplien 1996], [Buschmann 1996], [Coad et al. 1997], [Booch 1997], [Fowler 1997], [Ambler 1998], [Larman 02], etc. La première conférence internationale dédiée à la notion de patrons de conception date de 1994 (PLoP : Pattern Languages of Programming) et la première conférence européenne s'est déroulée en 1996 (EuroPLoP).

2.2.2. Patrons de différents niveaux d'ingénierie

Les patrons sont le plus souvent classifiés en fonction de la phase d'ingénierie à laquelle ils s'adressent. Nous distinguons, par exemple, les patrons d'analyse, de conception, etc. Les méta patterns et les patrons d'architecture sont d'autres types de patrons qui s'adressent généralement à la phase de conception.

– Les patrons dédiés à la phase de recensement de besoins permettent une description efficace des besoins des utilisateurs, sous la forme de comportements génériques attendus du système à développer. Il s'agit en général de comportements fonctionnels. Dans certains cas, les comportements non fonctionnels, tels que la performance ou la fiabilité, peuvent cependant être considérés. Les patrons dédiés aux systèmes de télécommunication tolérants aux pannes [Adams et al. 1995] traitent, par exemple, quelques besoins non fonctionnels tels que la fiabilité [Rising 2000]. Usuellement, dans la littérature, les patrons de recensement de besoins font partie des patrons d'analyse.

- Les patrons d'analyse traitent des problèmes qui apparaissent lors de l'analyse des besoins des systèmes. Ils aident le développeur dans la construction de modèles afin de représenter au mieux les besoins de son système d'information. Un patron de gestion de ressources permet, par exemple, de répondre à un problème de ressources humaines qui comprend l'affectation de personnes à des activités ou encore, à un problème de gestion de ressources matérielles, pour réaliser l'affectation de machines dans un contexte de production. Le patron Rôle (cf. tableau 3.1) proposé par P.Coad [Coad 1992] est un exemple de patron d'analyse.

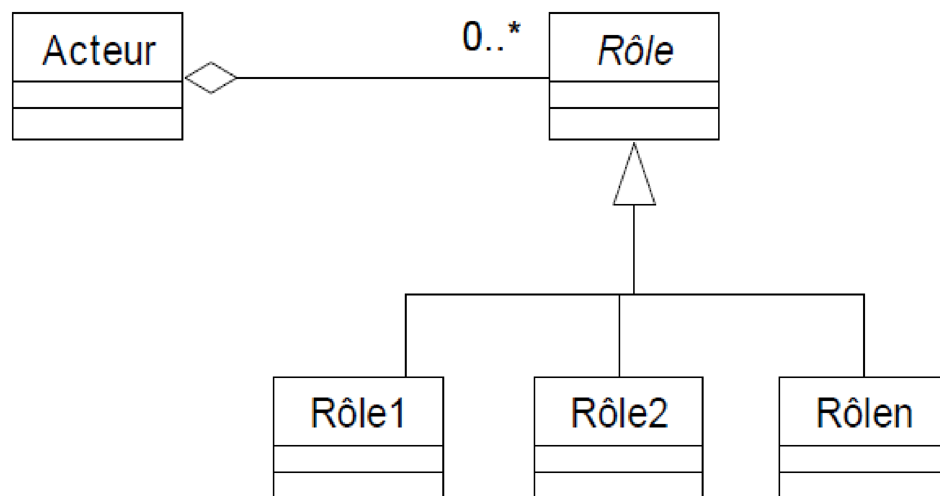
Patron Rôle
Intention. Le patron Rôle donne la possibilité à un objet d'adhérer et/ou de perdre plusieurs rôles.
Motivation. Tout objet peut avoir des propriétés variables selon le rôle joué. Un livre de bibliothèque a, par exemple, des propriétés intrinsèques telles que le numéro d'exemplaire, le N° ISBN, le nombre de pages, etc. Il pourra jouer plusieurs rôles fortement dépendants des processus métiers de la bibliothèque : l'emprunt, la réservation et la maintenance.
Solution. La classe Acteur est associée à une classe Rôle abstraite. Chaque acteur est lie à des instances des classes Rôles spécifiques concrètes (Rôle1, Rôle2, etc.), chacune d'elles représentant un de ses rôles. Cette approche a l'avantage d'être plus concise et flexible, que l'utilisation de l'héritage multiple. La figure ci-dessous montre le diagramme de classes associe à ce patron. 

Tableau 3.1. *Exemple d'un patron d'analyse générale (patron Rôle)*

- Les patrons de conception (design patterns) sont certainement les patrons les plus connus et sans doute les plus utilisés parmi les patrons d'ingénierie. Ils identifient, nomment et abstraient les connaissances convenues de plusieurs expériences, qui répondent à des problèmes de conception communs (qu'il s'agisse de conception détaillée, globale ou liée à des architectures particulières). Le patron Etat (cf. tableau 3.2) du GoF [Gamma et al. 1995] est un exemple de patron de conception.

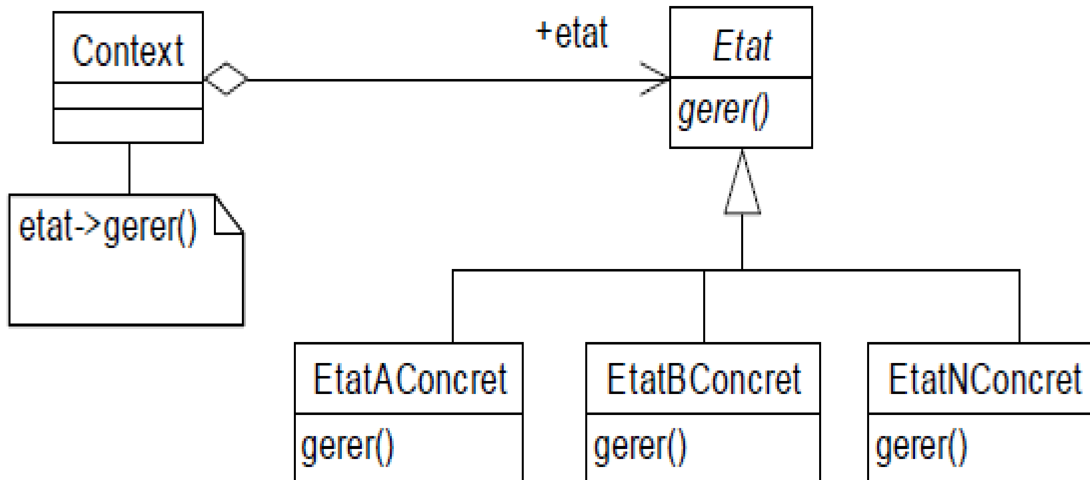
Patron Etat
Intention. Permet à un objet de modifier son comportement quand son état interne change. Tout se passe comme si l'objet changeait de classe.
Structure.  **Constituants.** La classe Contexte définit l'interface intéressant les clients (les états sont transparents pour l'utilisateur). Chaque contexte est lie à une instance qui définit son état courant. La classe Etat spécifie l'interface commune de toutes ses sous-classes ; il s'agit d'une classe abstraite. Les sous-classes concrètes de la classe Etat implantent le comportement spécifique associe à un état du contexte.

Tableau 3.2. *Exemple d'un patron de conception générale (patron Etat)*

– Les patrons d'architecture sont des patrons de conception qui déterminent la structure de base d'une application. Ils offrent un ensemble de sous-systèmes prédéfinis, spécifient leurs responsabilités, et les collaborations et communication entre ceux-ci. Ils incluent, de plus, les règles et les conseils pour établir leurs associations [Buschmann et al. 1996]. Tout patron d'architecture décrit en effet les règles d'organisation et de fonctionnement, les stratégies de haut niveau, les propriétés et les mécanismes globaux d'une application, permettant ainsi de créer des architectures extensibles et interopérables. Le patron MVC2 (cf. tableau 3.3) permet par exemple de dissocier le modèle de l'interface [Turner et al. 2003].

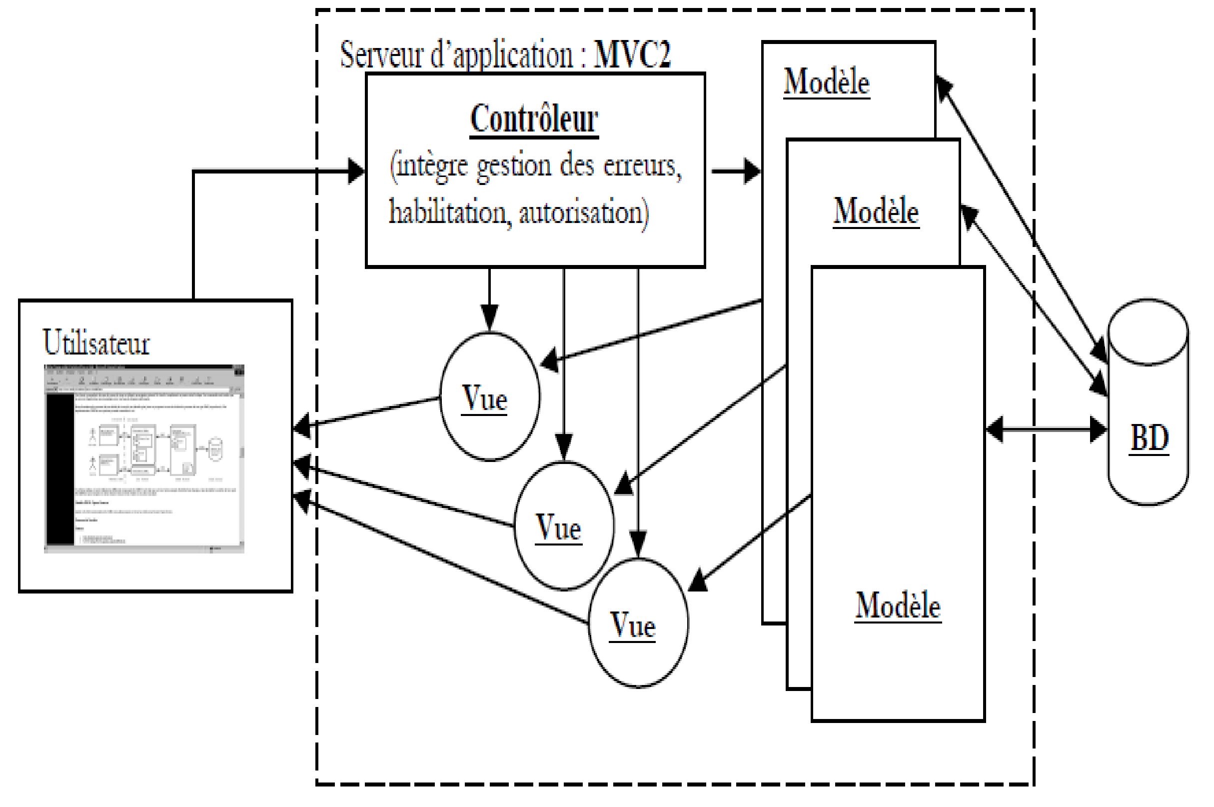
Modèle-Vue-Contrôleur 2
Intention. Assure l'indépendance du modèle par rapport à l'interface. Les objets sont séparés en trois composants : le modèle, la vue et le contrôleur. Le modèle contient la logique et l'état de l'application. La vue représente l'interface utilisateur. le contrôleur réagit aux requêtes de l'utilisateur en effectuant les actions nécessaires sur le modèle.
Solution. 

Tableau 3.3. *Exemple d'un patron d'architecture (MVC 2)*

– Les patrons d'implémentation (ou idiomes) sont des patrons de bas niveau qui traitent des problèmes d'implémentation. Ils peuvent décrire, par exemple, comment implémenter certains mécanismes absents dans des langages (simulation de l'héritage multiple en Java, par exemple). Ils sont généralement spécifiques à une technologie, voire, à un langage de programmation donné.

2.2.3. Patron pour la génération de code multi-cibles

Pour rendre l'approche orientée point de vue réellement utilisable, nous avons décidé d'une part de cibler les langages à objet du marché, et d'autre part de proposer une traduction automatisée du graphe de visibilité (chapitre 2) en un graphe d'implémentation multi-cibles (Java, C++, Eiffel, …). Nous avons ainsi défini un

patron d'implémentation générique inspiré des patrons *Rôle* et *Etat* [Coad 1992][Gamma et al. 1995]. C'est, en fait, un patron qui permet d'implémenter la relation de visibilité proposer par le langage VBOOL. Dans la suite, l'exemple est donné en Java mais la technique proposée est valable pour tous les langages objets supportant le polymorphisme.

La classe multi-vues CMV est traduite par l'ensemble de classes suivant : une classe du même nom reliée par composition à une classe Base_CMV (regroupant les attributs et les méthodes de la base), et à une classe View_CMV, et une liste correspondante aux vues. Les vues sont décrites par des sous classes de View_CMV. La classe CMV fournit la méthode permettant d'activer/désactiver une vue et la méthode qui retourne la vue active. La vue active est soit une instance de la classe View_CMV (vue par défaut quand aucune vue n'est active), soit une instance de la classe dérivant (Viewn_CMV) (cf. figure 3.1).

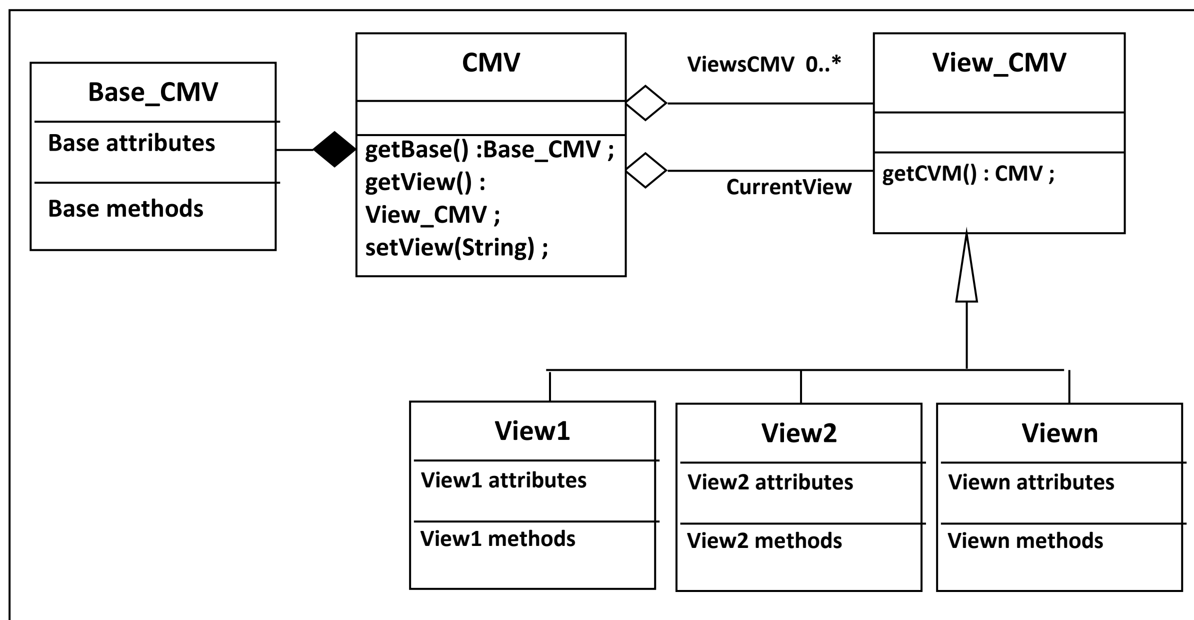


Figure 3.1. *: Patron d'implémentation pour la génération de code multi-cibles*

En appliquant ce patron à notre exemple de la médiathèque, nous obtenons le graphe d'implémentation multi-cibles suivant (cf. figure 3.2). Le modèle généré comprend la classe *Base_Media* contenant les attributs et les méthodes de base et la classe *Mediatheque* reliée via la relation de composition à la classe *Base_Media et la View_Media*. La classe *View_Media* possède un ensemble de sous classes – vues - descendantes (relation d'héritage) contenant les attributs et les méthodes de la vue par défaut.

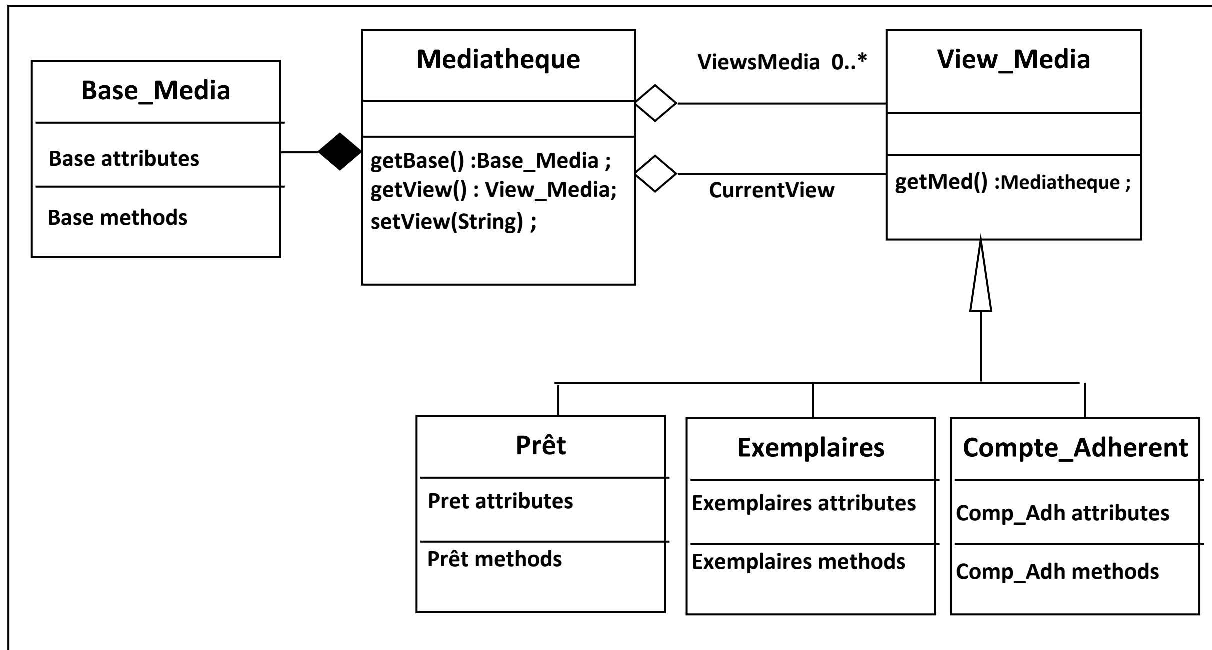


Figure 3.2. *: Graphe d'implémentation multi-cibles*

3.3. Relation de Composition comme support à la composition logicielle

Toute composition logiciel est considérée sous la forme de relation **Tout-Partie** : un composant **Tout** est composé de composants **Partie**. Ces derniers jouant le rôle de fournisseur de services pour ce composant **Tout**. Généralement, un composant logiciel s'appuie sur les services fournis par un ou plusieurs sous-composants pour proposer à son tour des services plus complexes. Cette vision est également celle adoptée par le récent workshop WCOP 2003 [Bosch et al. 2003]. Cette approche présente des similitudes avec le patron de conception Composite défini dans [Gamma et al. 1995].

En ce sens, lorsque le composant **Partie** est non partageable, il ne peut fournir ses services qu'au composant **Tout**. De même, le fait que ce sous-composant ne soit pas partageable implique que, si le composant **Tout** est lui-même le composant **Partie** d'un troisième composant du plus haut niveau, ce dernier ne pourra pas utiliser directement les services du composant **Partie** de plus bas niveau.

Pour déterminer quel type de combinaison de composants doit être utilisé pour réaliser un tel assemblage, UML 2.0 distingue deux types de relation de composition.

3.3.1. Composition horizontale

La composition horizontale représente l'échange des services entre composants de même granularité. Il s'agit de la relation la plus couramment utilisée pour assembler des composants. Pour ce faire, UML 2.0 propose le connecteur d’assemblage, connecteur entre deux composants, qui définit qu’un composant fournit le service qu’un autre composant requiert. Un connecteur d’assemblage doit être uniquement défini à partir d’une interface requise ou d’un port vers une interface fournie ou un port (cf. figure 3.5 et 3.6).

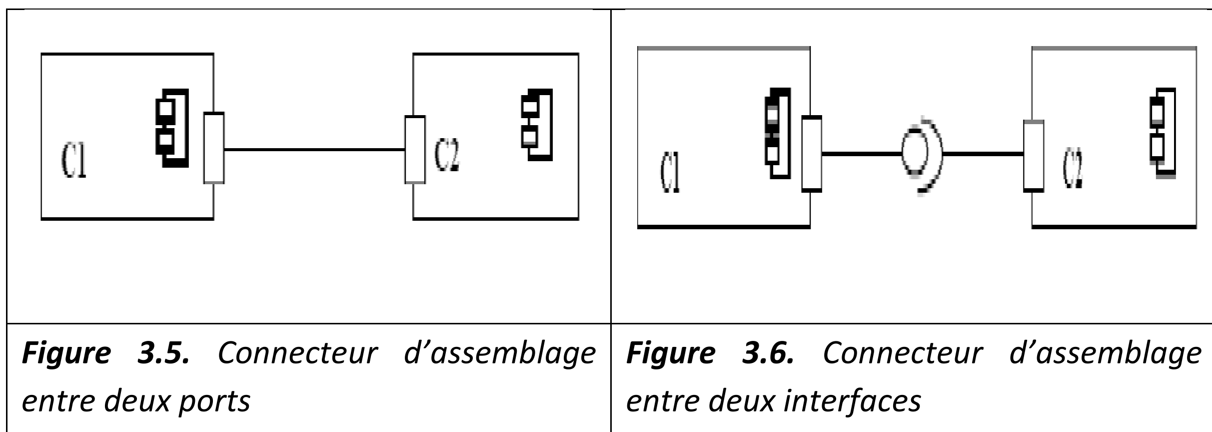


Figure 3.5. *Connecteur d’assemblage entre deux ports*	***Figure 3.6.*** *Connecteur d’assemblage entre deux interfaces*

3.3.2. Composition verticale

Cette composition concerne la réalisation d'un assemblage entre un composant de granularité faible (sous-composant) et un composant de granularité plus forte (composé). On parle alors d'un composant composé de sous-composants, ou de composition hiérarchique. UML 2.0 propose le connecteur de délégation qui relie le contrat externe d’un composant (spécifié par ses ports) à la réalisation de ce comportement par les parties internes du composant. Il permet de lier un port du composant composite vers un port d’un composant situé à l’intérieur du composant composite (relier par exemple un port requis à un autre port requis (cf. Figure 3.7)). Un connecteur de délégation doit être uniquement défini entre les interfaces utilisées ou des ports de même type, c’est-à-dire entre deux ports ou interfaces fournis par le composant ou entre deux ports ou interfaces requis par le composant.

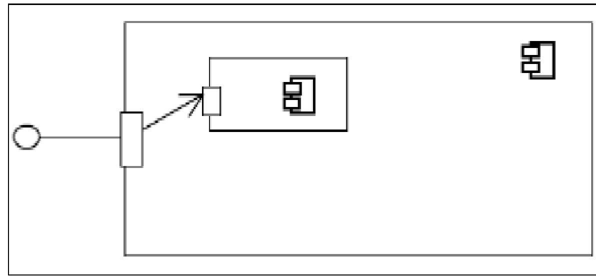


Figure 3.7 : *Connecteur de délégation entre un port externe et un port interne*

3.4. Patron d'assemblage pour les composants logiciels

Notre objectif dans cette section est de proposer un moyen permettant de réaliser efficacement l'assemblage des composants logiciels [Rehioui et al. 2014b] [Rehioui et al. 2015a]. Pour ce faire, nous essayons d'imiter et d'adapter le patron orienté point de vue, pour la génération du code multi-cibles, présenté dans la sous-section 2.2.3.

3.4.1. Motivation

L'idée d'utiliser le patron de génération du code multi-cibles (patron GCM), introduit dans le cadre de l'approche de la programmation orientée point de vue, relève principalement de la constatation suivante : le patron GCM consiste, le plus souvent, à établir une collaboration sur un ensemble de classes données sur lesquelles s'applique le patron. Les classes correspondent à des utilisations propres de chaque acteur au travers le système. Ces classes ne sont rien d'autres que les différentes vues du système. Les vues représentent encore les différentes fonctionnalités et services que doivent assurer le système. Dans le contexte de développement à base de composants ces services et fonctionnalités sont représentés et assurés par les composants logiciels qui constituent un tel système.

Afin d'assurer une meilleure utilisation et adaptation du patron GCM dans le contexte de développement à base de composants, nous devons penser à traduire les liaisons existantes entre les classes en des liaisons entre les composants logiciels.

3.4.2. Adaptation des relations de liaisons

La mise en œuvre du patron GCM dans le contexte de développement à base de composants logiciels nécessite des techniques d'implémentation appropriées. Les relations de liaisons entre les classes du paradigme objet, qui se base principalement sur le mécanisme d'héritage, sur la composition et sur la délégation, doivent être adaptées. Dans le cadre CBSE, toute liaison entre composant logiciel est considérée sous la forme de relation de composition (Tout-Partie). Pour remédier à ce problème de limite, il est possible, d'adapter la relation d'héritage entre deux classes

(paradigme objet) par la relation de composition verticale des composants logiciels. Cette adaptation est justifiée par le fait que la classe mère est une généralisation de la classe fille ; c'est à dire que la classe mère est la classe de granularité plus forte et donc c'est le **composé et** que la classe fille est la classe de granularité faible et donc c'est le **sous- composant** (cf. figure 3.8). Pour adapter la relation d'association entre deux classes par une relation entre les composants logiciels, la composition horizontale est la relation la plus appropriée. Cette adaptation est encore justifiée par le fait que les deux classes ont la même granularité et que par association, il y a possibilité qu'une classe délègue ou fait appel à une méthode de l'autre classe ; et ceci se traduit par l'échange des services entre les deux classes (cf. tableau 3.4).

Paradigme Objet	**Développement à base de composants**
Classe	Composant logiciel
Relation d'héritage	composition horizontale
Relation d'association	composition verticale

Tableau 3.4. Tableau d'adaptation

3.4.3. Représentation du patron pour les composants logiciels multi-vues

3.4.3.1. Canevas de présentation d'un patron

Un patron est avant tout une idée et comme toute idée n'a de valeur que si elle peut être communiquée. Il est donc très important d'avoir un format plus ou moins standard pour pouvoir décrire un patron. Il y a, en pratique, de nombreuses variations stylistiques à ce format. Les plus utilisés sont sans doute le format d'Alexander et celui du GoF. Ce dernier inverse en quelque sorte l'ordre de la présentation en commençant par la forme de la conception puis en décrivant le problème, le contexte et les exemples auxquels elle s'applique. Le canevas de présentation est donc structuré comme suit (cf. tableau 3.5) :

Nom du patron : le nom du patron est son identifiant. Il doit être choisi de manière à véhiculer succinctement l'essence du patron. Un nom bien choisi est absolument vital, car il devient ensuite un élément clé du vocabulaire de conception. Exemple : *Etat*.

Intention : un court texte ayant pour objectif de répondre à la question suivante : que cherche à faire le patron ? Quelle est l'intention sous-jacente ? À quel problème particulier s'attaque-t-on?

Exemple : *Définir une dépendance de* ***un objet vers plusieurs*** *entre les objets de façon à ce que, quand un objet change d'état, tous ses dépendants sont informés et mis à jour automatiquement.*

Structure : une description structurelle (en termes de classes et de leurs relations exprimées par exemple en UML) d'un exemple du type de solution proposée par ce patron. Exemple : A cet égard, il faut insister sur le fait que le schéma UML décrivant la structure du patron doit, au côté des autres rubriques, contribuer à véhiculer l'idée constituée par ce patron.

Utilisations connues : Exemples documentés d'application de ce patron dans des systèmes réels. L'effort de documentation d'une solution de conception n'est en effet valable que si cette solution peut être appliquée de manière récurrente pour résoudre le problème dans un contexte donné. Exemple : C'est bien sûr le cas du patron observateur qui se retrouve d'une manière ou d'une autre au cœur de toutes les interfaces graphiques conçues depuis plus de vingt-cinq ans.

Nom du patron	Identifiant du patron
Intention	Problème qu'on souhaite à résoudre
Structure	Solution proposée
Utilisations connues	Exemples d'application de ce patron

Tableau 3.5. *Canevas de présentation d'un patron*

3.4.3.2. Patron pour les composants logiciels multi-vues

Afin d'assister le développeur à réaliser un assemblage multi-vues de composants logiciels, nous avons décidé d'automatiser cette opération. Cette automatisation est réalisée sous la forme d'un patron permettant directement de proposer un modèle de liaison entre les différentes vues du système représentées par des composants logiciels (composants logiciels vues).

Pour des raisons purement liées à la simplicité et à la lisibilité de la représentation du patron, nous allons commencer par la définition de quelques composants logiciels essentiels qui constituent les éléments de base de ce patron.

Base_Component : Composant logiciel qui propose toutes les interfaces communes entre les différents composants logiciels vues.

Controller_Component : Composant logiciel qui dispose la majorité des contrôles (activer/désactiver, retourner le composant logiciel vue actif, etc.) : c'est un

composant logiciel fédérateur. Il est le seul composant qui a le droit de communiquer avec les autres composants.

Generic_Component : Composant logiciel intermédiaire entre le composant contrôleur (fédérateur) et les composants logiciels vues. Son principal rôle est de faciliter la communication avec tous les composants logiciels vues et d'avoir à un même niveau d'abstraction. Ce composant logiciel permet aussi d'avoir la possibilité d'ajouter ou de retirer un composant logiciel vue du système sans toucher le composant contrôleur.

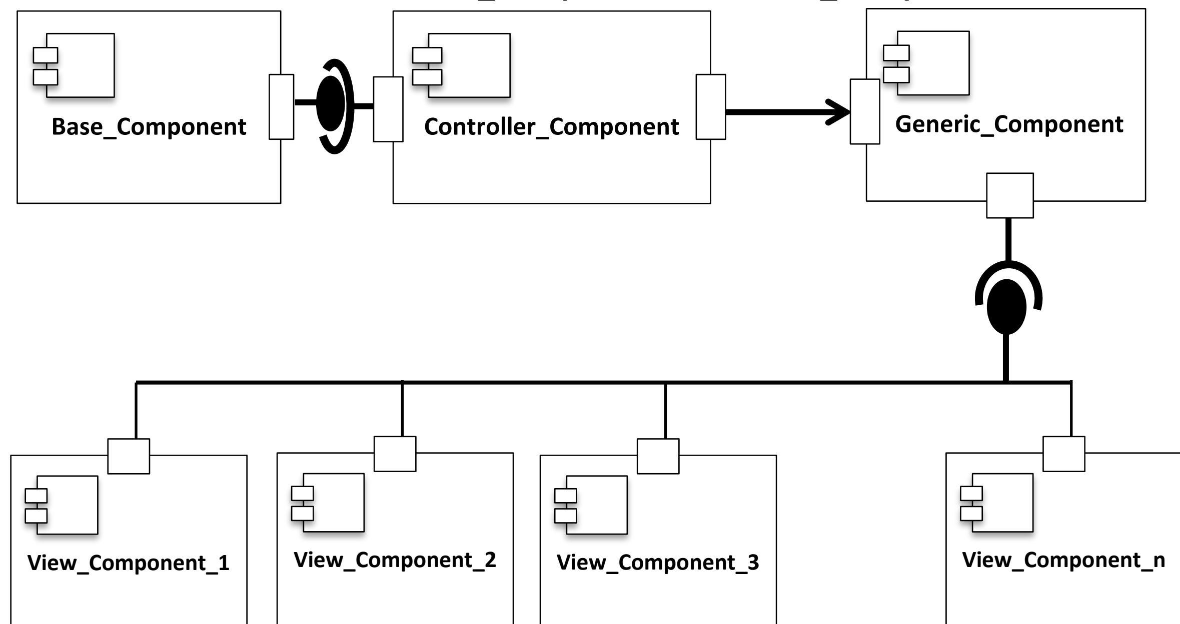
Patron Composant Logiciel multi-Vues (CLV)
Intention. Dans le développement à base de composants, tout système peut avoir plusieurs types d'utilisations représentées par ses composants logiciels vues. Le patron composant multi-vues propose de réaliser un assemblage multi-vues de ses composants logiciels.
Structure. La connexion s'établit entre les différents composants logiciels par la relation de composition. Une composition horizontale (connexion d'assemblage) entre **Controller_Component** et **Base_Component** et une autre de même type entre **Generic_Component** et les différents composants logiciels vues. Une composition verticale existe entre le **Controller_Component** et **Generic_Component.** 
Utilisations Connues. Tout type de développement orienté point de vue soit à classes ou composants logiciels.

Tableau 3.6. Représentation générale du patron CLV

3.5. Vers une représentation complète du patron CLV

3.5.1. Représentation complète du patron CLV

La figure 3.8 exprime une extension du patron CLV sous le standard UML. C'est une version améliorée avec une représentation explicite des hot-Spots (point chaud) [Larman 2002] [Fontoura et al. 2002]. Un hot-Spot est défini comme un aspect qui peut varier selon le type de l'instanciation du patron.

Le hot spot ***incomplete*** est une contrainte fournie par le stantard UML et indique dans notre contexte que plusieurs composants logiciels vues peuvent être ajoutés lors de l'instanciation du patron CLV. Le nombre de ces composants logiciels n'est pas limité.

Le mot clef ***final*** a presque le même sens que celui utilisé dans langage Java. Le mot clef ***final*** ajouté au composant logiciel signifie que ce composant ne peut être considéré comme un composant logiciel vue d'un autre composant logiciel. Cette restriction est imposée tout simplement pour simplifier la gestion d'évolution dynamique du système (activer/désactiver un composant logiciel).

Le stéréotype ***ForAllMethods*** est ajouté dans le composant logiciel **Controller_Component** pour indiquer que tous les services fournis par l'intermédiaire de ses interfaces doivent être redirigés vers le composant logiciel **Base_Component** (composant logiciel regroupant les services communs, entre les différents composants vues, sous forme des interfaces fournies). Si un service est inexistant dans l'une des interfaces de **Base_Component**, la redirection (aiguillage) doit être effectuée vers les composants logiciels vues.

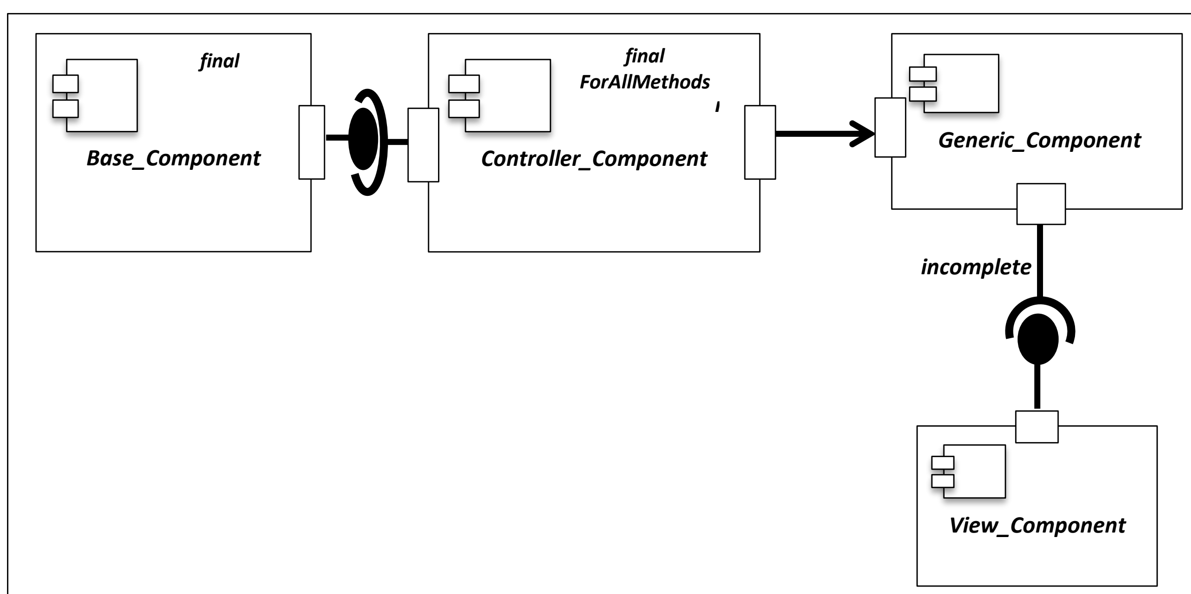


Figure 3.8. *Représentation Complète du patron CLV*

3.5.2. Mise en œuvre du patron CLV amélioré

Un client (dévellոppeur) qui veut utiliser le patron CLV est amené à définir les trois composants logiciels (Base_Component, Controller_Component, Generic_Component) et un ensemble de composants logiciels vues de son système (View_Component_1, View_Component_2, ..., View_Component_n). La compréhension de l'opération d'instanciation du patron CLV est très essentielle pour pouvoir l'utiliser correctement. Cependant, le tableau 3.5 ne donne aucune information sur cette opération.

A notre connaissance, aucune méthode ou langage de modélisation, à base de composants logiciels proposent un diagramme similaire, au niveau sémantique, à celui qu'on a proposé précédemment (cf. figure 3.8). Pour cette raison, nous avons décidé d'accompagner notre patron amélioré CLV, lors de son instanciation, par le diagramme d'activité du modèle UML. Le diagramme d'activité va permettre de définir l'ensemble des actions à suivre pour instancier le patron proposé (cf. figure 3.9).

L'instanciation du patron CLV est effectuée en utilisant la syntaxe définie par le diagramme d'activité UML, où chaque action définit une transformation sur le patron. La syntaxe de la transformation est sous-jacente à PROLOG-like et sa sémantique est assez intuitive. Parmi les transformations effectuées on trouve [Rehioui et al 2014a]:

- **Define (Base_Component)** : demande à l'utilisateur de définir le composant **Base_Component** ;

- **Define (Controller_Component)** : demande à l'utilisateur de définir le composant **Controller_Component** ;

- **Define (Generic_Component)** : demande à l'utilisateur de définir le composant Generic_Component ;

- **SubComponent(Generic_Component, View_Component) : View_Component** est le composant logiciel vue qui doit être ajouté comme sous composant de **Generic_Component**;

- **Composite-Horizontal (Controller_Component, Base_Component)** : demande à l'utilisateur d'ajouter une liaison de type composition horizontale entre les deux composants **Controller_Component** et **Base_Component** ;

- **Composite-Horizontal (Controller_Component, Generic_Component) :** demande à l'utilisateur d'ajouter une liaison de type composition verticale entre les deux composants **Controller_Component** et **Generic_Component.**

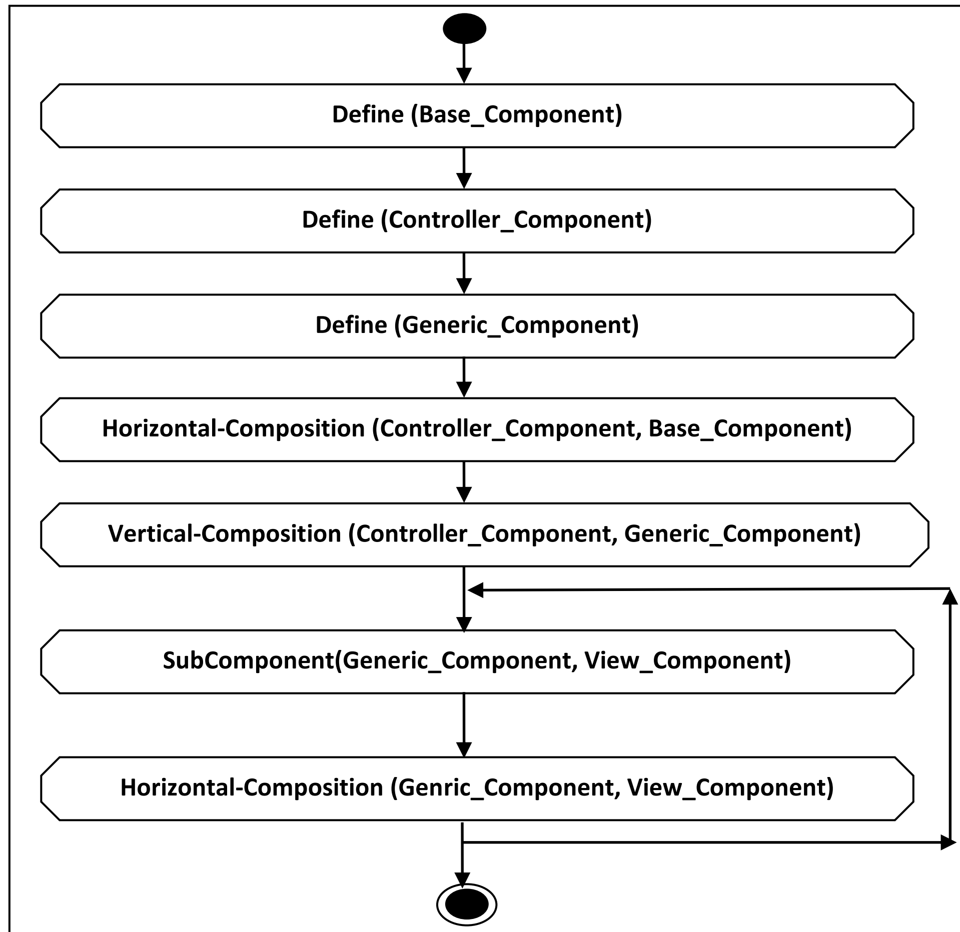


Figure 3.9. *Diagramme d'activité d'instanciation du patron amélioré CLV*

La Figure 3.8 est plus expressive en ce qui concerne l'instanciation du patron, mais elle peut être encore complétée par le diagramme d'activité d'instanciation présentée dans la figure 3.9. Les deux figures (cf. figure 3.8 et figure 3.9) se complètent mutuellement et fournissent ensemble une représentation plus expressive pour la mise en œuvre du patron amélioré CLV que celui montré dans le tableau 3.6.

La section suivante généralise notre solution proposée en décrivant la syntaxe et la sémantique des différentes extensions que nous avons ajoutées à la notation UML.

3.6. Profile UML pour les composants logiciels multi-vues

Généralement, pour un représenter un système à base composants, nous avons besoin du diagramme de composants. Un diagramme de composants décrit l'organisation du système du point de vue des éléments logiciels (Composant logiciel) et il permet de mettre en évidence les dépendances entre les composants [UML 2003] [Fernandez et al. 2004].

Pour que le développement à base de composants multi-vues soit convaincante et facile à utiliser, et pour ne pas pénaliser le développement à base de composants logiciels "classique", nous allons proposer une traduction automatique du diagramme de composants multi-vues représentant un système en un ensemble de composants logiciels "traditionnels". Cette traduction sera basée essentiellement sur le patron d'assemblage orienté point de vue développé précédemment [Hair 2016].

3.6.1. Traduction du diagramme de composants multi-vues

Pour illustrer la traduction d'un diagramme de composants multi-vues en un diagramme de composants logiciels "traditionnels", nous allons prendre l'exemple du Système d'Enseignement à Distance (SED) tel qu'il en existe dans les universités. Ce système permet aux des étudiants de suivre des formations à distance. Nous considérons que le système ne cible qu'un seul site, géré par un responsable. Un site propose des formations (cours) auxquelles peuvent s'inscrire des personnes (alors qualifiées d'étudiants), qui acquittent des droits en fonction des formations suivies. Pour simplifier l'illustration du diagramme de composants multi-vues (cf. figure 3.10), nous nous limitons uniquement aux acteurs suivants : Etudiants, Enseignants, Responsable de site et Conseil pédagogique. Chaque acteur a un point de vue spécifique sur le système. Dans notre modèle simplifié, nous spécifions seulement 4 composants logiciels vues associés aux acteurs *Responsable de site*, Conseil pédagogique, *Enseignant* et *Etudiant*. Le composant multivues *Formation* est *vue comme* quatre composants logiciels correspondant aux points de vue de ces acteurs. Le composant *Vue_Enseignant* décrit la spécificité d'une formation selon le point de vue d'un enseignant, Le composant *Vue_Con_Ped* décrit la spécificité d'une formation selon le point de vue d'un conseil pédagogique, et le composant *Vue_Res_Site* décrit la spécificité d'une formation selon le point de vue d'un responsable de site. On peut toujours faire évoluer le système SED en ajoutant un point de vue.

Les entités principales du système sont : Formation, Vue_Etudiant, Vue_Enseignant, Vue_Res_Site, Vue_Res_Ped, Enseignant, Etudiant, Cours Examen, Inscription, Exercice, etc. Pour simplifier l'illustration, nous focalisons notre attention sur l'entité « Formation » et uniquement sur les quatre acteurs, en proposant le diagramme de composants multi-vues illustré sur la figure 3.10.

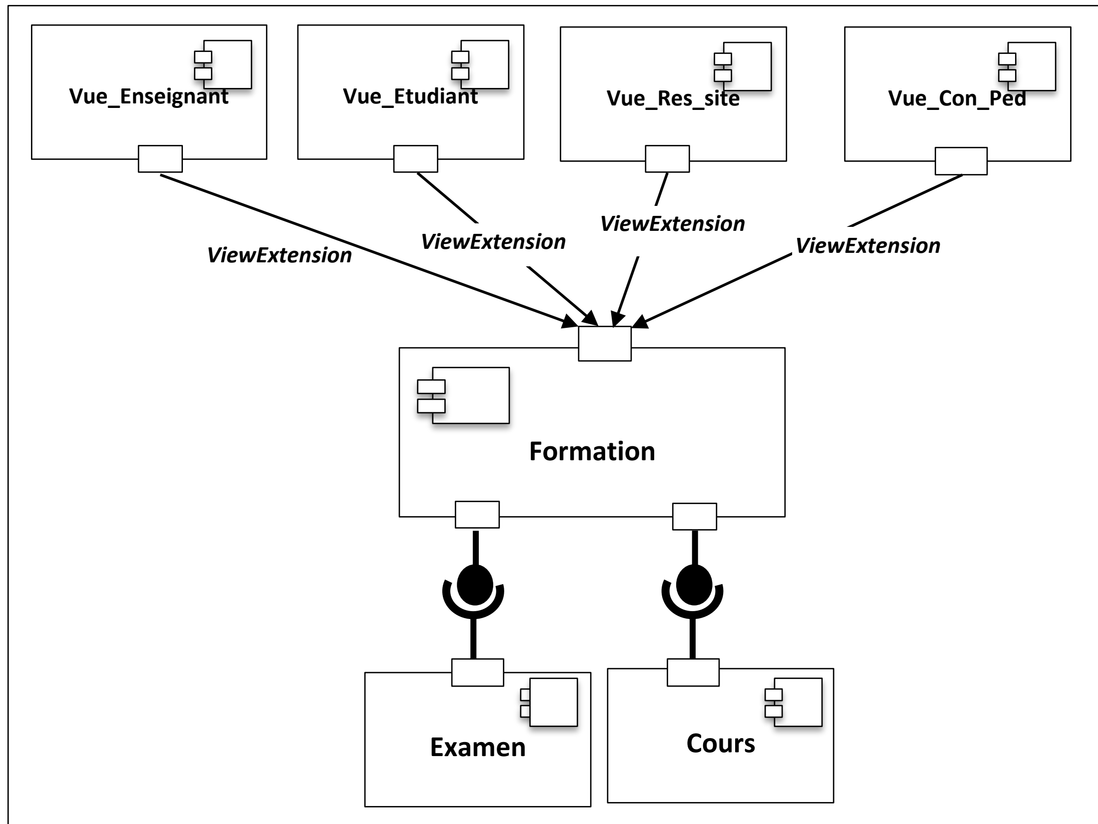


Figure 3.10. *Extrait du Diagramme à base de composant multi-vues Formation*

Dans cet exemple, *Formation* est un composant multi-vues. Les interfaces offertes par ce composant diffèrent selon la vue active. Un Enseignant ne voit pas la formation comme le voit un Etudiant. De ce fait, le composant multi-vues Formation publie différemment les interfaces de ses composants logiciels vues selon la vue active. Le composant Examen est un composant logiciel qui interagit avec le composant multi-vues Formation. Cette interaction est conditionnée par le point de vue actif. Par exemple, le Conseil pédagogique s'intéressera au type et l'origine de l'examen pour faire le bilan pédagogique afin de définir la stratégie du SED, alors qu'un enseignant propose et corrige un examen pour évaluer les étudiants inscrits dans ce module. Par contre, le Responsable de site voit le système de la manière pour ajouter de nouvelles formations, gèrer les inscriptions des étudiants et affecter les enseignants responsables des formations.

L'application du patron amélioré CLV sur le composant multi-vues Formation, nous permet d'obtenir le diagramme de composant logiciel traditionnel du standard UML. Les trois composants composants Base_Formation, Controlleur_Formation et Vue_Generique de la figure 3.11 sont les composants logiciels ajoutés par le patron. Les deux composants Examen et Cours seront liés au Controlleur_Formation.

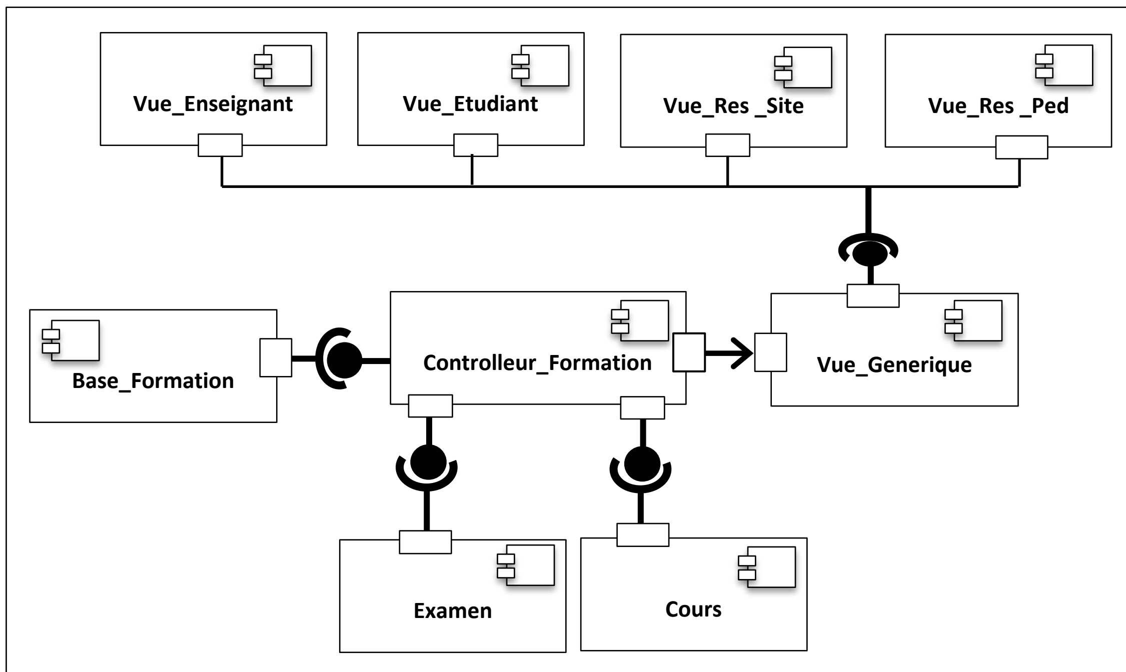


Figure 3.11. *Diagramme traditionnel du diagramme à base de composant multi-vues Formation*

3.6.2. Profile UML

Les notions de profile et de métamodèle sont très proches et une utilisation combinée de celles-ci peut être bénéfique pour un projet [Peltier 2002]. Ceci favorise l'émergence de spécifications contenant à la fois la définition de leur profil et leur métamodèle MOF (Meta Object Facility). Dans cette optique, nous avons défini un métamodèle pour décrire la manière dont les concepts de notre profil s'intègrent dans le métamodèle UML.

Ce métamodèle étend celui d'UML avec de nouveaux éléments de modélisation : *Incomplete, Final, Multi_views_Component, et ViewExtension*. Le métamodèle défini montre qu'un composant multi-vues est composé d'un composant de base, d'un composant contrôleur, d'un composant générique et d'un ensemble de composants vues (cf. figure 3.12). Ces dernières sont reliées au composant multi-vues via une relation de dépendance particulière (ViewExtension). Chaque composant vue est associée à un acteur unique. Les tableaux 3.7 et 3.8 décrivent, pour chaque stéréotype et élément ajouté, le nom, l'élément de modélisation associé et une description informelle.

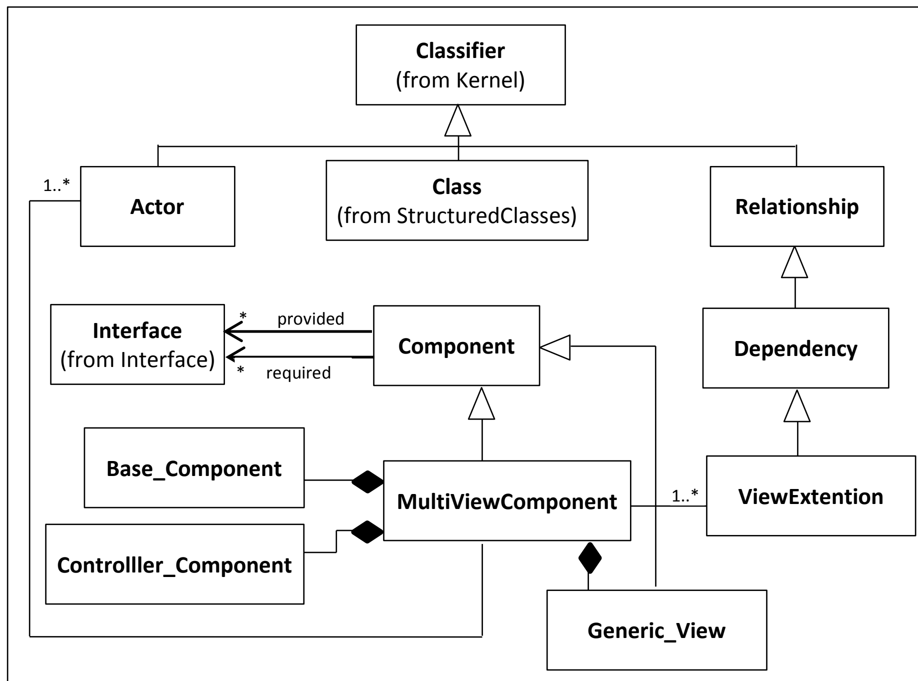


Figure 3.12. *Fragment du métamodèle associé au profil par un développement à base de CMV*

Stéréotype	Elément de modélisation	Sémantique informelle
Final	Composant	Composant ne peut être un composant logiciel d'un autre composant multi-vues.
incomplete	Contrainte	Possibilité d'ajouter des composants logiciels vues.
ViewExtension	Dépendance	Relation entre le composant multi-vues et composant logiciel vue.
MultiViewsComponent	Composant	Composant représentant un composant multi-vues non éclatée.
ForAllMethod	Contrainte	Toutes les méthodes doivent être redirigées

Tableau 3.7. *Stéréotypes introduits dans le développement à base de composants multi-vues*

Elément Ajouté	Elément de modélisation	Sémantique informelle
Base_Component	Composant	Composant décrivant les interfaces communes et partagées entre les composants logiciels vues.
Generic_View	Composant	Composant permettant d'unifier l'accès aux différentes interfaces des composants vues.
Controller_Component	Composant	Composant contenant les interfaces permettant d'activer/désactiver, ajouter/retirer un composant logiciel vue et retourner le composant logiciel actif.

Tableau 3.8. *Eléments ajoutés dans le développement à base de composants multi-vues*

3.7. Conclusion

L'apport du travail présenté dans ce chapitre se situe plus précisément dans l'introduction de la notion de composant multi-vues dans le développement à base de composants logiciels (CBSE). Ceci a été traduit par la proposition du patron d’assemblage CLV sous sa première version et sa deuxième version (patron amélioré). L'ensemble des éléments ajoutés et introduits (stéréotypes, contraintes, composants) pour élaborer des composants multi-vues est regroupé dans le profil UML.

L'élaboration d'une méthodologie d'analyse/conception à base de composants logiciels centrée utilisateur (par points de vue) nous apparait une nécessitée pour bien mener le développement d’un tel système complexe. L’objectif du chapitre suivant est ainsi de proposer une démarche dirigée par les modèles dans le cadre MDA (Model Driven Architecture) et de fournir des moyens pour identifier les composants logiciels du système.

Chapitre 4

Vers une démarche à base de composants logiciels

4.1. Introduction

Les systèmes logiciels modernes sont de plus en plus grands, complexes et mal contrôlés, ce qui a entraîné des coûts de développement élevés, une faible productivité, et une qualité incontrôlable des produits logiciels [Cai et al. 2000]. Par conséquent, il y a une demande croissante pour la recherche d'un nouveau paradigme du développement logiciel, efficace et rentable. Une solution clé à ces problèmes est le développement basé Composant.

Le développement basé composant est une approche de développement logiciel où tous les aspects et phases du cycle de vie y compris l'analyse de besoins, la conception, la réalisation, test, déploiement, et également la gestion de projet, sont basés sur les composants [Herzum et al. 2000]. Le développement basé composant consiste en la construction de logiciel en utilisant une méthodologie basée composant dans la mesure où tout le développement logiciel sera centré sur les composants. Une méthodologie est une manière systématique de faire les choses. C'est un processus répétitif que nous pouvons suivre à partir des premiers pas du développement logiciel jusqu'à la maintenance du système opérationnel [Barbier et al. 2004]. Ceci signifie qu'on observe le système depuis la première notion de son existence jusqu'à son fonctionnement et sa gestion [Anderson et al. 1992]. Plusieurs approches ont été proposées pour le développement du cycle de vie d'un logiciel. Elles sont toutes basées sur les mêmes activités et se diffèrent seulement par la manière dont elles sont réalisées.

Proposer une nouvelle approche va être le sujet de ce chapitre dans lequel nous allons présenter tout d'abord quelques approches de développement logiciel basées composants. Ensuite, nous présentons la démarche sous les différents modèles (cas d'utilisation, interaction, conception et fusion) puis sous ses différentes phases de

développement. Après, nous situons la démarche dans le contexte MDA et nous concluons par le prototype support.

4.2. Ingénierie logicielle basée composant (CBSE)

L'ingénierie logicielle basée composant (Component Based Software Engineering : CBSE) désigne les méthodes, techniques et outils permettant la conception et la mise en œuvre d'applications basées composants. La CBSE est connue comme étant une nouvelle approche puissante qui a amélioré de manière significative si ce n'est pas révolutionner le développement et l'utilisation des logiciels en général. Parmi les principales missions de la CBSE est de fournir un support pour le développement des composants.

4.2.1. Catalysis

Catalysis [D'Souza et al. 1998] est une méthodologie de conception d'applications à base de composants, flexible et capable de s'adapter à tout type de projet d'ingénierie. Catalysis définit à la fois des techniques de modélisation et un processus complet de développement d'applications qui commence par la découverte des concepts métiers et va jusqu'à l'implémentation. Les modèles de Catalysis sont basés sur UML mais s'éloignent tout de même du standard.

En ce qui concerne le processus de développement, les modèles de Catalysis concernent trois niveaux de modélisation :

- Métier ou domaine : il s'agit de spécifier les concepts et règles métier. Pour cela, il faut notamment décrire l'usage que font les utilisateurs du système à construire.
- La spécification des composants : il faut définir les composants qui forment l'application, les services qu'ils offrent et requièrent et leur sémantique. Il s'agit de définir les frontières des éléments qui forment l'application mais sans décrire comment chaque élément est construit.
- L'implémentation des composants : pour chaque composant, il faut définir son fonctionnement interne, les objets qui le composent et comment son implémentation est réalisée. C'est l'intérieur du composant qui est donc traité.

Pour le processus de développement en lui-même, de manière générale, le développement complet d'une application comporte cinq phases, qui recoupent les trois niveaux de modélisation que nous venons de citer :

- Modélisation du métier : description des concepts et règles métiers.
- Spécification des besoins : définition de ce que le logiciel doit faire.
- Conception des composants : définition à haut niveau des composants et de leurs collaborations et interactions.
- Conception des objets : description du fonctionnement interne des composants à un niveau d'implémentation.
- Architecture de jeu de composants : description d'éléments communs à une famille de composants, afin notamment d'aider à la réalisation de composants d'une même ligne de produit.

Pour terminer, Catalysis met en avant trois grands principes lors de l'application de la méthodologie et du processus de développement :

- L'abstraction qui consiste à cacher des éléments, des parties ou des détails de conception qui n'ont pas d'intérêt à un niveau donné afin de se focaliser sur les aspects principaux de ce niveau. L'abstraction est essentielle pour pouvoir bien gérer la complexité.
- La précision : il s'agit d'éviter de se retrouver avec des modèles sémantiquement incomplets ou non-entièrement définis.
- Des éléments connectables : des éléments logiciels de tous niveaux (conception, implémentation, etc.) qui sont conçus pour être assemblés entre eux. Le but est de favoriser la réutilisabilité de ces éléments et de forcer à spécifier précisément les interfaces d'utilisation de ces éléments afin de bien ou de mieux pouvoir les utiliser.

2.4.2. UML Component

La démarche UML Component [Cheesman et al. 2001] spécialise la démarche Catalysis [D'Souza et al. 1998] et définit une méthode "précise" de spécification d'applications à base de composants logiciels de l'étude du métier au déploiement des composants dans des technologies diverses. L'identification des composants est faite à partir de concepts métiers, décrits par la notion de "types". Le composant regroupe un ensemble de types. Ensuite, l'étude des interactions entre composants permet de définir les interfaces des composants. Les interactions entre composants sont définies grâce aux interfaces et donc aux appels de méthodes et services entre composants.

Dans le cadre de l'ingénierie des modèles, UML Component définit 3 types de modèles :

- **Le modèle conceptuel** qui est un modèle indépendant d'une plate-forme technologique. (PIM)

- **Le modèle de spécification** qui définit un modèle spécifique à une plate-forme technologique et qui décrit l'extérieur du composant, les services qu'il offre et qu'il requiert (PSM).

- **Le modèle d'implémentation** qui définit un modèle spécifique à une plate-forme technologique et qui décrit l'intérieur du composant, son implémentation, c'est-à-dire comment il répond aux services décrits dans le modèle de spécification (PSM).

Le processus UML Component n'est pas MDA dans le sens où il ne donne pas une définition abstraite et complète du système (PIM) ni une définition de projection de ce modèle dans une technologie (PSM). Cependant, il permet une bonne séparation des préoccupations et des activités de type PIM et de type PSM. L'avantage de UML Component est de limiter les problématiques d'intégration des préoccupations techniques dans le résultat d'une réflexion neutre.

Le développement à base de composants en UML Component est différent des autres approches dans sa séparation des spécifications de composant de l'implémentation et la division de ses spécifications en interfaces. La connexion entre les composants est alors spécifiée par la notion d'interfaces. Ainsi, la gestion du changement est améliorée. En effet, un composant peut être remplacé par un autre composant ayant la même interface.

Un processus à base de composants doit permettre la définition d'interfaces du composant et séparer les spécifications du composant de l'implémentation. Par contre, l'objectif de UML Component étant surtout d'obtenir les "bonnes" interfaces d'un système. On peut craindre que les livrables produits pour mener à ce résultat soient oubliés lors de la livraison des composants, ce qui nuit à la gestion de l'évolution. Le courtage d'un composant devient classique et se fait à partir de ses interfaces.

UML Component définit des stéréotypes pour spécialiser UML pour la définition de systèmes à base de composants. Les auteurs soulèvent l'inconvénient de ne pas trouver d'outils permettant d'exploiter les contraintes exprimées en OCL sous-jacentes à chaque stéréotype.

4.2.3. Component Unified Process *:* CUP

Le Component Unified Process (Processus Unifié à base de Composants) est une démarche d'ingénierie itérative, basée sur les besoins, et orientée composant,

destinée à modéliser et à développer une application. Il s'inscrit dans l'initiative Model Driven Architecture (MDA) en préconisant la réalisation d'un modèle indépendant avant la conception automatisée par projection vers des modèles spécifiques. Enfin, il offre la séparation des préoccupations, en permettant à plusieurs intervenants d'un projet d'avoir chacun son propre point de vue sur l'architecture, tout en maintenant une certaine cohérence.

C'est l'approche la plus couramment employée aujourd'hui. Afin de répondre correctement à un besoin, et concevoir des cas de tests en adéquation avec l'usage du système qu'en fera l'utilisateur final, une activité prépondérante consiste à définir les cas d'utilisation du système. De cette activité découlent les autres. L'analyste des cas d'utilisation et les utilisateurs finaux font la liste des cas d'utilisation et les décrivent. L'analyste des interactions réalise chaque scénario de cas d'utilisation comportant une séquence d'étapes, en décrivant les interactions entre les éléments de l'architecture du système dans des collaborations. Le concepteur de composant ajoute une interface de composant ou crée un nouveau composant, selon les collaborations définies. Ces interfaces permettent de connecter les composants pour construire le modèle du système par assemblage. Cette dernière activité est réalisée par l'assembleur de composants.

La démarche CUP est orientée composant. Elle facilite l'intégration et la réutilisation de composants. Pour cela, les différents points de vue sur un composant sont décrits dans le formalisme CUP, par des sous-modèles appelés vues. La notion de composant logique est une notion transverse aux différentes vues et ceci afin de maintenir la cohérence globale du système. Intégrer un composant, c'est intégrer chacune de ses vues dans le modèle du système :

- **La vue de cas d'utilisation d'un composant** permet à l'analyste des cas d'utilisation d'exprimer les besoins attendus par les utilisateurs finaux de ce composant. Dans ce modèle, le composant logique est défini par un ensemble de cas d'utilisation qui décrivent ses spécifications fonctionnelles. L'intégration d'un composant à ce niveau enrichit la vue de cas d'utilisation entière du système.

- **La vue d'interactions d'un composant** permet à l'analyste des interactions d'exprimer le comportement des éléments de l'architecture du composant, les uns par rapport aux autres, lors de la réalisation des cas d'utilisation. Ces éléments sont essentiellement des instances des classes constituant les parties internes du composant. Le comportement interne d'un composant est décrit par la vue d'interactions qui s'apparente aux diagrammes de séquences UML. Nous l'avons vu dans un chapitre précédent, un composant logique regroupe un certain nombre de collaborations fortement liées, c'est-à-dire faisant intervenir les rôles des mêmes

entités. Ainsi, cette vue est constituée de plusieurs diagrammes représentant les différentes collaborations.

- **La vue de conception de composants** permet au concepteur de composant de décrire la structure des éléments de l'architecture du composant ainsi que leurs relations. Les différents constituants du composant et leurs associations avec des éléments internes ou externes à ce composant sont représentés dans un formalisme objet. L'intégration du composant à ce niveau enrichit la vue de conception de composant qui est un modèle statique d'analyse ressemblant au diagramme de classes par association de ses éléments et de ceux du système.

- **La vue d'assemblage de composants** donne une représentation des composants et de leurs interactions indépendamment d'une technologie mais dans un formalisme composant. L'assembleur intègre de nouveaux composants par la notion de connexion dans la vue d'assemblage.

4.3. Démarche à base de composants multi-vues

Proposer un composant multi-vues ou un patron d'assemblage des composants logiciels centré utilisateur n'étaient pas les seuls objectifs de ce travail, mais également définir une méthodologie à base de composants multi-vues (Multi-views Component Method : **MCM**) [Rehioui 2014b][Rehioui 2014c]. La démarche MCM s'effectue en 3 phases : phase centralisée de modélisation des exigences (phase d'analyse), phase décentralisée de modélisation du système suivant chaque type d'utilisateur (élaboration des sous-modèles) et une phase centralisée de fusion et de modélisation produisant le diagramme de composants multi-vues.

4.3.1. Modèles de la démarche MCM

La démarche MCM est orientée composant. Les phases principales de cette démarche sont inspirées principalement de la méthode U_VBOOM [Hair 2005] et la méthode CUP. Elle facilite l'intégration et la réutilisation de composants. Le modèle général de la démarche proposée peut être basiquement schématisé en quatre modèles (cf. figure 4.1) [Hair 2016].

- **Modèle de cas d'utilisation** permet à l'analyste d'exprimer les besoins attendus par les utilisateurs finaux. Dans ce modèle, un système est défini par un ensemble de cas d'utilisation qui décrivent ses spécifications fonctionnelles. L'intégration des composants qui constituent le système à ce niveau enrichit le modèle de cas d'utilisation entière du système.

- **Modèle d'interactions** permet à l'analyste d'exprimer le comportement des éléments de l'architecture, les uns par rapport aux autres, lors de la réalisation des cas d'utilisation. Ces éléments sont essentiellement des instances des composants constituant les parties internes du composant. Le comportement interne d'un composant est décrit par la vue d'interactions qui s'apparente aux diagrammes de séquences UML. Un composant regroupe un certain nombre de collaborations fortement liées, c'est-à-dire faisant intervenir les rôles des mêmes entités. Ainsi, ce modèle est constitué de plusieurs diagrammes représentant les différentes collaborations.

- **Modèle de conception** permet au concepteur de décrire la structure des éléments du système ainsi que leurs relations. Les différents constituants du composant et leurs associations avec des éléments internes ou externes à ce composant sont représentés dans un formalisme objet. L'intégration du composant à ce niveau enrichit la vue de conception de composant qui est un modèle statique d'analyse ressemblant au diagramme de classes par association de ses éléments et de ceux du système.

- **Modèle de fusion** permet de représenter la connexion des composants entre eux et de leurs interactions indépendamment d'une technologie. Il permet aussi de construire le système global ou de réutiliser de nouveaux composants existants.

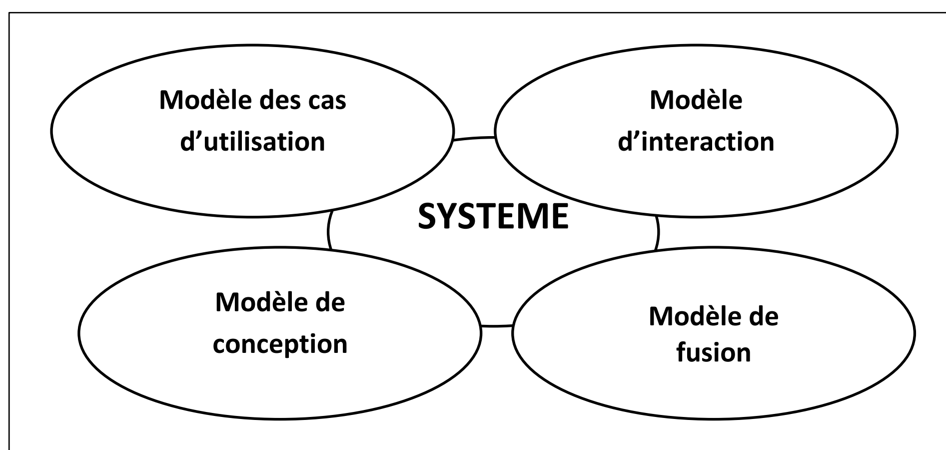


Figure 4.1. *Représentation du modèle de développement*

4.3.2. Description générale de la démarche MCM

La démarche MCM est orientée composant. Les phases principales de cette démarche sont inspirées principalement de la méthode U_VBOOM [Hair 2005] [Hair 2004b] et la méthode CUP. Elle facilite l'intégration et la réutilisation de composants (cf. figure 4.2.).

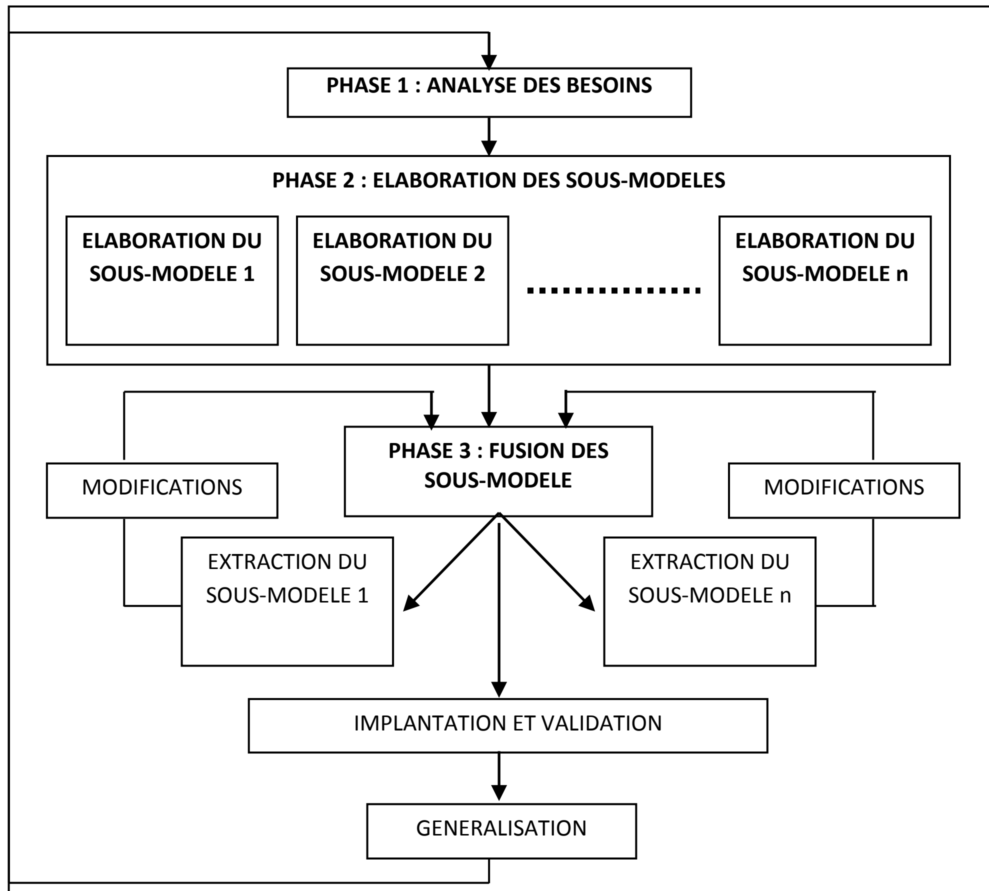


Figure 4.2. *Représentation du modèle de développement de la démarche MCM*

Nous nous appuyons sur la modélisation de l'étude de cas *Gestion d'une conférence Scientifique* pour illustrer les phases de la démarche. Une telle conférence a pour objectif de présenter les derniers résultats de recherche d'un domaine scientifique. Les différents acteurs du système Conférence sont [Hair 2012] :

- Le **comité de programme** chargé de :
  - traiter le contenu scientifique de la conférence,
  - préparer une liste de personnes pour lesquels un appel à communiquer est à envoyer,
  - répartir les papiers entre ceux soumis à l'arbitrage des lecteurs et ceux qui ne le sont pas et de sélectionner les papiers à inclure dans le programme,
  - etc.

- Le **comité d’organisation** chargé de :

- trouver un local pour la conférence,
- préparer les invitations et d’assurer le service hôtelier pour les participants,
- etc.

- Le **comptable** chargé de :

- questions financières telles que le traitement des droits d’inscriptions,
- etc.

- Le **secrétariat** de la conférence qui a pour rôle de :

- enregistrer les lettres d’intention reçues en réponse à l’appel à communication,
- enregistrer les contributions et d’inscrire les participants,
- etc.

- Le **président** de la conférence qui se charge de :

- définir le budget de la conférence et de veiller à son bon déroulement
- etc.

4.3.2. Phase 1: Analyse des besoins

Les cas d'utilisation servent à exprimer les besoins fonctionnels des utilisateurs d'un système. D'autres types d'exigences peuvent être joints aux descriptions de cas d'utilisation, notamment les exigences non fonctionnelles qui ne sont pas prises en compte volontairement. Le workflow « capture des besoins » correspond à l'identification des besoins qui, une fois implémentés, apportent un plus aux utilisateurs d'un système. Ils sont exprimés de façon compréhensible par les utilisateurs dans le modèle de cas d'utilisation. Le modèle de cas d'utilisation joue le rôle d'intermédiaire entre le langage naturel de l'utilisateur expliquant ce qu'il attend du système, et le langage des développeurs. Un système est utilisé par plusieurs types d'utilisateurs catégorisés en acteurs. Les interactions entre les acteurs et le système sont exprimées dans les cas d'utilisation [Caron et al. 2004].

L'enchaînement d'activités de capture des besoins commence par l'identification des acteurs. Tous les utilisateurs et les systèmes qui doivent dialoguer avec le système sont catégorisés en acteurs. Un acteur est une entité humaine ou machine externe au système et qui interagit avec le système. L'ensemble des acteurs définit l’environnement du système.

Les cas d'utilisation spécifient le système. Ils sont représentés par des ovales. En analysant toutes les utilisations potentielles des différents acteurs, l'ensemble des besoins fonctionnels est répertorié dans les cas d'utilisation. La figure 4.3. représente les acteurs et cas d'utilisation de l'étude de cas du système de Gestion d'une Conférence Scientifique.

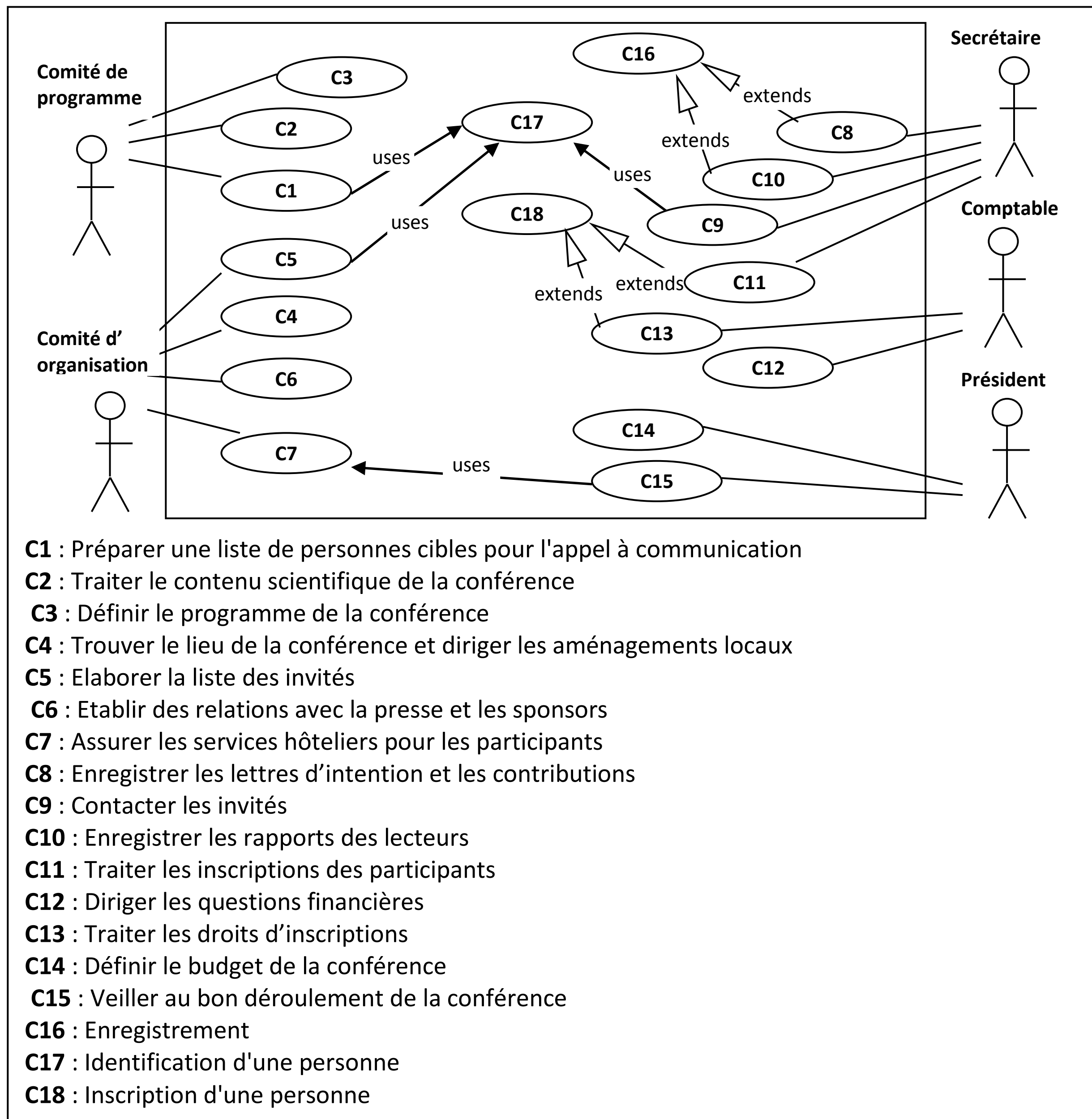


Figure 4.3. *Diagramme de cas d'utilisation du système conférence*

Les fonctionnalités décrites dans ce modèle sont amenées à être prises en charge par un ou plusieurs composants, constituant la structure interne du système considéré. Pour élaborer les composants vues du système, une technique de décomposition de cas d'utilisation est utilisée. Cette technique consiste à décrire chaque cas d'utilisation par un ensemble de cas d'utilisation plus élémentaire. Le Tableau 4.1. ci-

après illustre la description des cas d'utilisation du système conférence selon les différents acteurs. Chaque cas d'utilisation élémentaire possède un numéro et un libellé (colonne "Décomposition en cas d'utilisation élémentaires").

Acteurs	Cas d'utilisation	Décomposition en cas d'utilisation élémentaires
Comité de programme	Préparer une liste de personnes pour lesquels un appel à communiquer est à envoyer	**a1 :** Consulter les actes précédents **a2 :** Prendre informations et préparer une liste sur les anciens participants **a3 :** Contrôler l'envoi des appels à communication aux anciens participants
	Traiter le contenu scientifique de la conférence	**a4 :** Proposer les domaines scientifiques **a5 :** Choisir les domaines de débats
	Définir le programme de la conférence	**a6 :** Choisir les papiers à faire relire **a7 :** Contrôler l'envoi des papiers à faire relire **a8 :** Recueillir les rapports des lecteurs **a9 :** Sélectionner les papiers à inclure dans le programme **a10 :** Grouper les papiers sélectionnés en session pour leur représentation **a11 :** Choisir un président pour chaque session
Comité d' organisation	Trouver le lieu de la conférence et faire les aménagements de locaux	**a12 :** Chercher des lieux où la conférence peut se dérouler **a13 :** Recevoir l'accord d'acceptation du responsable d'un lieu **a14 :** Aménager les locaux
	Préparer une liste des invités	**a15 :** Consulter les thèmes de la conférence **a16 :** Préparer une liste des invités **a17 :** Contrôler le contact avec invités
	Etablir des relations avec la presse et les sponsors	**a18 :** Visiter la presse **a19 :** Etablir des relations avec la presse **a20 :** Visiter les sponsors **a21 :** Etablir des relations avec les sponsors
	Assurer les services hôteliers pour les participants	**a22 :** Contacter et visiter les hôtels **a23 :** Réserver les chambres et les tables de repas pour les participants
Secrétaire	Enregistrer les lettres d'intention reçues en réponse à l'appel à communication et enregistrer les contributions dès leur réception	**a24 :** Prendre la liste des anciens participants **a25 :** envoyer des appels à communiquer aux anciens participants **a26 :** Recevoir les lettres d'intention en réponse à l'appel à communication **a27 :** Enregistrer les lettres d'intention reçues **a28 :** Recevoir les contributions **a29 :** Enregistrer les contributions
	Contacter les invités	**a30 :** Prendre la liste des invités **a31 :** Envoyer les invitations aux invités
	Enregistrer les rapports des lecteurs	**a32 :** Répartir les papiers entre les relecteurs **a33 :** Recevoir les rapports **a34 :** Enregistrer les rapports
	Traiter les inscriptions des participants	**a35 :** Identifier le participant **a36 :** Distribuer et vendre les actes aux participants
Comptable	Gérer les finances	**a37 :** Recevoir les factures **a38 :** Traiter les factures **a39 :** Recevoir les sponsors **a40 :** Traiter les subventions des sponsors
	Traiter les droits d'inscriptions	**a41 :** Identifier la personne **a42 :** Traiter le droit d'inscription

Président de la conférence	Définir le budget de la conférence	**a43 :** Voir les budgets des anciennes conférences **a44 :** Chercher des sponsors **a45** : Etablir les relations avec les sponsors
	Veiller au bon déroulement de la conférence	**a46 :** Contacter les deux comités, la secrétaire et l'agent comptable **a47 :** Faire des réunions avec les différents acteurs de la conférence

Tableau 4.1. *Décomposition des différents cas d'utilisation en cas d'utilisations élémentaires*

Les cas d'utilisation élémentaires identifiés vont permettre d'obtenir les composants logiciels vues du système. Un composant logiciel vue est un "abstraction de données" correspondant à une liste de cas d'utilisation élémentaires propres à un acteur donné ou résultat d'une intersection des cas d'utilisation élémentaires d'un groupe d'acteurs. Ce composant correspond alors à la réalisation d'un regroupement de cas d'utilisation qui spécifie un besoin donné. Cette façon de découper le système porte le nom de modularité des cas d'utilisation introduit par Jacobson [Jacobson 1994][Pawlak et al. 2004]. Ainsi, la factorisation des données associées aux cas d'utilisation élémentaires (a1 à a47) des 5 acteurs, **Comité de Programme, Comité d'Organisation, Secrétaire, Comptable, Directeur,** du système *Conférence* nous a conduit à produire les 9 composants logiciels vues suivants comme décrit sur la figure 4.4 :

- **Actes** : {a1, a36}
- **Informations** : {a2, a3, a16, a17, a24, a25, a26, a27, a30, a31}
- **Animation_scientifique** : {a4, a5, a15}
- **Contributions** : {a6, a7, a8, a9, a10, a29, a33, a34}
- **Logistique** : {a12, a13, a14, a22, a23}
- **Relations_Extérieures** : {a18, a19, a20, a21, a44, a45}
- **Inscriptions** : {a35, a41, a42}
- **Finances** : {a37, a38, a39, a40, a43}
- **Déroulement :** {a46, a47} (cf. figure 4.4).

A titre d'exemple, l'acteur comité de programme peut utiliser le système selon les composants vues **Actes, Informations**, **Contributions** et **Animation_scientifique**. Autrement dit, cet acteur voit la conférence à travers les **Actes,** les **Informations**, les **Contributions** et une **Animation scientifique**. De même, pour les autres acteurs : Comité d'organisation, Secrétaire, Comptable et Directeur.

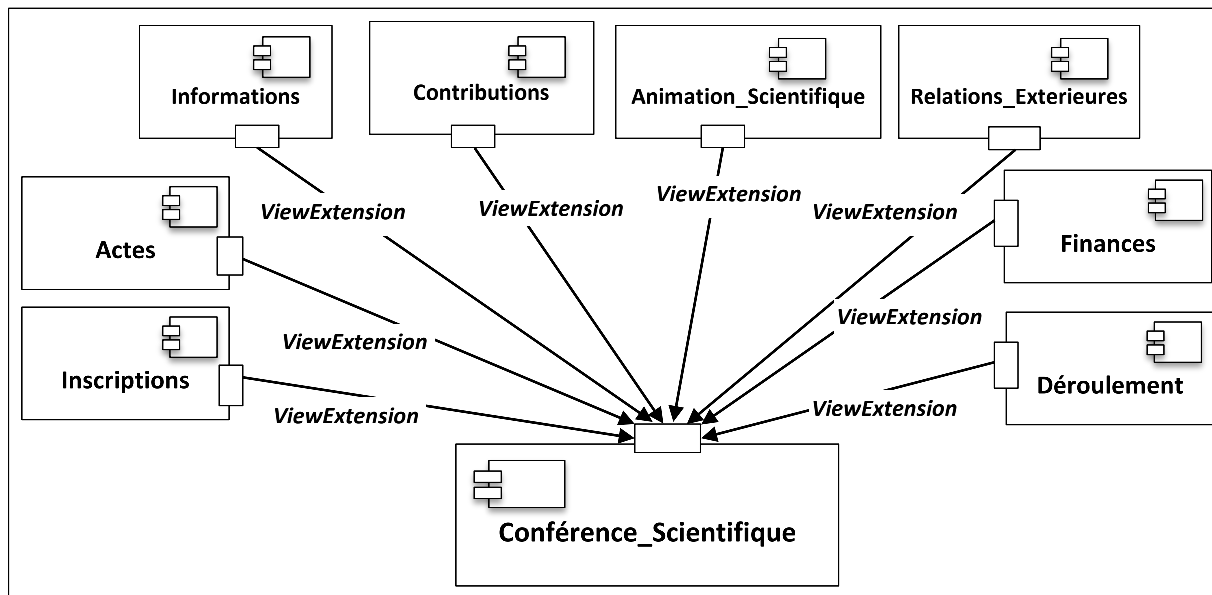


Figure 4.4. *Diagramme de composant multi-vues*

4.3.3. Phase 2 : Elaboration des sous-modèles

L'objectif de cette deuxième phase est de concevoir les sous-modèles du système associés aux acteurs identifiés précédemment. C'est une phase importante dans la définition des composants et la compréhension de leur fonctionnement dans le système. Elle consiste à découper l'espace de solution du problème en sous-domaines plus simples à aborder et modéliser séparément [Hair et al. 2002].

Le découpage du système peut être réalisé en deux temps :

- découpage selon les acteurs (type d'utilisation) ;
- découpage selon les cas d'utilisation.

Cette deuxième phase doit commencer par le découpage selon les acteurs pour obtenir les sous-modèles du système. Les sous-modèles représentent une sous-structure logique du modèle général puisqu'ils se reposent sur le concept d'acteurs du système. Nous pouvons considérer que chaque sous-modèle correspond à un acteur du système parmi les acteurs identifiés au cours de la phase précédente.

La conception de chaque sous-modèle peut se faire de façon autonome et indépendante des autres sous-modèles et peut être menée en parallèle avec la conception des autres sous-modèles par plusieurs concepteurs, ce qui donne une grande souplesse dans la modélisation des systèmes complexes.

En se basant sur l'ensemble de composants logiciels communs (composants partagés) et identifiés précédemment (au cours de la première phase), le concepteur doit élaborer pour chaque sous-modèle un dictionnaire partiel de composants. Dans le cas

de notre exemple, le système *conférence* peut être découpé en cinq sous-modèles : **Programmation, Organisation, Secrétariat, Comptabilité** et **Direction**.

Concernant le deuxième découpage, il consiste à décomposer chaque sous-modèle selon les cas d'utilisation de l'acteur. Ainsi, pour chaque cas d'utilisation on obtient un modèle de cas d'utilisation. Ce découpage est encore logique parce qu'il est basé sur le concept de cas d'utilisation.

La conception d'un sous-modèle peut être menée de deux façons différentes : un développement en parallèle (décentralisé) ou un développement incrémental en réalisant ses modèles de cas d'utilisation.

Développement en parallèle : La conception des modèles de cas d'utilisation est élaborée de manière disparate et autonome par les différents concepteurs. L'ordonnancement du démarrage des conceptions des modèles de cas d'utilisation est variable en fonction des ressources humaines et matérielles et de l'expérience des concepteurs (cf. figure 4.5).

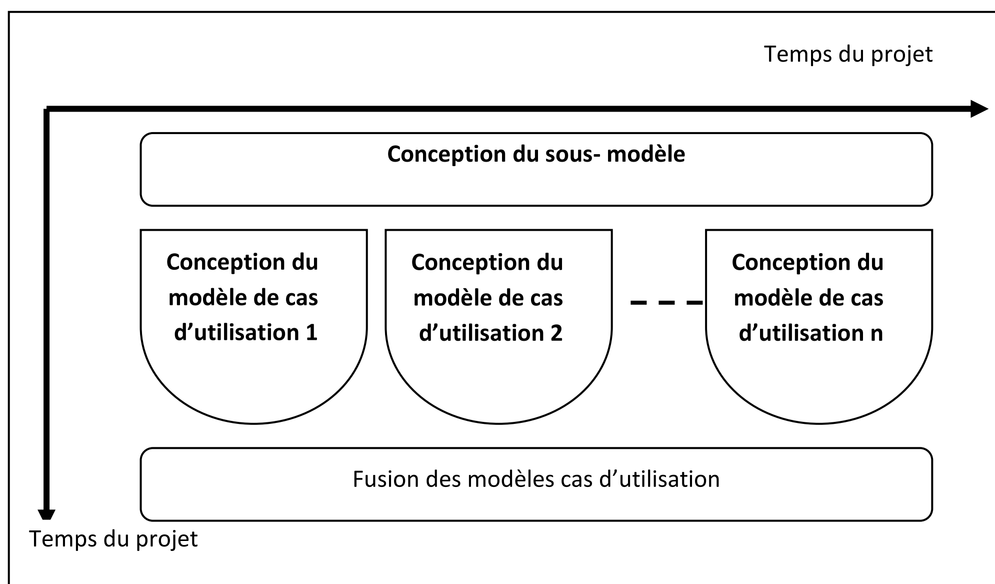


Figure 4.5. *Développement en parallèle des modèles de cas d'utilisation*

Pour obtenir le diagramme partiel de composants du sous-modèle, l'opération de fusion des diagrammes de composants des modèles de cas d'utilisation doit être réalisée. Cette opération de fusion peut être synthétisée par les deux sous-étapes suivantes :

1- Fusionner selon un certain ordre les interfaces partielles des composants, en gérant les conflits et les redondances qui apparaissent au cours de la fusion des modèles de cas d'utilisation.

2- Optimiser les interfaces, puis produire le diagramme partiel de composants du sous-modèle.

La conception du sous-modèle PROGRAMMATION associé à l'acteur Comité de programme du système *conférence*, revient à la conception et la fusion des quatre modèles de cas d'utilisation :

1- Préparer une liste de personnes pour lesquelles un appel à communiquer est à envoyer,

2- Traiter le contenu scientifique de la conférence,

3- Répartir les papiers soumis à l'arbitrage,

4- Définir le programme de la conférence.

Développement incrémental : Dans le cas extrême lorsque le démarrage d'un développement en parallèle ne peut pas se produire, alors le développement incrémental des modèles de cas d'utilisation est obligatoire (cf. figure 4.6). Avec ce type de développement, l'étape de la fusion des modèles de cas d'utilisation sera éliminée et l'élaboration du diagramme de composants et la définition des interfaces des composants du sous-modèle s'enrichissent au fur et à mesure de la conception de chaque modèle de cas d'utilisation.

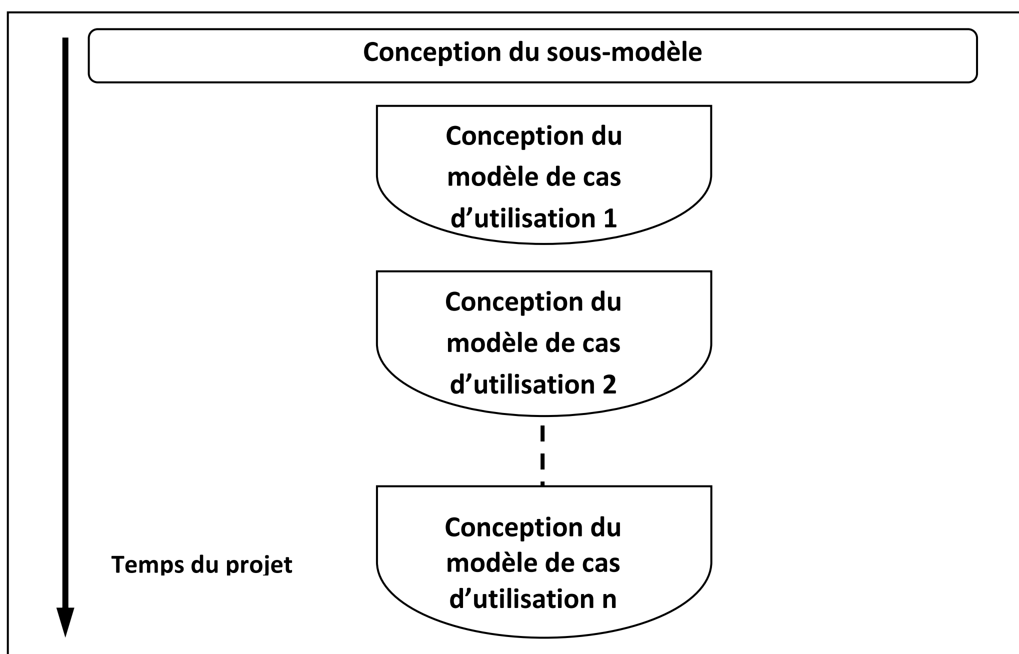


Figure 4.6. *Développement incrémental des modèles de cas d'utilisation*

Pour éviter toute lourdeur dans le développement d'un système complexe, nous conseillons de suivre un développement incrémental des modèles de cas d'utilisation. Ainsi, nous adoptons nous même le développement incrémental pour concevoir un tel sous-modèle.

La conception d'un sous-modèle peut être réalisée par les sous-étapes suivantes :

- élaboration du diagramme de collaboration décrivant le cas d'utilisation et définition des composants qui sont propre au modèle de cas d'utilisation ;

- élaboration du diagramme de composants associé au modèle de cas d'utilisation. Ce diagramme permet alors à chaque fois d'enrichir le diagramme de composants du sous-modèle correspondant.

Le dictionnaire initial de composants du modèle de cas d'utilisation est celui de son sous-modèle à l'exception des composants logiciels vues qui n'ont pas fait appel à un cas d'utilisation élémentaire du cas d'utilisation courant. Par exemple pour le sous-modèle PROGRAMMATION, les composants logiciels vues qui font appel à un cas d'utilisation élémentaire du cas d'utilisation *Préparer une liste de personnes pour lesquelles un appel à communiquer est à envoyer* sont : **C1=Acte**, **C2=Information** ; pour le cas d'utilisation *Traiter le contenu scientifique de la conférence* est le composant logiciel vue **C3=Scientifique**, et enfin pour le cas d'utilisation *Définir le programme de la conférence* est composant logiciel vue **C4=Contributions**.

4.3.3.1. Elaboration de diagramme de collaboration et description des composants

Le concepteur peut ajouter des composants qui sont propres au modèle de cas d'utilisation et qui sont également propres au sous-modèle associé. Ces composants sont appelés : composants partiels.

Selon Jacobson [Jacobson et al. 1999] [Jacobson 2003], un cas d'utilisation correspond à une séquence d'actions, et ses variantes, pouvant être effectuées par le système et produisant un résultat satisfaisant pour un acteur particulier. Cette séquence d'actions est nommée scénario. Le scénario correspond à un chemin d'exécution du cas d'utilisation. Chaque scénario, décrivant un cas d'utilisation, peut être réalisé par un diagramme de séquence UML pour représenter les interactions entre les composants. Et puis, par un diagramme de collaboration pour illustrer la coopération entre les composants utilisés pour la réalisation d'une fonctionnalité [Muller 1997] [OMG 2004]. Les figures ci-dessus (4.7, 4.8 et 4.9) illustrent respectivement le scénario, le diagramme de séquence et le diagramme de collaboration entre composants associés au cas d'utilisation *Définir le programme de la conférence* du système de la conférence.

1. Le comité de programme envoie les papiers soumis aux lecteurs et relance une date limite.

2. Les lecteurs rédigent des rapports et les envoient au comité de programme.

3. Le comité de programme rassemble les rapports et sélectionne les papiers à inclure dans le programme.

4. Le comité de programme répartit les papiers sélectionnés en session.

5. Le comité de programme désigne un président pour la session.

Figure 4.7. *Description textuelle du scénario Définir le programme de la conférence*

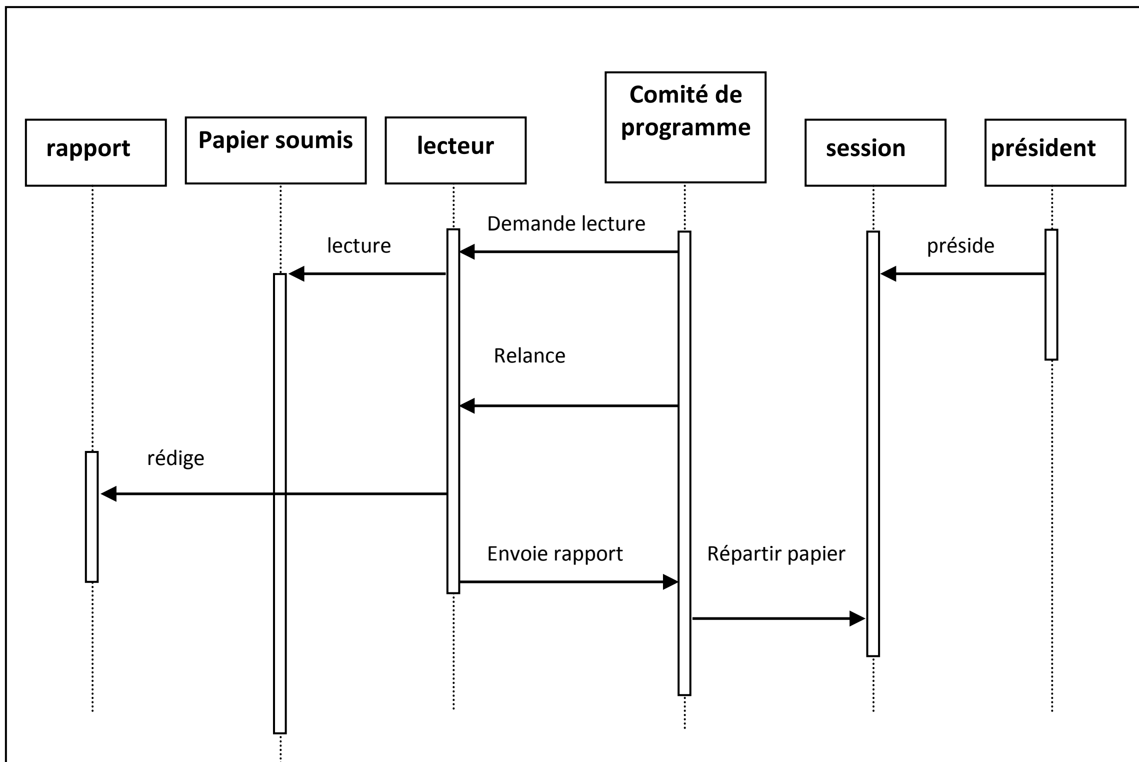


Figure 4.8. *Diagramme de séquence du scénario : Définir le programme de la conférence*

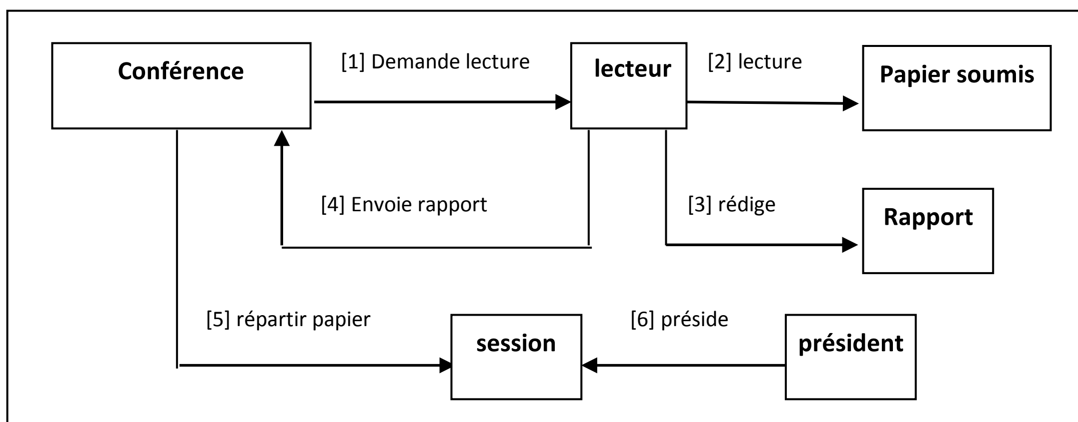


Figure 4.9. *Diagramme de collaboration entre composants du scénario : Définir le programme de la conférence*

Après avoir identifié les différents composants réalisant le scénario du cas d'utilisation, on procède à leurs descriptions partielles. La description partielle d'un composant consiste à définir les interfaces (fournies et requises) du composant logiciel en précisant le typage des attributs, traduire les relations d'associations, d'agrégations, etc. La figure 4.10 représente les deux composants SESSION et CONFERENCE.

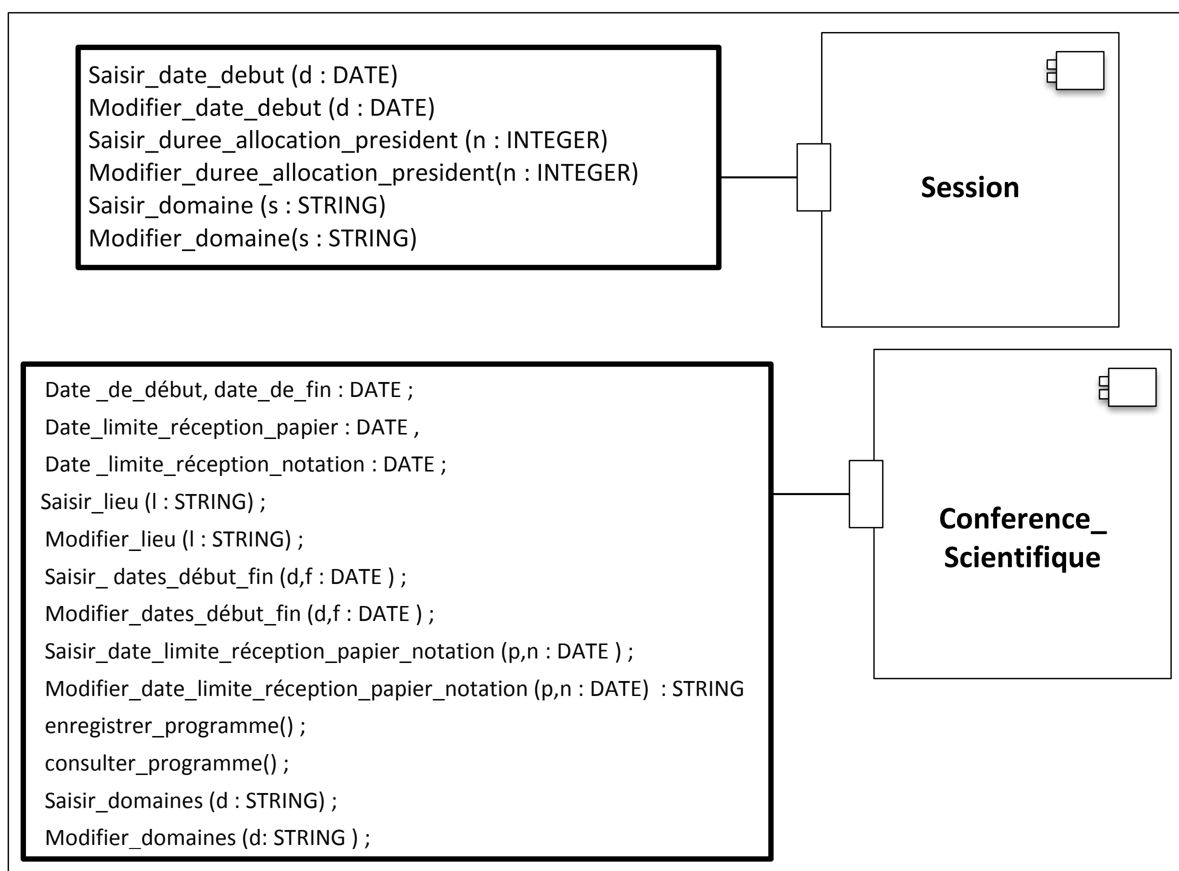


Figure 4.10. *Composants partiels de Session et Conference_Scientifique associés sous-modèle PROGRAMMATION*

4.3.3.2. Définition de diagramme partiel de composants

Le diagramme partiel de composants d'un modèle de cas d'utilisation, représente une partie de diagramme de composants du sous-modèle. La Figure 4.11 illustre le diagramme partiel de composants correspondant au modèle de cas d'utilisation *Définir le programme de la conférence*.

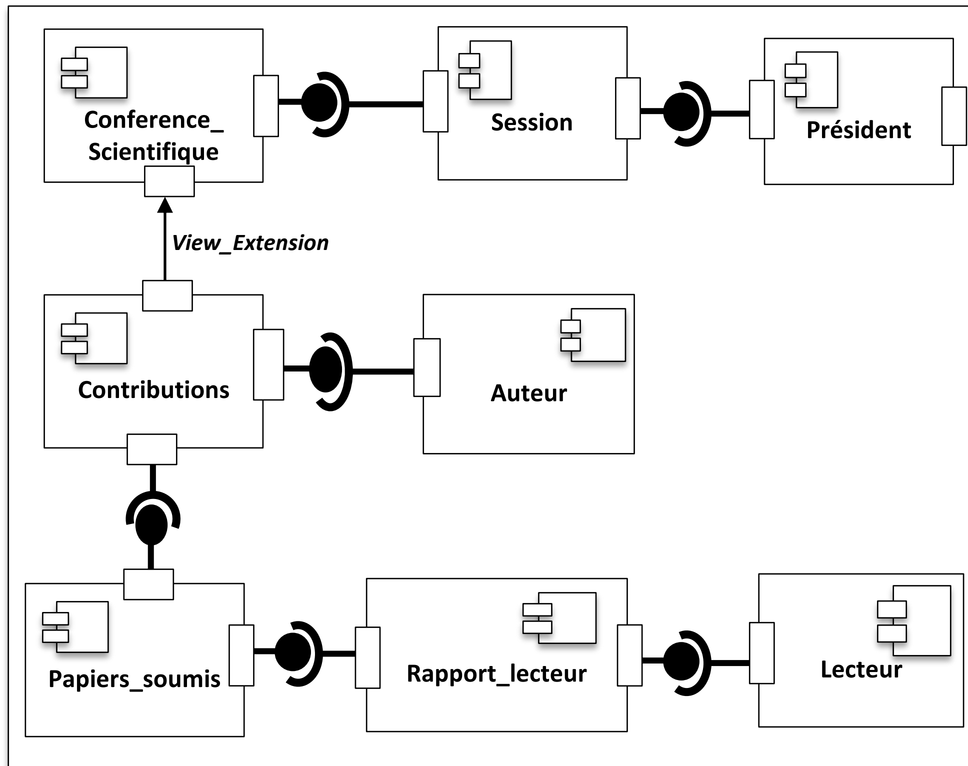


Figure 4.11. *Diagramme partiel de composants du modèle de cas d'utilisation Définir le programme de la conférence*

4.3.3.3. Elaboration de diagramme de composants du sous-modèle

En procédant de la même façon, c'est à dire en appliquant les mêmes sous-étapes de conception d'un modèle de cas d'utilisation aux autres cas d'utilisation du sous-modèle PROGRAMMATION, on obtient le diagramme de composants associé au sous-modèle (cf. figure 4.12). Rehioui et al. [Rehioui et al 2015a] [Rehioui et al 2015b] contiennent plus de détails de la conception des autres modèles de cas d'utilisations.

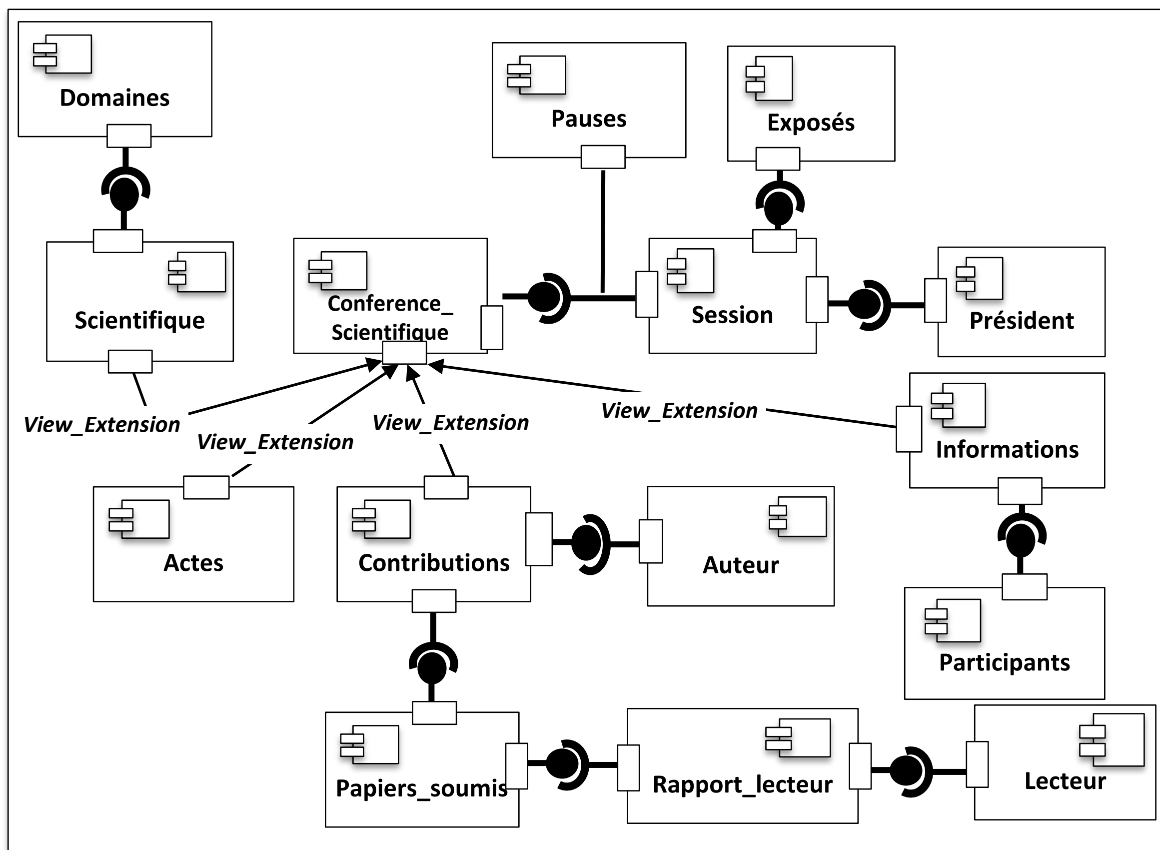


Figure 4.12. *Diagramme partiel de composants du sous-modèle PROGRAMMATION*

4.3.4. Phase 3 : Conception du modèle global

Au cours de cette phase les concepteurs sont assistés par une démarche pour fusionner le plus efficacement possible les résultats partiels issus de la conception des différents sous-modèles en leur permettant de résoudre stratégiquement les différents conflits pouvant apparaître au cours de la fusion. Les solutions proposées sont automatiques à chaque fois que c'est possible et interactives dans le cas où l'intervention du concepteur serait indispensable. Cette dernière phase fait l'objet de la fusion des résultats partiels obtenus à l'issue de la phase précédente pour élaborer la solution générale du système. Elle est décomposée en deux étapes.

4.3.4.1. Fusion des composants partiels

L'objectif de cette étape est d'obtenir une description complète de chacun des composants après fusion des composants partiels issus des sous-modèles. Cette fusion est accompagnée d'une gestion des conflits : la synonymie et la polysémie des caractéristiques, l'adaptation des paramètres des méthodes et éventuellement le typage, le renommage des caractéristiques dans le cas de synonymie, etc. [Hair 2007][Hair 2012].

4.3.4.2. Etablissement du diagramme de composants du modèle général

On pourrait intuitivement considérer que le diagramme de composants du modèle général est une superposition des sous-modèles correspondant à chaque acteur couvrant l'ensemble des différents types d'utilisations du système. Il suffit d'imaginer la représentation de chaque sous-modèle illustrée sur un transparent et de superposer l'ensemble de ces transparents pour obtenir le diagramme de composants du modèle général.

Cette étape consiste à illustrer le diagramme de composants du modèle général (cf. figure 4.13) déduit à partir des composants fusionnés. Il est souvent utile de mener une comparaison avec les résultats issus d'une superposition des diagrammes de composants des sous modèles pour des éventuelles adaptations en cas de conflits persistant.

4.4. MCM une démarche dirigée par les modèles et outil support

4.4.1. MCM sous MDA

Le développement d'un système à l'aide de la démarche MCM s'effectue en quatre phases (cf. figure 4.14) : trois phases de la démarche MCM et une phase de prise en compte des plates-formes d'exécution.

La première phase de la démarche MCM est la modélisation des exigences qui est une phase centralisée et fait partie du niveau CIM (Computation Independent Model) de l'approche MDA. Cette phase est principalement réalisée par les cas d'utilisation. Elle permet d'identifier les différents acteurs qui interagissent avec l'application ainsi que les besoins de chaque acteur. De plus, des liens de traçabilité peuvent être créés depuis les exigences vers le code de l'application.

La deuxième phase de la démarche MCM est l'élaboration des sous-modèles (PIM). Ces PIM devraient en théorie être générés à partir des modèles CIM élaborés dans la phase de modélisation des exigences. Cette phase consiste à faire les spécifications par acteur. Pour chaque acteur, nous élaborons les diagrammes de séquence de chaque cas d'utilisation auquel il participe. A partir de ces diagrammes de séquences nous identifions les composants partiels du diagramme de composants associé à cet acteur. Le résultat de cette phase est un diagramme de composants partiels par acteur qui montre clairement les informations pertinentes auxquelles peut accéder l'acteur associé. Les différents PIM par acteurs élaborés dans la deuxième phase de modélisation doivent être enrichis par étapes successives.

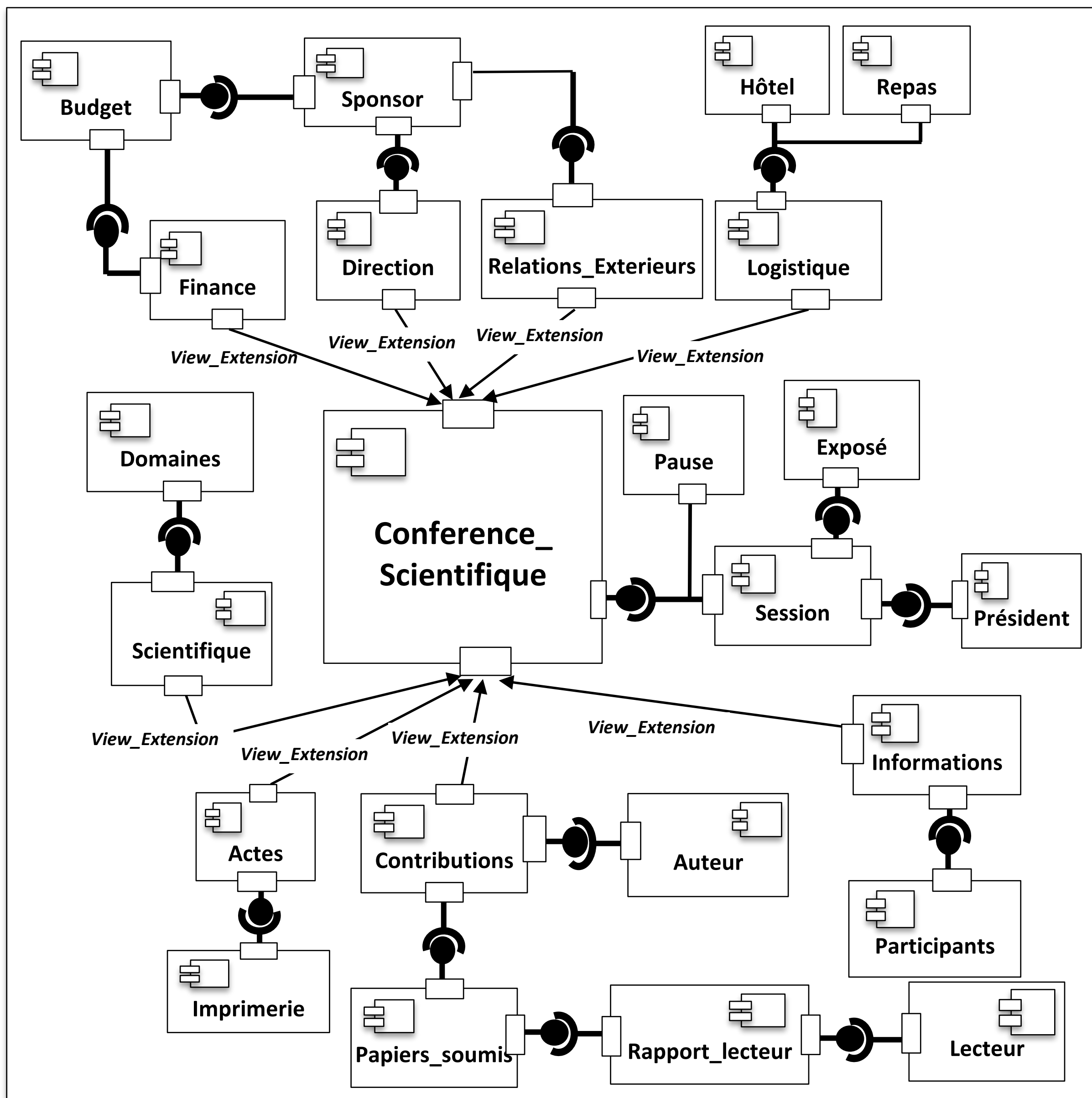


Figure 4.13. *Diagramme de composants de l'exemple conférence scientifique*

Une fois les PIM par acteur sont suffisamment détaillés, la troisième phase de fusion de ces sous-modèles de la démarche peut commencer. Cette fusion peut être considérée comme une transformation de modèles (PIM -> PIM). En effet, une telle transformation prend en entrée des PIM par acteur et génère en sortie un PIM.

Cette phase est ajoutée aux autres phases de la démarche et elle consiste à la prise en compte des plates-formes d'exécution dans le cycle de vie des applications. Elle a pour objectif de gérer la dépendance des applications vis-à-vis de leur plate-forme d'exécution. Dans le MDA, ces dépendances sont gérées comme une transformation de modèles. Nous préconisons dans cette phase l'utilisation du patron développé

dans le chapitre 3. Le modèle PSM généré peut être facilement utilisé pour générer du code dans des langages orientés composants.

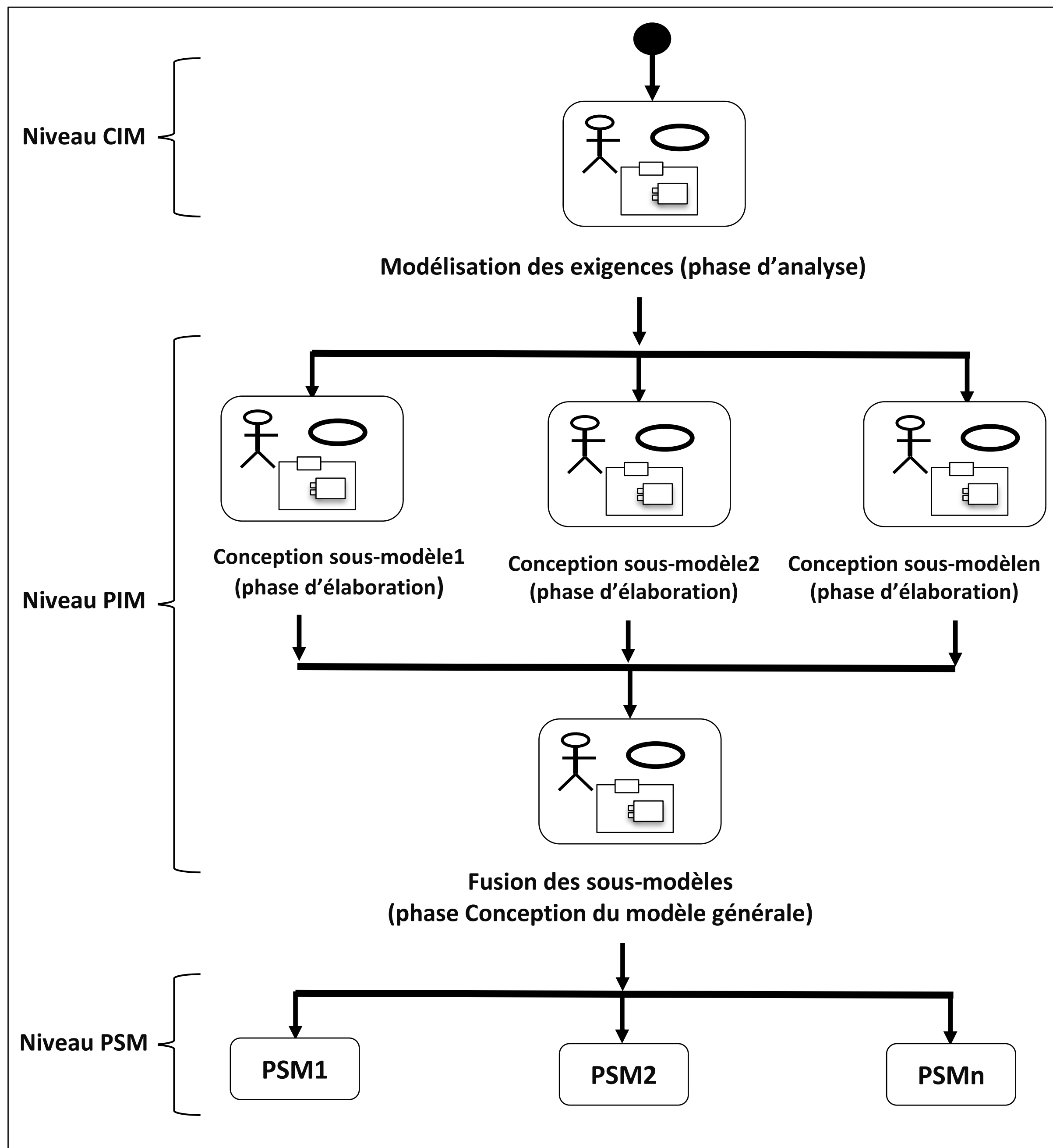


Figure 4.14. *Démarche MCM dirigée par les modèles MDA*

4.4.2. Outil support à MCM

Pour appliquer efficacement l'approche MCM, nous avons réalisé un prototype support à MCM. Ce prototype a été développé sous l'atelier Objecteering/UML [Objecteering 2015]. Le choix de cet atelier comme plate-forme de développement se justifie par le fait qu'il supporte une gestion de profils. En effet, grâce aux profils il est

possible d'adapter l'outil *Objecteering/UML Modeler* (qui fait partie de l'atelier *Objecteering/UML*) pour tenir compte des contraintes de modélisation propres à des projets (cf. figure 4.15). Ceci permet de réduire considérablement le temps nécessaire pour réaliser des outils supportant des extensions d'UML.

Le langage J [Objecteering 2015] est l'un des constituants essentiels de la puissance de *UML Profile Builder*. C'est un langage qui permet de paramétrer et de piloter l'atelier *Objecteering/UML*. J est un langage objet interprété, à la syntaxe Java, dédié à manipuler les modèles. Son code peut ainsi être modifié et testé rapidement. J s'exécute au niveau métamodèle et manipule les éléments de modélisation créés par l'utilisateur.

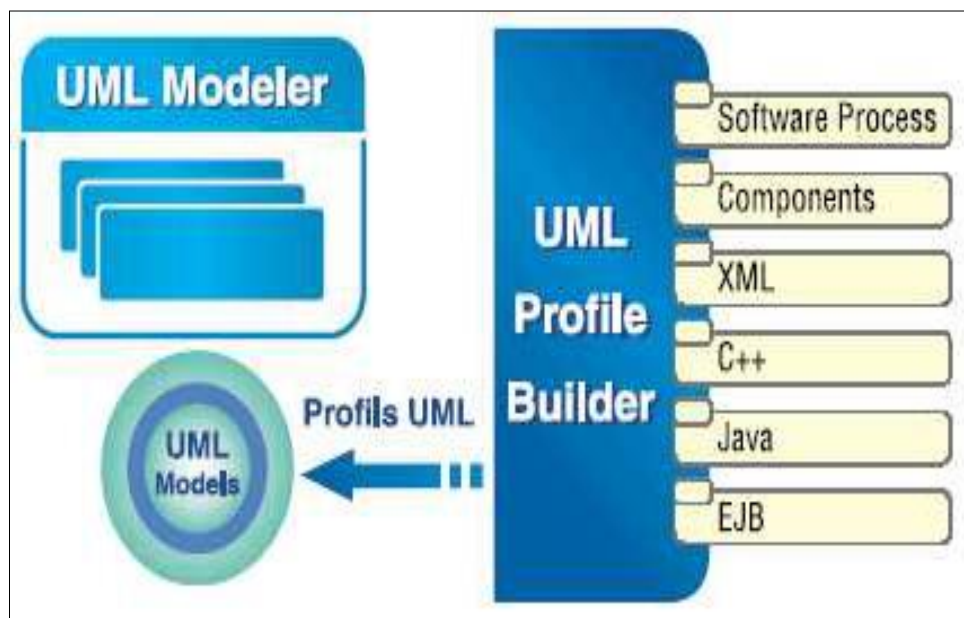


Figure 4.15. *Adaptation d'Objecteering/UML Modeler par des Profils UML réalisés sous Objecteering/UML Profile Builder*

Afin de mettre en œuvre le prototype support à MCM, nous avons développé deux profils Objecteering :

- Le profil Objecteering MCM qui permet de mener une modélisation selon la démarche MCM et de vérifier la conformité des diagrammes avec la sémantique du MCM. En fait, la sémantique statique de MCM est définie par le métamodèle, des règles de bonne modélisation (well-formedness rules) exprimées en langage formel OCL (Object Constraint Language) et des descriptions textuelles informelles. Vu que l'atelier Objecteering ne supporte pas le langage OCL, nous avons traduit les règles de bonne modélisation en langage J.

- Le profil Objecteering « Générateur de code Java » qui permet la génération du code Java à partir d'une modélisation MCM. Cette génération de code s'appuie sur le patron d'assemblage à base de composants multi-vues.

4.5. Conclusion

Nous avons présenté dans ce chapitre la méthode MCM à base de composants multi-vues. La méthode est basée sur les quatre modèles : modèle de cas d'utilisation, modèle d'interaction, modèle de conception et modèle de fusion. La démarche MCM propose trois phases de modélisation. Une phase centralisée de modélisation des exigences, une phase décentralisée de modélisation du système suivant chaque type d'utilisateur (acteur) et une phase centralisée de fusion et de modélisation produisant le diagramme de composants multi-vues.

En ajoutant une phase de prise en compte des plates-formes d'exécution la démarche MCM s'intègre facilement dans l'approche MDA.

Ce travail a été validé par la réalisation d'un prototype support à MCM. Il a été implémenté en particularisant l'atelier *Objecteering/UML* grâce à la technique des profils.

Un travail de thèse a été lancé récemment sur l'affinage de la méthode MCM et son intégration dans l'approche MDA.

Conclusion Générale et Perspectives

Les recherches menées portent principalement sur le développement des systèmes logiciels à composants, ainsi que leur réalisation. Dans ce domaine, le CBSE est actuellement reconnu, tant par le monde académique que par le monde industriel, comme une approche prometteuse. Son utilisation contribue notamment à améliorer la réutilisabilité, la structuration des architectures logicielles et la modularité des applications.

Dans ce mémoire, nous avons présenté notre étude visant à améliorer la prise en compte de l'utilisateur (vue/point de vue) dans l'assemblage des composants logiciels et nous nous sommes encore intéressés aux méthodes basées composants centrées utilisateur dans le cadre de l'ingénierie logicielle basée composant CBSE.

Dans un premier temps, nous avons mené une étude terminologique sur les composants logiciels et les concepts liés et nous nous sommes positionnés sur les plates-formes les plus utilisés. Dans un deuxième temps, nous avons dressé un état de l'art sur l'architecture dirigée par les modèles MDA et en particulier le langage UML 2.0, qui est actuellement un standard de la modélisation.

Intégrer le concept vue/point de vue dans la construction d'un système logiciel à base de composants permet de tenir compte simultanément le maximum possible de tous les besoins des acteurs.

Dans ce cadre, nous avons proposé deux modèles d'assemblage de composants logiciels basés sur le concept vue/point de vue. Le premier modèle est hiérarchique basé sur les propriétés de la relation de visibilité issue de ce concept. Nous avons proposé dans ce modèle un nouveau connecteur appelé connecteur de visibilité et nous avons établi les règles d'assemblage entre les composants logiciels pour construire un composant multi-vues. Le deuxième modèle est un patron

d’assemblage basé sur l’implémentation de la relation de visibilité. Ce patron a été amené par un diagramme d’état instanciation qui illustre les différentes étapes à suivre pour utiliser ce patron d’assemblage. Dans ce modèle, nous avons proposé un profil UML qui implémente ce modèle d’assemblage. Ce profil définit les extensions, les règles et les contraintes à respecter pour construire un assemblage de composant basé sur ce modèle. Enfin, nous avons développé une démarche d’analyse et de conception centrée utilisateur et basée sur les composants logiciels. La démarche est basée sur quatre modèles : modèle de cas d’utilisation, modèle d'interactions, modèle de conception et modèle de fusion. Cette démarche propose trois phases de développement : la première phase est la modélisation des exigences qui est une phase centralisée ; la deuxième est la phase d’élaboration des sous modèles associés à chaque type d’utilisateur du système et qui une phase décentralisée et la troisième phase est la phase de conception du modèle global qui est assistée par une démarche de fusion des sous modèles. En ajoutant une quatrième phase de prise en compte des plates-formes d’exécution, la démarche proposée a été intégrée facilement dans l’approche MDA. Nous avons outillé notre approche en vue de fournir une plate-forme de développement à base de composants la plus complète possible. L’outil a été implémenté en particularisant l’atelier Objecteering/UML grâce à la technique des profils.

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

La démarche MCM proposée repose sur la décomposition des cas d’utilisation pour identifier les composants du système. Les cas d’utilisation sont exprimés de manière textuelle. Le langage humain est soumis à des interprétations et à des incompréhensions, une mauvaise lecture peut conduire un système à sa perte. Décomposer les cas d’utilisation en se servant de scénarios écrits en langage naturel et de les formaliser grâce à une syntaxe ET/OU et des bases mathématiques solides étayés par des propriétés prouvées et des algorithmes validés. L'intégration de ces travaux sous la forme d'une vue supplémentaire apporterait plus de fiabilité à la démarche et offrirait davantage de critères d'identification de composants.

La démarche MCM fournit un modèle indépendant d'une plate-forme technologique qui permet de mieux caractériser un composant. Une qualité MDA du processus est que le modèle MCM peut être projeté vers diverses plates-formes technologiques. Un travail de thèse a été lancé récemment sur l'affinage de la méthode MCM et son intégration dans l'approche MDA.

La réalisation d'un outil support complet de la démarche avec l'intégration d'un outil de vérification d'assemblage de composants ne peut qu'offrir et enrichir davantage la démarche.

Bibliographie

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

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

[Adams et al. 1995] Adams M., Coplien J., Gamoke R., Hanmer R., Keeve F., Nicodemus K., Fault-Tolerent Telecommunication System Patterns. AT&T Bell Laboratories, 1995.

[Alexander 79] Alexander C., The Timeless Way of Building, Oxford University Press, New York, NY, 1979.

[Alexander et al. 77] Alexander C., Ishikawa S., Silverstein M., Jacobson M., Fiksdahl-king I., Angel S., A Pattern Language. Oxford University Press, New York, NY, 1977.

[Ambler 1998] Ambler S.W., Process Patterns: Building Large-Scale Systems Using Object Technology. Cambridge University Press, 1998.

[Anderson et al. 1992] Anderson E.P. and Reenskaugt KAUG T., System Design by Composing Structures of Interacting Objects. In ECOOP'92, pages 133-152, Netherlands. Springer-Verlag, 1992.

[Anwar et al. 2010] Anwar A., Ebersold S., Coulette, B., Nassar, M., Kriouile, A.: A Rule-Driven Approach for composing Viewpoint-oriented Models. Journal of Object Technology, Zurich, Vol. 9, No. 2, pp. 89-114, 2010.

[Atkinson et al. 2001] Atkinson C., Paech B., Reinhold J., Sander T., Developing and Applying Component-BasedModel-Driven Architectures in KobrA. edoc, 00:0212, 2001.

[Bálek et al. 2001] Bálek D., Plásil F., Software connectors and their role in component deployment. In Proceedings of DAIS'01, Krakow, Poland, September 2001. Kluwer Academic Publishers, 2001.

[Barbier et al. 2004] Barbier F., Cauvet C., 0ussalah M., Rieu D., Bennasri S., Souveyet C., Concepts clés et techniques de réutilisation dans l'ingénierie des systèmes d'information. Ingénierie des composants dans les systèmes d'information, Numéro spécial de la revue l'Objet,10(1), 2004.

[Beck et al. 1987] Beck K., Cunningham W., Using pattern languages for object-oriented programs. Technical Report CR-87-43, Tektronix Inc, September 1987.

[Beugnard et al. 99] Beugnard A., Jézéquel J., Plouzeau N., and Watkins D., Making components contract aware. Lecture Notes in Computer Science, 32(7) :38–45, July 1999.

[Bézivin et al. 2004] Bézivin J., Baar T., Gardner T., Gogolla M., Hähnle R., Hußmann H., Patrascoiu O., Schmitt P. H., Warmer J., OCL and Model Driven Engineering. UML Satellite Activities ; pp. 67-75, 2004.

[Booch 1997] Booch G., Ingenierie du logiciel avec ADA: de la conception à la réalisation. InterEditions, Décembre 1997.

[Booch et al. 2005] Booch G., Rumbaugh J., Jacobson I., The Unified Modeling Language User Guide . (2nd Edition) Amazon, 2005.

[Bosch et al. 2003] Bosch J., Szyperski C., Weck W., Ws5, report on the Eighth International Workshop on Component-Oriented Programming (WCOP 2003), http://research.microsoft.com/~cszypers , 2003.

[Bray 04] T. BrayT., Extensible Markup Language(XML). W3C Recommendation, Février 2004.

[Brown 2000] Brown A. W., Large-Scale, Component-Based Development, Prentice Hall, 2000.

[Bruneton et al. 2003] Bruneton E., Coupaye T., Stefani J., Fractal Component Model Draft 2.0-2. France Telecom R&D INRIA, Septembre 2003.

[Bruneton 2004] Bruneton E., Developping with fractal. [Online]. Available : http ://fractal.objectweb.org /tutorial/ index.html, Septembre 2004.

[Bruneton et al. 2004] Bruneton E., Coupaye T., Stefani J. B., The Fractal Component Model, http://fractal.objectweb.org/. 2004.

[Buschmann 1996] Buschmann F., What is a pattern?. Object Expert, vol. 1, n°3, pp.17-18, 1996.

[Cheesman et al. 2001] Cheesman J. , Daniels J., UML Components - A Simple Process for SpecifyingComponent-Based Software. Addison-Wesley, 2001.

[Cai et al. 2000] Cai X., Lyu M.R., Wong K., Ko R., Component-Based Software Engineering : Technologies, Development Frameworks and Quality Assurance Schemes, in Proceedings APSEC 2000, Seventh Asia-Pacific Software Engineering Conference, Singapore, pp372-379, Décembre 2000

[Caron et al. 2004] Caron 0., Renaux E., and Geib J.M.. Vers de Véritables Composant EJB Réutilisables.Ingénierie des composants dans les systèmes d'information, Numéro spécial de la revue l'Objet, 10(1), 2004.

[Carré et al. 1991] Carré B., Geib J.M., The Point of View Notion for Multiple Inheritance, Proc. Of ECOOP/OOPSLA, 1991.

[Carrez 2003] Carrez C., Contrats Comportementaux pour Composants. Thèse de doctorat, École Nationale Supérieure des Télécommunications, Décembre 2003.

[Christensen et al. 2001] Christensen E., Curbera F., Meredith G., Weerawarana S., Web services description language (WSDL) 1.1. Technical report, W3C Note, Mars 2001.

[Coad 1992] Coad P., Object-Oriented Patterns. Communications of the ACM 35(9):152-159, 1992.

[Coad 1995] Coad P., Object Models – strategies, patterns and applications. Yourdon press computing series, 1995.

[Coad et al. 1997] Coad P., North D., Mayfield M., Object Models: Strategies, Patterns, and Applications. Prentice Hall, 1997.

[Collet et al. 2005] Philippe C., Roger R., Thierry C., Nicolas R,. A Contracting System for Hierarchical Components. In Component-Based Software Engineering, 8th International Symposium(CBSE'2005), volume 3489 de LNCS, pages 187–202, St-Louis (Missouri), USA, Springer Verlag, 2005.

[Coplien 1996] Coplien J.O., Patterns, the Patterns White Paper. 1996.

[Coulette et al. 1995] Coulette B., Kriouile A., Marcaillou S., L'approche par points de vue dans le développement orienté objet des systèmes complexes, Revue l'Objet, vol. 2, n°4, pp. 13-20, Février 1996.

[Crnkovic et al. 2002] Crnkovic I., Larsson M., Building Reliable Component-Based Software Systems, Artech House, 2002.

[D'Souza et al. 1998] D'Souza D., Wills A. C. , Objects, Components and Frameworks: The Catalysis Approach, Reading, MA», Addison-Wesley, 1998.

[Edwards et al. 2002] W. Edwards, S. Deshpande, Intégration de CORBA et d'EJB, Livre Blanc, pp. 1-14, Janvier 2002.

[Farias 2003] Farias A., Un modèle de composants avec des protocoles explicites. Thèse de doctorat, Université de Nantes, Décembre 2003.

[Fernandez et al. 2004] Fernandez L.F., A. Mareno V., An Introduction to UML profiles. European Journal for the Informatics Professional, pp. 6-13, 2004.

[Finkelstein et al. 1990] Finkelstein A., Kramer J., Goedicke M., Viewpoint Oriented Software Development, Proc. of S. Eng. and Applications Conference, Toulouse, pp. 337-351, Décembre 1990.

[Fontoura et al. 2002] Fontoura, M., Pree, W., Rumpe, B., The UML Profile for Framework Architectures. Addison-Wesley. 2002.

[Fowler 1997] Fowler M., Analysis Patterns – Reusable Object Models. Addison Wesley, 1997.

[Gamma 1991] Gamma E., Object-Oriented Software Developments based on ET++: Design Patterns, Class Library, Tools. PhD thesis, Université de Zurich, 1991.

[Gamma et al. 1995] Gamma E., Helm R., Johnson R., Vlissides J., Design Patterns, Elements of reusable Object-Oriented Software. Addison-Wesley Publishing Company, 1995.

[Garlen et al. 2002] Garlan D., Cheng S.W., Kompanek A., Reconciling the Needs of Architectural Description with Object-Modeling Notations, Science of Computer Programming 44, pp. 23-49, 2002.

[Goulao et al. 2003] Goulao. M, F.B. Abreu, Bridging the gap between ACME and UML 2.0 for CBS. In: Proceedings Workshop of Specification and Verification of Component-Based Systems, Helsinki, Finland, 2003.

[Greenfiel 01] Greenfield J., UML Profile for EJB, Technical report, Rational Software Corporation, Mai 2001.

[Gröne et al. 2005] Gröne B., Knöpfel A., et Tabeling P., Component vs. component : Why we need more than one definition. In ECBS, pages 550–552. IEEE Computer Society, 2005.

[Gudgin 2001] Gudgin M., Essential IDL: Interface Design for COM, Addison-Wesley, 2001.

[Hair 1997] Hair A., Conception de VBTOOL, outil support de la méthode VBOOM, réalisation des fonctionnalités : fusion des modèles visuels et génération du code. Thèse pour l'obtention du diplôme de spécialité de 3ème cycle de l'université Mohamed V, Rabat, Maroc, 1997.

[Hair et al. 1998] Hair A., El Asri B., Kriouile A., Coulette B., Outil support la méthode VBOOM, fonctionnalités : fusion et génération du code ; Acte de conférence, 4ème colloque africain sur la recherche en informatique, CARI'98, Dakar, Sénégal, 9-16 Octobre 1998.

[Hair et al. 2002] Hair A., Kriouile A, Coulette B. : Un processus d'analyse et de conception unifié basé sur le concept de point de vue. Actes 6ème colloque africain sur la recherche en informatique CARI'02, -Yaoundé, Cameroun -, pp.229-237, 14-17 Octobre 2002.

[Hair 2004a] Hair A., U_VBOOM: Unified analysis and design process based on the viewpoint concept, Actes de conférence, 6th International Conference on Enterprise Information systems (Porto/Portugal) ICEIS'04, 14-17 Avril 2004.

[Hair 2004b] Hair A., Analysis and design process based on the viewpoint concept, In RITA - Revista de Informática Teorica e Aplicada -, Vol. X, No. 2, pp. 63-75, 2004.

[Hair 2004c] Hair A., Implantation de la relation de visibilité, approche par le patron d'analyse rôle. Actes de conférence, 7ème colloque africain sur la recherche en informatique, CARI'04, pp. 181-189, Hammamet, Tunisie, 22-25 Novembre 2004.

[Hair 2005] Hair A., A UML extension for viewpoint-oriented modeling, IADIS International Conference IADIS Appied Compting, IADIS'05, Algarve, Portugal, 22-25 Février 2005.

[Hair 2006] Hair A., Vers un modèle de composants logiciels multi-vues ; RIST-Revue de l'Information Scientifique et Technique- Vol.16 n.2, pp. 121-139, 2006.

[Hair 2007] Hair A., Assemblage de Composants Logiciels multi-vues ; Workshop de Modélisation et Calcul, Arougou (Khénifra), Maroc, 24-26 Mai 2007.

[Hair 2012] Hair A., UML profiles for Viewpoint-oriented modeling ; Journal of Information and Organizational Sciences, JIOS, (Electronic version), Vol.29 n.2, pp. 13-24, 2012.

[Hair 2016] Hair A., A new software component approach ; International Journal of Computer Applications (IJCA) USA, Volume 135 – No.2 , Février *2016*.

[Harrison et al. 1993] Harrison W., Ossher H., Subject-oriented programming : a critique of pure objects, Proc. of OOPSLA'93, Washington D.C., 1993, pp. 411-428. 1993.

[Herzum et al. 2000] Herzum P., Sims O., Business component Factory, A Comprehensive Overview of Component-Based Development for the Enterprise, Wiley Computer Publishing, 2000.

[Hoff et al. 1997] Hoff A., Partovi H., Thai T., The Open Software Description Format (OSD), www.w3.org/TR/NOTE-OSD, 1997.

[Ivers et al. 2004] Ivers J., Clements P., Garlan D., Nord R., Schmerl B., Silva J.R.O., Documenting Component and Connector Views with UML 2.0. Technical report CMU/SEI-2004-TR-008, Avril 2004.

[Jacobson 1994] Jacobson I., « Basic Use Case Modeling », ROAD 1, 1994.

[Jacobson 2003] Jacobson I., Use Cases and Aspects - Working Seamlessly Together, Journal of Object Technology, vol. 2, num. 4, ETH Zurich, pp. 7-28, Juillet-Aôut 2003.

[Jacobson et al. 1999] JACOBSON I., BOOCH G. , RUMBAUGH J., The Unified Software Development Process ; Addison Wesley, Inc., 1999.

[Kandé et al. 2000] Kandé M. M., Strohmeier A., Towards a UML Profile for Software architecture description, Proceedings of UML'2000, Third International Conference, York, UK, 2-6 Octobre 2000.

[Kiczales et al. 1997] Kiczales G., Lampng J., Mendhekar A., Maeda C., Lopes C. V., Aspect-Oriented Programming, Proc. of the European Conference on Object-Oriented Programming (ECOOP). Finland. Springer-Verlag LNCS 1241, 1997.

[Kriouile 1995] Kriouile A., VBOOM, une méthode d'analyse et de conception par objet fondée sur les points de vue, Thèse d'état, faculté des sciences de Rabat, 1995.

[Kristensen 1996] Kristensen B. B., Object-oriented modeling with roles, In John Murphy and Brian Stone, editors, Proceedings of the 2nd International Conference on Object-Oriented Information Systems, , Springer-Verlag, pp. 57–71, 1996.

[Krumeich et al. 2014] Krumeich J., Werth D., Loos P., Conceiving a method for viewpoint-based modeling using recommender systems in a multiple-user environment – Conceptual approach and proof-of- concept. In Proceedings of the Twenty Second European Conference on Information Systems, 2014.

[Larman 2002] Larman C., UML et les designs patterns. Campus press, 2002.

[Larman 2002] Larman, C., Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and the Unified Process. Prentice Hall. 2002.

[Lau et al. 2005] Lau K. K., and Wang Z., A survey of software component models. Technical report, School of Computer Science, The University of Manchester. Pre-print CSPP-30 , Avril 2005.

[Mcllroy 1968] Mcllroy D. Mass-produced Software Components.1st International Conference on Software Engineering, Garmish Pattenkirschen, Germany, 1968.

[Marcaillou et al. 1994] Marcaillou S., Coulette B., Kriouile A., Visibility : A new relationship for complex system modelling, TOOLS USA'94, Santa Barbara, Prentice Hall,1-5 Aôut 1994.

[Marvie et al. 2001] Marvie R, Merle P, CORBA Component Model: Discussion and Use with Open CCM, rapport technique du Laboratoire d'Informatique Fondamentale de Lille Université des Sciences et Techniques de Lille, Lille, France, Janvier.2001.

[Marvie 2002] Marvie R., Séparation des préoccupations et méta-modélisation pour environnements de manipulation d'architectures logicielles à base de composants. PhD thesis, Université des Sciences et Techniques de Lille, LIFL, Villeneuve d'Ascq, Décembre 2002.

[Marzak 97] Marzak A., "Conception de VBTOOL, outil support de la méthode VBOOM, réalisation des fonctionnalités : Analyse et conception, Thèse pour l'obtention du diplôme de spécialité de 3ème cycle de l'université Mohamed V, 1997.

[Medvidovic et al. 2000] Medvidovic N., Taylor R. N. A Classification and Comparison Framework for Software Architecture Description Languages, IEEE Transactions on Software Engineering, vol. 26, n° 1, p. 70–93, Janvier 2000.

[Medvidovic et al. 2002] Medvidovic N., Rosenblum D.S., Redmiles D.F., Robbins J.E., Modeling Software Architectures in the Unified Modeling Language, ACM Transactions on Software Engineering and Methodology, Vol. 11, No .1, Janvier 2002.

[Mellor et al. 2004] Mellor S.J., Kendall S., Uhl A., Weise D., Model Driven Architecture, Addison-Wesley Pub Co, Mars 2004.

[Meyer 1992] Meyer B., Design by contract, IEEE Computer, Vol 25, No. 10, 1992.

[Microsoft 2005] Microsoft, COM, COM Fundamentals, http://msdn.microsoft.com/library/, 2005.

[Microsoft 2010] Microsoft, .NET, http://www.microsoft.com/net/, 2010.

[Miles et al. 2013] Miles R. , Hamilton K., Learning UML 2.0. O'Reilly Media, USA, 2013.

[Miller et al. 2003] Miller J. , Mukerji J., MDA Guide Version 1.0, OMG Document: omg/2003-05 01, http://www.omg.org/mda/mda_files/MDA_Guide_Version1-0.pdf, Mai 2003.

[Muller 97] Muller P.A., Modélisation objet avec UML ; Edition Eyrolles, 1997.

[Nasser 2005] Nasser M., Analyse/conception par points de vue : le profil VUML, Thèse de doctorat, Université de Toulouse, Septembre 2005.

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

[Objecteering 2015] Objecteering-site. http://www.objecteering.com, 2015.

[OMG 2002a] Object Management Group, CORBA Components Version 3.0 : An adopted Specification of the Object Management Group, Juin 2002.

[OMG 2002b] Object Management Group, CORBA 3.0 - OMG IDL Syntax and Semantics. Technical report, OMG, Juillet 2002.

[OMG 2002c] Object Management Group, CORBA Components , Version 3.0, Object Management Group, Juin 2002.

[OMG 2004] Object Management Group, Unified Modeling Language, Version 2.0 ; Object Management Group, http://www.omg.org, 2004.

[OMG 2007] Object Management Group, CORBA Component Model Specification OMG Available Specification Version 4.0, 2007.

[OSGI 2010] OSGI, OSGI Service Gateway Specification, Release 1.0, http://www.osgi.org, 2010.

[Oussalah 2005] Oussalah M., Ingénierie des composants : concepts, techniques et outils. Editions Vuibert, 2005.

[Pastor et al. 2013] Pastor O., Molina J. C., Model-Driven Architecture in Practice: A Software Production Environment Based on Conceptual Modeling. Springer, 2013.

[Pawlak et al. 2004] Pawlak R., Younessi H., On Getting Use Cases and Aspects to Work Together, Journal of Object Technology, vol. 3, num. 1, ETH Zurich, pages 15-26, Janvier- Février 2004.

[Peltier 2002] Peltier M., « Transformation entre un profil UML et un métamodèle MOF », *Revue l'Objet*, vol. 8, n°1-2/2002, LMO'2002, pp. 25-40. 2002.

[Platt 2002] Platt D. S., Introducing Microsoft .Net, Second Edition. Microsoft Press, Redmond, WA, USA, 2002.

[Rational 2015] Rational-site, http://www.rational.com, 2015.

[Rehioui et al. 2014a] Rehioui F., Hair A., A new approach of modeling based on software components ; 3th Edition Arougou Meeting of Analysis and Applications (AM2A'14), Arougou (Khénifra), Maroc, 22-24 Mai 2014.

[Rehioui et al. 2014b] Rehioui F., Hair A., Approche de modélisation basée sur les composants logiciels ; 1ière Edition des Journées Doctorales (JDoc'14), 20ème anniversaire de la Faculté des Sciences et Techniques sous le thème Recherche Scientifique : Innovation et Développement, Béni Mellal, Maroc, 10-11 Juin 2014.

[Rehioui et al. 2014c] Rehioui F., Hair A., Proposition d'une méthode d'ingénierie à base de composants ; 6ième édition des Journées Doctorales en technologies de l'information et de la Communication (JDTic' 2014), Rabat, Maroc, 19-20 Juin 2014.

[Rehioui et al. 2014d] Rehioui F., Hair A., Towards a modeling approach based on software components; International Journal of Computer Applications (IJCA) USA, Volume 98 – No 1, Décembre *2014*.

[Rehioui et al. 2015a] Rehioui F., Hair A., Un patron orienté point de vue d'assemblage des composants logiciels ; 2ème Edition des Journées Doctorales (JDoc'15), Béni Mellal, Maroc, 26-28 Mars 2015.

[Rehioui et al. 2015b] Rehioui F., Hair A., Towards a new approach combination management software components ; Second international Conference on Business Intelligence (CBI'15), Béni Mellal, Maroc, 23-25 Avril 2015.

[Rising 2000] Rizing L., The Pattern Almanac 2000. Software Pattern Series, Addison-Wesley, 2000.

[Roh et al. 2004] Roh S., Kim K. et Jeon T., Architecture Modeling Language based on UML2.0, Software Engineering Conference, APSEC'04, Proceedings of the 11th Asia-Pacific Software Engineering Conference, 2004.

[Rubinger et al. 2010] Rubinger A. L., Burke B., Enterprise JavaBeans 3.1, 6th Edition O'Reilly Media, USA, (2010).

[Schmidt 2006] Schmidt D. C.,Model-driven engineering. *IEEE Computer*, 39(2):25–31, 2006.

[SUN 2010] Sun Microsystems, JavaBeans 1.01 Specification, http://java.sun.com/beans, 2010.

[Szyperski 1998] Szyperski C., Component Software Beyond Object- Oriented Programming, Addison-Wesley, 1998.

[Traverson et al. 2002] Traverson B., Yahiaoui N., Connector for CORBA Components, In 8th International Conference on Object-Oriented Information Systems, Septembre 2002.

[Tarr et al. 1999] Tarr P. L., Ossher H., Harrison W. H., Sutton S. M. Jr., N Degrees of Separation: Multi-Dimensional Separation of Concerns, International Conference on Software Engineering, Workshop, pp. 107-119, 1999.

[Turner et al. 2003] Turner J., Bedell K., Stratuts. Edition CampusPress, Juin 2003.

[UML 2003] OMG, Unified Modeling Language: Superstructure, Technical Report Version 2.0, Object Management Group, 2003.

[Wetherbee et al. 2015] Wetherbee J., Rathod C., Kodal R., Zadrozny P., Beginning EJB 3: Java EE 7 Edition, Kindle Edition, France, 2015.

[Zarras et al. 2001] Zarras A., Issarny V., Kloukinas Ch., Nguyen V. K., Towards a Base UML Profile For Architecture Description, INRIA Rocquencourt, www.soi.city.ac.uk/~kloukin/pubs/icse01.pdf.2001.