Numéro d'ordre 3522

UNIVERSITÉ DES SCIENCES ET TECHNOLOGIES DE LILLE
ECOLE DOCTORALE SCIENCES POUR L'INGÉNIEUR DE LILLE

THÈSE

pour obtenir le titre de

DOCTEUR EN INFORMATIQUE

par

Emmanuel RENAUX

DÉFINITION D'UNE DÉMARCHE DE CONCEPTION DE SYSTÈMES À BASE DE COMPOSANTS

Soutenue publiquement le 7 décembre 2004 devant la commission d'examen :

Président	:	Pr Alain DERYCKE	TRIGONE, Université de Lille I
Rapporteurs	:	Pr Dominique RIEU	LSR - IMAG, Université de Grenoble
		Pr Eric LEFEBVRE	ETS, Montréal, Québec
Examinateurs	:	Pascal FLAMENT	Directeur de Projet, NORSYS
		Dr Sylvain LECOMTE	LAMIH, Université de Valenciennes
Directeur	:	Pr Jean-Marc GEIB	LIFL, Université de Lille I
Co-encadrant	:	Dr Olivier CARON	LIFL, Université de Lille I

Encore 99 km et je n'ai même pas mal aux jambes ;-)

Remerciements

Alain DERYCKE m'a fait l'honneur de présider le jury de la soutenance de ma thèse. Je le remercie sincèrement ainsi que Dominique RIEU et Eric LEFEBVRE pour avoir accepté de rapporter ma thèse et pour l'intérêt qu'ils ont porté à mon travail. Merci à Sylvain LECOMTE qui a bien voulu participer au jury de ma thèse ainsi que Pascal FLAMENT qui a été mon responsable industriel au sein de la société NORSYS. Il a été d'un très grand soutien aux moments où j'en avais besoin.

Olivier CARON a fait plus qu'encadrer ce travail. Son amitié, son soutien et ses qualités professionnelles ont permis l'achèvement de la thèse et m'ont appris le métier d'enseignant chercheur. Je ne le remercierai jamais assez pour cela. Jean-Marc GEIB m'a encadré durant mon DEA et ma thèse. Je le remercie sincérement pour la confiance qu'il a eu envers moi.

Je pense aussi aux nombreux voisins de bureaux et collègues que je ne pourrais sûrement pas citer sur cette page, je pense entre autre à Bernard, Laurent, Tof, JF, Philippe, Benoît, Gilles, Gaëtan, Rafi, Mathieu, Sylvain, Gauthier, Lionel, Olivier, Alexis, Jérôme, Romain, Christophe(s), Pierre, Fred, Aresky, Dolores, Nicolas, Missi, Bassem, Eric, Anne-Françoise et Renaud en ce qui concerne notre équipe. Je remercie particuliérement Laurence DUCHIEN pour tout ce qu'elle a fait depuis son arrivée au laboratoire. 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.

Ma thèse s'est déroulée en partie dans le cadre d'une thèse CIFRE au sein de la société NORSYS. Je tiens à remercier Sylvain BREUZARD qui m'a accueilli au sein de sa société ainsi que l'ensemble de ses employés qui pour beaucoup sont aussi des amis.

Je remercie chaleureusement mes parents et ma petite soeur sans qui tout cela n'aurait pas été possible, Edwige, mon épouse qui m'a toujours supporté, et ce quelques soient mes choix. Leur soutien et leurs encouragements ont été sans limites. Je remercie ma famille et amis pour qui je n'ai pas forcément toujours été présent au cours de ces années. A toutes les personnes qui ne sont pas citées sur cette page : sachez simplement que vous tenez une place dans mon coeur, et que je ne serais pas ce que je suis si je ne vous avez pas rencontré.

Table des matières

Table des figures

Chapitre 1

Introduction

1.1 Contexte

La conception d'applications d'entreprise a évolué pour garantir un maximum de qualité logicielle, mais s'est complexifiée en conséquence. Auparavant, client-serveur, les applications étaient monolithiques, développées et déployées sur un serveur, hébergeant une base de données. Aujourd'hui les applications tendent à être déployées de manière distribuée. La difficulté réside dans l'évolution du client-serveur à deux niveaux au client serveur à trois niveaux. La réflexion sur la conception d'applications réparties considère la séparation en trois niveaux : l'interface homme/machine, le traitement métier et le système de persistance. Cette problématique se pose dans les applications développées par des sociétés de services comme la société NORSYS[1], collaboratrice de ce travail.

Dans les premiers projets, autour des technologies Java et notamment de la plate-forme Java 2 Entreprise Edition [86], les investissements de la société dans ces projets innovants étaient risqués car les technologies et les méthodologies étaient à ce moment là émergentes. Les spécifications n'étaient pas forcément respectées de la même façon par les constructeurs d'outils. La recherche et développement dans ces domaines, aborde des problématiques à échelles multiples : de petits projets intranet aux projets globaux de refonte de Systèmes d'Information, notamment pour des grands pontes tels que le GAN Patrimoine, La Mondiale, les 3 Suisses, Voies Navigables de France.

Pour aborder de tels projets, le besoin était de monter en abstraction afin de garantir la pérennité des travaux d'analyse et conception des systèmes (cf figure 1.1). Le langage UML [90] [2] s'est imposé comme standard de modélisation des systèmes d'informations. A l'époque celui-ci était encore mal maîtrisé à la fois par les prestataires de services et par leurs clients mais semblait répondre aux besoins. Cependant, les modèles proposés par UML étaient mal utilisés. En effet, ils abordent des préoccupations différentes des modèles largement utilisés comme le MCD (Modèle Conceptuel de Données issu de la méthode MERISE [51]), focalisés sur la schématisation des données. Outre le problème de sémantique, les chefs de projets et analystes étaient confrontés au besoin de retrouver un aspect méthodologique éprouvé, comme la méthode Merise. La démarche était à reconstruire. Avec UML, est né le Processus Unifié [62], fédérant les bonnes pratiques du génie logiciel et utilisant ce langage standard de modélisation. Ma contribution fut la maîtrise du processus d'ingénierie Unified Process et une réflexion sur la façon d'adapter ce processus à la méthodologie et l'infrastructure de la société de services NORSYS. Dans des projets à large échelle, notamment refonte de SI, la notion de composant apparaît essentielle. Les clients souhaitent capitaliser leurs investissements dans le développement par la réalisation de composants métiers et techniques. La méthodologie UP n'intègre pas directement la notion de composant dans son approche. Or les applications sont développées sur des plates-formes dites à composants, telles que les Enterprise Java Beans [84] de SUN.

L'OMG[3], groupement d'industriels et d'organismes de recherche dans les domaines de l'orienté objet, s'active sur une formalisation d'une approche permettant de concevoir un système d'information de manière neutre vis-à-vis d'une technologie, modélisé dans un formalisme indépendant, puis de définir des mécanismes de projection de ce modèle vers des technologies. Cette approche est nommée Model Driven Architecture [33]. En faisant converger le Processus Unifié et l'architecture MDA, on peut convenir que l'analyse fournit un modèle PIM (Platform Independent Model), la conception

[1] *http ://www.norsys.fr*
[2] Unified Modeling Language : langage de modélisation unifié
[3] Object Management Group

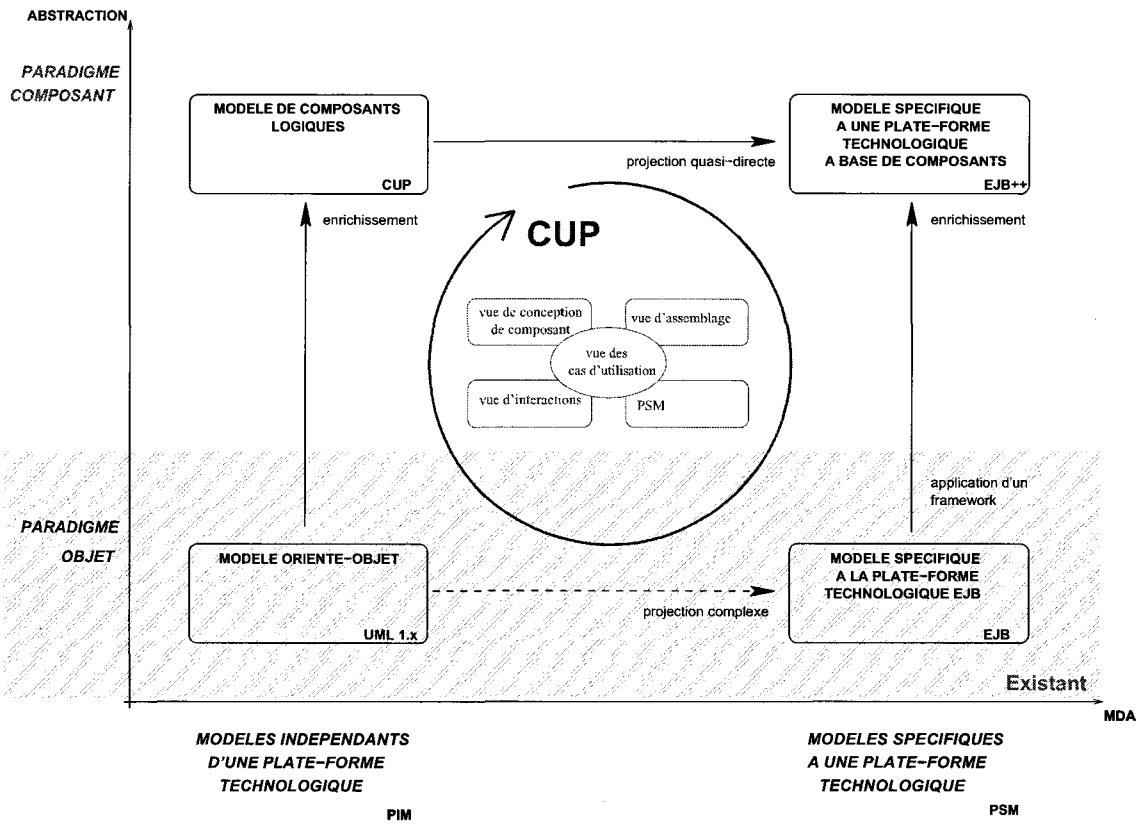


FIG. 1.1 – Schéma de l'approche à base de composants dirigée par les modèles

et l'implémentation fournissent un modèle PSM (Platform Specific Model), cependant, le Processus Unifié ne définit pas de mécanisme de projection.

La recherche, dans le domaine des standards d'architecture à base de composants offre une définition du composant plus large que celle des plates-formes Enterprise Java Beans : les spécifications CORBA Component Model [35] et sur les processus de production sous-jacents. La convergence des préoccupations entre les organismes de recherche, pour l'appréhension des nouvelles normes sur les composants, et les entreprises permet d'appliquer des idées plutôt fondamentales sur des points critiques de la construction d'applications à base de composants en entreprise : gestion de la correspondance objet/relationel, processus, profils UML, définition de composant, ... La problématique de cette thèse est le besoin d'une définition d'une véritable démarche à base de composants, facteur de qualité du logiciel, en réponse à la correspondance difficile entre un modèle PIM orienté objet exprimé en UML et un modèle PSM contenant la notion de composant. Celle-ci va de la spécification des besoins utilisateur, en passant par l'analyse/conception avec UML, jusqu'au déploiement des applications en appliquant des mécanismes MDA.

1.2 Contribution de la thèse

La complexité des systèmes d'information d'entreprise augmente. Aujourd'hui, le moyen reconnu pour gérer cette complexité est l'adoption d'une approche à base de composants. Actuellement, les méthodes ne tiennent pas compte de la notion de composant avant les problématiques d'implémentation et de déploiement. En effet, l'effort des travaux de recherche s'est d'abord porté sur les plates-formes technologiques, afin de répondre à un besoin de standard d'interopérabilité. Un fait établi d'expérience sur des projets d'ingénierie est que, plus tard on considère la notion de composant, plus chère est l'intégration de ce composant dans un système. Pour réduire ce coût, nous sommes convaincus que la notion de composant doit être prise en compte tout au long du processus de développement logiciel et qu'en conséquence, la notion de composant doit être identifiée suffisamment tôt. Comment réutiliser un composant, si on ne le détecte qu'à la fin du cycle de production d'application ?

Un travail de fond dans la société de services Norsys, par l'étude et la critique de sa méthodologie dans le cadre des projets "nouvelles technologies" et l'adaptation des méthodes classiques, pour la conception de systèmes d'informations, agrémentée des règles et bonnes pratiques apportées par le processus unifié, a permis de proposer une démarche à base de composants de modélisation (cf figure 1.1).
Les démarches classiques qui exploitent le paradigme objet offrent un découpage trop fin des systèmes et omettent la démarche de conception et de réutilisation de composants métiers, ou fonctionnels, transverses aux systèmes d'informations de ses clients. La contribution de cette thèse, de par l'implication de ces acteurs sur des projets de tailles réelles, notamment des rénovations de systèmes d'informations vers les architectures J2EE, a consisté à définir la notion de composant de modélisation, nommé **composant logique**. Cette notion est omniprésente dans les différentes vues qu'ont les différents acteurs d'un projet. Aussi, cette caractéristique permet de garantir la cohérence entre les différentes vues et offre un moyen de vérifier cette cohérence par l'application de contraintes définies sur chaque vue.
Dans le cadre d'une démarche de construction de systèmes d'information larges, la notion d'objet et les démarches d'analyse supportant UML ne sont donc plus suffisantes. La projection de modèles UML orientés objet vers des plates-formes à base de composants comme les Enterprise Java Beans est

complexe. De nombreux choix de conception qui demandent une expertise, sont à prendre. Un modèle de composants est proposé comme un enrichissement de modèles UML 1.4. Cet enrichissement ajoute des propriétés, des comportements, des paramètres garants de la généricité du composant logique et des contraintes. Enfin une démarche à base de composants logiques est proposée pour mener à bien ce type de projet. Cela passe par la définition de nouveaux acteurs du projet, ainsi que la définition d'un processus de production.
L'étude effectuée sur les architectures J2EE, incluant les spécifications des composants EJB, sur des projets d'entreprise, et sur une conception orientée objet grâce à la formalisation des modèles par les langages de modélisation standards tels qu'UML a montré la faiblesse du modèle pour définir au mieux les composants "métiers" et la démarche sous-jacente. Créées pour la conception d'applications objets distribuées au coeur de systèmes d'informations contenant des applications existantes, ces architectures logicielles ne permettent pas de définir de véritables composants réutilisables. Un modèle intermédiaire est proposé pour enrichir le concept de composant de la plate-forme EJB. Ce qui permet, dans une démarche MDA, de projeter le modèle de composants logiques vers la plate-forme EJB, via ce modèle intermédiaire qui rend la correspondance entre les éléments des différents modèles plus directe.

1.3 Plan du mémoire

Le chapitre 2 présente le contexte et le développement d'applications d'entreprise. Ce chapitre présente la problématique de conception d'applications d'entreprise et la représentation de systèmes à base de composants. Il montre l'insuffisance des formalismes, et démarches existantes pour permettre une projection directe vers le code des applications. Un véritable support d'une démarche à base de composants implique des activités nouvelles telles que l'identification de composants, la réutilisation de composants préfabriqués, la spécification de nouveaux composants en vue de leur publication "sur l'étagère".
Le Chapitre 3 amène la notion de composant logique en tenant compte comme unité de comportement, celui d'un ensemble d'objets qui œuvrent pour un but commun, plutôt que le comportement encapsulé dans un unique objet. Ce chapitre est inspiré de divers travaux de recherches sur les collaborations UML [3], les cadres de conception [89] et les cas d'utilisation [60, 61]. Ce chapitre contribue à l'enrichissement des modèles orientés objet existants.
Le chapitre 4 présente notre contribution, le processus **Component Unified Process** (CUP). Il présente notre proposition de représentation PIM d'un système spécifié par un méta-modèle de composants logiques qui enrichit le méta-modèle UML 1.4, et la démarche "unifiée" sous-jacente qui adapte le Processus Unifié à une approche à base de composants.
Le chapitre 5 présente une réalisation d'une approche MDA en proposant le prototype **CUPTool** de projection vers un modèle de plate-forme spécifique. Celui-ci valide l'application d'une démarche MDA par des mécanismes de projections automatisées du modèle PIM vers des technologies à base de composants. Ce chapitre montre les avantages d'utiliser un cadre de conception intermédiaire tranchant des choix complexes de projection d'une plate-forme indépendante d'une technologie vers un modèle spécifique de plate-forme technologique EJB et rendant cette projection quasi-directe.
Enfin, le chapitre 6 valide expérimentalement notre contribution sur des cas concrets de l'industrie du logiciel et montre par retour d'expérience l'intérêt d'appliquer une démarche d'ingénierie à base de modèles.
Finalement, nous concluons en résumant les principales contributions de cette thèse et ses perspectives.

Chapitre 2

Méthodologies de conception d'applications à base de composants

Aujourd'hui, l'ingénierie logicielle consiste à architecturer et implémenter une application à partir de ses spécifications. Les langages objets et les méthodologies orientées objet sont devenus des standards de l'ingénierie logicielle. Le formalisme Unified Modeling Language [90], spécifié par l'OMG[1] [39], est utilisé par les industriels qui disposent d'outils de modélisation UML intégrés aux outils de développement. UML n'est pas une méthodologie mais cette notation fournit un cadre pour l'analyse de systèmes d'information conçus par l'élaboration itérative de modèles.

La société d'ingénierie Norsys, développe des logiciels pour des clients dont le métier se situe dans les domaines de la banque, l'assurance, la santé et la grande distribution. Ces projets s'intègrent dans des systèmes d'information larges avec des problématiques métiers complexes. Des études montrent l'avantage de capitaliser l'expérience d'experts métiers. Le procédé est de réutiliser les développements issus de cette expertise, en les rendant réutilisables dans la plupart des cas, et dans une forme prévue pour être distribués et facilement branchés (pluggés) dans n'importe quel système d'information [2].

Dans le cadre de ma thèse, j'ai intégré des équipes de développement d'applications en apportant une expertise sur la modélisation objet avec UML. L'application de processus de modélisation des applications a soulevé des questions qui le remettent en cause. Doit-on obligatoirement analyser et concevoir un système, sachant que ce travail sera peut être remis en question par des choix d'implémentation ? Doit-on redévelopper, ou réutiliser ? Dans le cas où l'on choisit de réutiliser des composants préfabriqués, doit-on faire du reverse-engineering pour produire des documents d'analyse attenant à ces composants ?

Les démarches d'ingénierie orientées objets ont évolué avec l'ingénierie des modèles. L'initiative Model Driven Architecture de l'OMG préconise l'utilisation de modèles et la transformation de modèles pour mener à bien la construction de logiciel. La réutilisation de composants peut ainsi inclure la réutilisation de modèles de composants, et non seulement le code ou le binaire de ces composants.

[1] Object Management Group : Consortium de standardisation de l'orienté objet

Le marché n'est pas suffisamment approvisionné en de tels composants et il est très difficile d'identifier le composant qui convient. La convergence des travaux de recherche sur les technologies à base de composants et l'ingénierie des modèles contribue à l'émergence de **démarches d'ingénierie à base de composants**. La réutilisation et la publication d'un composant, sous-entend que l'on souhaite réutiliser et publier l'ensemble des modèles et documentation qui a été produit pour concevoir ce composant.

Ce chapitre présente les tenants et aboutissants d'une approche à base de composants. La section 2.1 présente les nouvelles architectures technologiques à base de composants. Cette section expose les caractéristiques qui nous semblent indispensables à la notion de composant. Puis, la section 2.2 présente l'initiative Model Driven Architecture de l'OMG. La section 2.3 présente les caractéristiques qui permettent de qualifier d'unifiée, une démarche en référence au Processus Unifié qui définit les meilleures pratiques de l'ingénierie logicielle. La section 2.4 définit la notion de démarche d'ingénierie à base de composants. La section 2.5 expose les moyens de représenter un système à base de composants et les moyens de spécifier au mieux un composant afin de garantir une meilleure réutilisation de celui-ci. Les démarches et représentations sont positionnées par rapport à leur adéquation avec la notion de composant que nous considérons, ainsi qu'une ingénierie des modèles. Enfin, la section 2.6 présente les travaux actuels sur l'aspectisation de la modélisation.

2.1 L'architecture à base de composants des applications d'entreprise

La notion de composant vise à identifier une partie d'un système, autonome et bien délimitée. Même si l'ambition de départ est la réutilisation de composants pour concevoir un système par assemblage de ceux-ci, la notion de composant mérite d'être considérée comme un véritable élément d'architecture pour l'analyse et la conception de système d'information. En effet, les efforts menés se sont surtout concentrés sur les plate-formes technologiques objet qui ont été revisitées pour devenir des plates-formes à base de composants (CCM, EJB, ...). Maintenant, il est devenu nécessaire d'adapter les approches de conception de tels systèmes, qui n'exploitaient que les caractéristiques de l'Orienté Objet.

2.1.1 Nouvelles architectures des applications d'entreprise

Les applications N-tiers

Les nouvelles technologies de l'information englobent une nouvelle architecture d'application. Les architectures 3-tiers structurent l'application en trois niveaux.

- **Le niveau Présentation** présente les informations de l'application ou du système à leurs utilisateurs. Ce tier ne contient pas de traitement complexe, il se préoccupe uniquement du formatage des données pour leur présentation ainsi que de la saisie d'informations.

- **Le niveau Métier** gère les traitements métiers qui peuvent être des règles de gestion, des services, ... Cette couche correspond aux résultats des travaux d'analyse qui à partir de scénarii définissent des services métiers. Ce niveau contient les "objets métiers" qui implémentent les "services métiers".
- **Le niveau Données** gère l'ensemble des sources d'information, c'est-à-dire la persistance des données de l'application, et l'accès à d'autres applications.

Ces architectures sont développées de manière à séparer les trois niveaux, afin de réduire l'impact de modification d'un niveau sur un autre.

Le patron Modèle-Vue-Contrôleur

Ce patron de conception organise le niveau présentation. Le modèle MVC a comme principe de séparer les objets de leurs représentations (cf figure 2.1).

- La partie **Modèle** correspond à l'implémentation des traitements métiers et la gestion des données. Elle garantit l'intégrité transactionnelle, permet l'accès rapide aux données applicatives, gère le workflow applicatif et permet l'intégration de systèmes informatiques déjà existants dans l'entreprise.
- La partie **Vue** correspond au formatage des résultats des traitements et la construction dynamique des pages de présentation. La vue est une représentation d'une partie du modèle métier à un instant donné.
- Le **Contrôleur** reçoit les requêtes des utilisateurs, appelle les traitements métiers correspondants et génère la vue appropriée pour présenter le résultat. Il synchronise les représentations d'un même

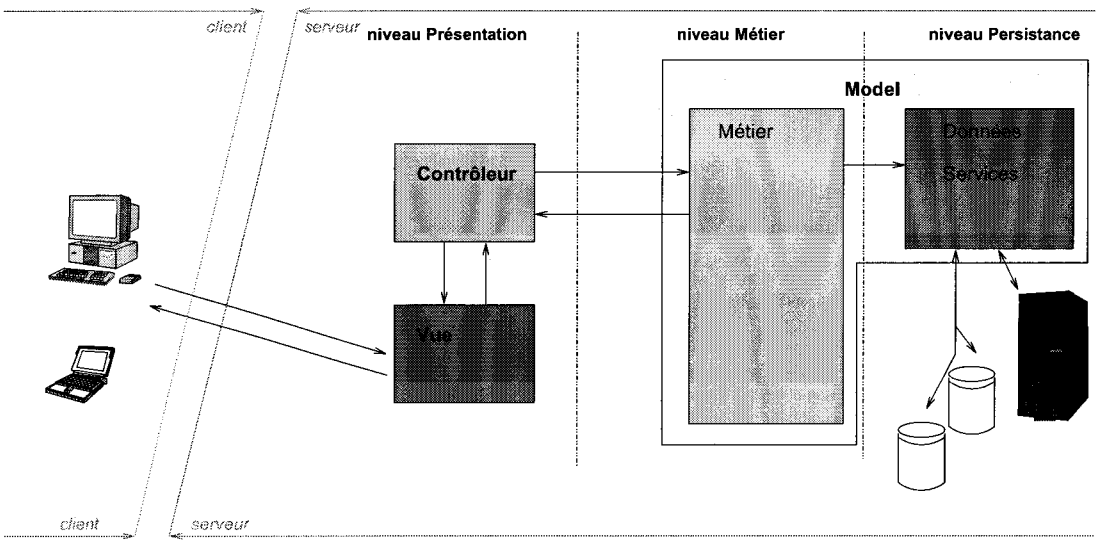


FIG. 2.1 – Patron de conception Modèle-Vue-Contrôleur

objet lorsque son état est modifié par les services métiers ou les interactions utilisateurs. Le contrôleur valide les paramètres d'une requête, vérifie que l'utilisateur est habilité aux traitements qu'il réclame, invoque les services nécessaires pour répondre à une requête du client.

Il existe la notion de composant dans chacun des niveaux présentés ici. Les composants d'interfaces homme-machine, comme les Java Server Pages, un bouton sur une page HTML, ..., par exemple. Nous ne considérons pas, dans ce travail, ces types de composants. Nous nous intéressons aux composants dits "métiers" définis plus loin dans ce document.

2.1.2 Caractéristiques des plate-formes technologiques à base de composants

Dans le domaine des applications distribuées à base de composants logiciels d'entreprise, il existe aujourd'hui trois plate-formes concurrentielles : les *Enterprise Java Beans* (EJB) de Sun [84], .NET de Microsoft et enfin le modèle de composants *CORBA Component Model* (CCM) de l'OMG [35]. Ces alternatives sont issues de travaux conduits pour faire évoluer des intergiciels (Middleware) orientés objet respectivement Java, COM/DCOM et CORBA 2. Analysons les propositions de Sun et de l'OMG afin de caractériser la notion de composant. .Net n'est pas abordé dans ce mémoire, car les développements sur cette plate-forme sont encore peu nombreux aujourd'hui.

La plate-forme Enterprise Java Beans

Les Enterprise Java Beans (cf. figure 2.2), spécifiés par Sun, sont pourvus de deux types d'interfaces.

1. Les interfaces maison (**Home**) permettent de gérer le cycle de vie du composant. Elles permettent, en outre, la création, la recherche et la destruction d'une instance de composant.
2. Les interfaces de services (**Remote**), exposent les méthodes métier et accesseurs et permettent au composant de rendre des services.

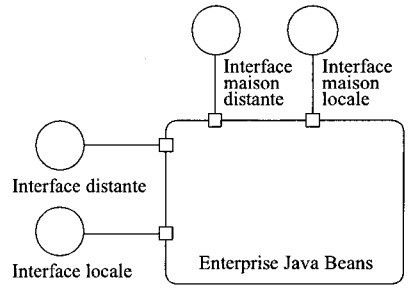


FIG. 2.2 – Modèle abstrait EJB

Ces interfaces peuvent être locales ou distantes. Implémenté par de nombreux industriels, ce choix technologique d'implémentation a pris le monopole des applications de gestion d'entreprise. Cette plate-forme sera détaillée plus précisément dans le chapitre 5 qui présentera un prototype d'implémentation de notre travail sur cette plate-forme. Le principal atout mis en avant est la simplicité de développement de tels composants grâce au concept de conteneur d'EJB, qui est un environnement de déploiement et d'exécution permettant de gèrer les services non fonctionnels, c'est-à-dire, autres que métiers, comme la persistance, la transaction et la gestion du cycle de vie. Ainsi, le développeur d'EJB peut se concentrer sur le développement du métier. Ce modèle est donc simple, et permet le développement rapide d'applications d'entreprise. Parmi les critiques, on peut citer :

- le nombre limité de services non fonctionnels implémentés par les constructeurs de conteneurs d'EJB ;
- l'impossibilité d'ajouter d'autres services qui ne sont pas pris en compte ;
- le manque de maturité de certains services, comme par exemple, la persistance qui ne se montrent pas assez aboutis, notamment dans la correspondance des informations d'une application EJB et celles stockées dans une base de données relationnelle déjà existante et non modifiable. Les développeurs préfèrent, sur des systèmes d'informations larges, développer eux-mêmes la couche de persistance ;
- enfin, l'aspect mono-langage des EJB, développés exclusivement en Java.

Le modèle CORBA Component Model

Les composants du modèle de composants CORBA (CCM) (cf. figure 2.3) résultent d'un consensus entre industriels et académiques. Les spécifications proposent un modèle de composants logiciels distribués et hétérogènes. Multi-langages et multi-systèmes, ce modèle n'a aucune limitation dans son utilisation, si ce n'est qu'il reste complexe car justement il tente de combler toutes les lacunes des autres propositions.

Le modèle repose sur le principe de développement des composants CCM en implémentant uniquement le fonctionnel du composant, et en décrivant les services non fonctionnels. Ce modèle est le plus proche des diverses définitions que l'on peut donner aujourd'hui de la notion de composant logiciel. Le modèle abstrait CCM simplifie cette définition par un ensemble de ports, requis et fournis, synchrones et asynchrones. Une interface de référence permettant de naviguer d'une interface à l'autre. Les services non fonctionnels sont gérés comme dans la technologie EJB par des conteneurs, environnements de déploiement et d'exécution de composants CCM.

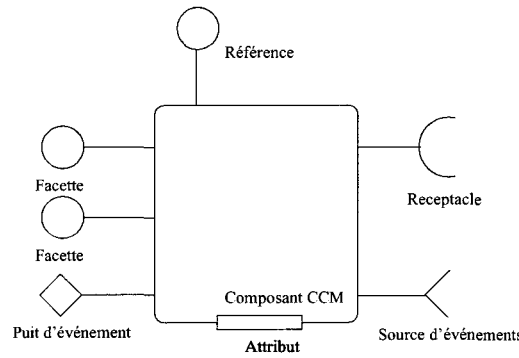


FIG. 2.3 – Modèle abstrait CCM

Les caractéristiques d'un modèle spécifique à base de composants

Pour synthétiser, et pour identifier les caractéristiques des différentes implémentations possibles de composants métiers, on peut résumer la notion de composant par la liste suivante :

- Les **ports fournis** d'un composant permettent de catégoriser les comportements offerts par ce composant *(CCM, .Net, une seule interface fournie pour EJB)*.
- Les **ports requis** d'un composant permettent de préciser les comportements externes attendus par ce composant pour réaliser ses traitements. Ces comportements sont servis par des interfaces fournies d'autres composants *(CCM, .Net)*.
- Le **connecteur** est un mécanisme permettant de connecter des composants entre eux par leurs interfaces fournies et requises que l'on appelle, dans le contexte d'une connexion, **ports** des composants *(notion de port CCM, mais pas de connecteur)*.
- Le **mode de communication synchrone ou asynchrone** correspond respectivement à deux types de communication : l'invocation de méthodes, l'envoi et la réception d'événements. *(CCM, EJB, .Net)*
- Le composant **composite** est une notion propre au paradigme composant, elle permet de concevoir des composants composés de composants existants, manipulés comme des composants basiques.
- La **séparation du fonctionnel et du non fonctionnel** est une caractéristique offerte par la plupart des environnements qui prennent en charge tout ce qui n'est pas code métier *(CCM, EJB)*.

Dans le cadre de notre étude, nous considérons la plate-forme EJB comme un bon choix pour la conception d'application d'entreprise, étant donné le marché des applications de gestion des domaines des banques, assurance et santé. Par contre, nous ne nous limitons pas à la définition des composants EJB, tenant compte de l'ensemble des caractéristiques ci-dessus. Cependant, nous considérerons uniquement le mode de communication synchrone. L'objectif est de concevoir une démarche à base de composants qui ne se focalise pas sur une seule plate-forme technologique cible, et qui peut être exploitée quelque soit le choix technologique.

Nous avons caractérisé la notion de composant, qui semble promettre une meilleure qualité de logiciel, notamment en ce qui concerne le critère de réutilisation. La section suivante fait l'état de l'art des méthodologies actuelles permettant la construction d'applications à base de composants et montre la faiblesse de ces démarches dans l'objectif de concevoir des applications d'entreprise à base de composants.

2.2 L'ingénierie dirigée par les modèles

Les architectures distribuées sont très évolutives. Les technologies évoluent rapidement, et il est souvent nécessaire de recommencer l'analyse et la conception d'un système pour le re-développer. L'ingénierie des modèles est la réponse actuelle à cette problématique. Elle consiste à modéliser un système d'une façon neutre vis-à-vis d'une technologie, et d'exploiter ce modèle à chaque changement technologique. Le développement conduit par les modèles est standardisé dans la spécification de l'OMG [39] : **Model Driven Architecture** [43, 33]. Reconnue comme la meilleure façon de concevoir et de prendre des décisions d'architecture avant tout effort coûteux d'implémentation, la spécification MDA identifie deux types de modèles :

Platform Independent Models (PIM). Un modèle PIM ne contient aucune préoccupation technologique.

Platform Specific Models (PSM). Un modèle PSM est dépendant des caractéristiques d'une plate-forme technologique. Souvent il décrit du code exécutable ou des préoccupations de distribution.

Un processus d'ingénierie qui applique l'approche MDA peut être décomposé en quatre étapes [30] (voir figure 2.4) :

1. *La réalisation d'un modèle PIM initial.* Ce modèle peut être un support aux activités d'analyse et de conception. Il permet essentiellement de modéliser des entités métiers et leurs relations, ainsi que des processus métiers.

2. *Le raffinement du modèle initial.* Des raffinements successifs peuvent intégrer des informations supplémentaires sans détails technologiques spécifiques à une plate-forme. Les informations ajoutées peuvent concerner des problématiques non-fonctionnelles, telles que des aspects de persistance, sécurité, ...Ce type d'information peut aider à projeter ce modèle vers un modèle PSM.

3. *La projection vers un modèle PSM.* Le modèle PSM se concentre sur des aspects relatifs à une plate-forme technologique, souvent dans les dernières phases du cycle de développement d'un logiciel. Ce modèle peut être décrit soit par un nouveau méta-modèle, soit en utilisant des profils UML [79]. Le concept de profil permet d'étendre le méta-modèle UML standard, selon des règles, en utilisant des stéréotypes, valeurs marquées et contraintes. Les profils UML de plates-formes technologiques existent déjà, par exemple le profil EJB [83] ainsi que le profil CORBA [38]. Cette solution permet d'utiliser des outils du marché offrant des possibilités d'extension du méta-modèle UML grâce à la notion de profil.
   Cette étape peut être automatisée grâce à la méta-modélisation [25]. La transformation de modèles [80] est une opération qui consiste à traduire un modèle issu d'un méta-modèle en un modèle issu d'un autre méta-modèle. En conception d'application, un modèle d'analyse suffisamment riche est projeté vers un modèle de conception ou implémentation.

4. *Le raffinement du modèle PSM.* Un modèle PSM peut être enrichi par l'ajout d'informations comme à l'étape 2. Le but est d'obtenir un système exécutable et d'adapter de manière adéquate la projection PIM vers PSM.

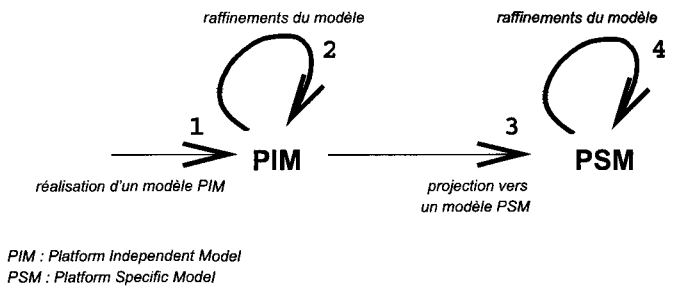


FIG. 2.4 – Les étapes d'un processus MDA

Une des principales caractéristiques de l'architecture de l'OMG est la transformation d'un modèle PIM vers un ou plusieurs modèles PSM. Le modèle abstrait permet d'être indépendant d'une technologie et de choisir le plus tard possible la technologie la mieux adaptée dans une cycle de production logicielle. Si possible, la projection PIM vers PSM peut être automatisée pour fournir une véritable chaîne de production d'applications. Le coût du changement technologique est réduit en capitalisant sur les premières étapes du cycle de production qui restent alors valides.

L'OMG recommande l'usage d'UML comme formalisme pour spécifier les divers modèles. Le standard UML 1.4 est actuellement utilisé pour décrire les modèles d'application essentiellement dans les activités d'analyse et de conception et pas forcément dans tout le cycle de développement logiciel. Le futur standard UML 2.0 vise à élargir le domaine d'utilisation d'UML, notamment en introduisant un modèle enrichi de composant logique (exemple : composant métier, composant de processus). Des modèles comme le profil UML pour Enterprise Distributed Object Computing [41] offrent un autre formalisme à base de composants issu du modèle de référence RM-ODP [81].

L'approche MDA nous semble pérenne. Elle promet que les investissements dans l'analyse de système d'information seront amortis par de multiples projections dans des technologies différentes. Cette approche sera choisie dans la suite du mémoire afin d'argumenter des choix de conception et de structurer notre architecture.

2.3 Les principes des démarches d'ingénierie actuelles

Un processus d'ingénierie est, dans sa définition la plus simpliste, la réalisation d'activités par des rôles. Il décrit les livrables nécessaires et produits pour chaque activité :. documents, diagrammes UML, code source, librairies,...(cf. figure 2.5 extrait du Software Process Engeering Metamodel [37] de l'OMG). Par exemple, le "spécifieur" de cas d'utilisation est responsable de la description textuelle d'un cas d'utilisation. Cette activité nécessite l'identification des acteurs et des fonctionnalités offertes aux acteurs par le système et fournit des documents de spécification de l'application à développer.

Il existe plusieurs processus orientés objet exploités dans l'industrie logicielle : le Processus Unifié [62], le Two Track Unified Process [82], eXtreme Programming [54], ... Nous allons tenter d'extraire les caractéristiques qui semblent importantes pour produire des logiciels de qualité, avec une architecture à base de composants.

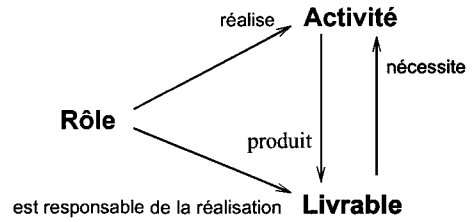


FIG. 2.5 – Modèle du processus d'ingénierie

Découpage en workflows.
Les méthodes définissent des ensembles d'activités qui doivent être réalisées. Chacune décrit ce qui doit être fait, par qui. Le Processus Unifié introduit le concept de workflow pour décrire une séquence d'activités menant à un résultat observable. La plupart des méthodes actuelles définissent un ensemble de workflows, qui doivent être accomplis dans chaque itération et qui raffinent et enrichissent chaque vue sur l'architecture logicielle. Ces workflows ne doivent pas être pris en charge successivement. Cela serait trop restrictif et conduirait à un rejet par les équipes d'ingénierie. En fait, l'introduction d'un nouvel élément à un moment donné peut se faire dans n'importe quel modèle du système. Cependant, cela stimule la réalisation d'autres activités dans d'autres workflows et provoque le raffinement d'autres modèles du système.

Un processus itératif.
Les approches itératives réduisent la complexité en réalisant les étapes de la méthode par des raffinements successifs et des enrichissements. Ainsi, les points les plus difficiles sont abordés en priorité, pour réduire les risques, et apportent rapidement des éléments de validation aux utilisateurs finaux. La production d'exécutables jalonne et valide chaque itération. Ceci réduit les coûts de retour arrière rencontrés couramment dans une approche en cascade traditionnelle.

Un découpage temporel en phase.
Chaque méthode découpe le processus en phases dépendantes les unes des autres. Chaque phase met l'accent sur des préoccupations particulières de l'ingénierie logicielle. Par exemple, les premières phases se concentrent essentiellement sur les besoins des utilisateurs finaux et des clients, et les dernières phases, sont plus concernées par des problématiques de déploiement et d'accompagnement de l'utilisateur final.

Les caractéristiques des bonnes démarches d'ingénierie sont les caractéristiques du Processus Unifié, conçu justement pour répertorier l'ensemble des bonnes pratiques de l'ingénierie logicielle.

Le Processus Unifié

Le Processus Unifié [63] décrit une chaîne de production de systèmes informatiques. Cette démarche fait office de référence, car elle regroupe toutes les bonnes pratiques de l'ingénierie du logiciel et la plupart des démarches mises en place en entreprise tentent de respecter le Processus Unifié. Le Processus Unifié découpe le cycle de vie d'un projet d'ingénierie en quatre phases effectuées l'une après l'autre après validation de leurs jalons respectifs. Chaque phase est découpée en un certain nombre d'itérations. Ce nombre dépend de la taille et de la complexité du projet. Le jalon d'une itération est, en outre, la production et le test d'un livrable validé par le client et l'utilisateur final.

Les phases d'un cycle de production sont :

1. **la phase de lancement.** Les activités dominantes de cette phase permettent d'appréhender le contexte métier, d'estimer le retour sur investissement par l'ébauche du périmètre du projet à partir des principaux besoins.
2. **la phase d'élaboration.** L'architecture logicielle est élaborée, notamment par les activités d'analyse et de conception objet, tout en continuant à spécifier des besoins secondaires.
3. **la phase de construction.** Elle fournit une première version du logiciel répondant à la quasi-totalité des besoins utilisateur et client.
4. **la phase de transition.** Le système est finalisé et l'utilisateur final est accompagné dans son utilisation grâce aux activités de packaging, de documentation, d'organisation de formations.

Un intérêt du Processus Unifié est de concevoir un système informatique comme réponse aux besoins exprimés par les utilisateurs finaux et le client. Pour aider à mieux appréhender l'infrastructure et le fonctionnement d'un organisme, une analyse métier permet d'en connaître les différents processus métier. La recherche des limites fonctionnelles du système se base sur le recueil et l'expression des besoins et sur l'analyse métier. Les frontières du système sont décrites dans le **modèle de cas d'utilisation**. Dans celui-ci, les acteurs catégorisent les différents rôles des membres de l'organisme étudié et les cas d'utilisation expriment leurs besoins. L'ensemble des cas d'utilisation, interactions entre les acteurs et le système représente ce que le système doit faire. Un même cas d'utilisation peut s'exécuter suivant différents scénarii. Un scénario est une suite d'étapes menant à un comportement observable et attendu du système. Le scénario nominal est le chemin d'exécution type du cas d'utilisation. Les autres sont des scénarii alternatifs au scénario nominal. Enfin, les exceptions interrompent l'exécution du cas d'utilisation sans que le comportement attendu soit atteint. (fig. 2.6)

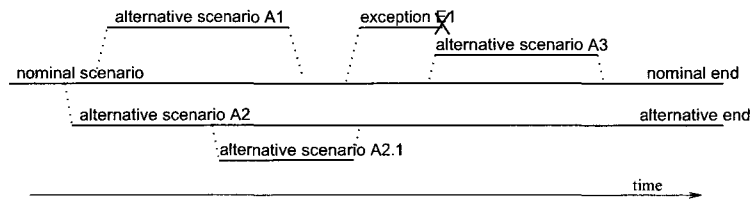


FIG. 2.6 – Vue schématique des scénarii d'un cas d'utilisation

L'architecture interne du système est exprimée dans le **modèle d'analyse**, qui correspond, dans une terminologie MDA à un modèle PIM. Les activités nécessaires pour construire le modèle d'analyse doit permettre de répondre aux sollicitations de l'utilisateur final décrites dans les cas d'utilisation. Pour cela, l'activité d'analyse objet transforme les concepts mis en jeu dans les cas d'utilisation, en objets métiers. A la différence d'une démarche de type client-serveur, l'analyse objet doit tenir compte à la fois des données et des traitements pris en charge par une même entité (objet métier). Le modèle d'analyse permet de décrire le coeur de l'architecture du système grâce à l'ensemble des objets métiers et leurs relations, indépendamment d'une technologie.

L'activité de conception permet ensuite de construire le **modèle de conception**. A partir du modèle d'analyse, elle définit concrètement comment doit être implémenté le système sur une plate-forme technique. Les développeurs exploitent ce modèle pour produire le **modèle d'implémentation** correspondant au code livrable de l'application. Les **modèles de composants et de déploiement** modélisent les composantes du système et leur intégration sur la plate-forme physique.

Le processus unifié est itératif et incrémental. Chacune des quatre phases est découpée en un certain nombre d'itérations. Une itération consiste en des enrichissements et raffinements successifs de chaque modèle présenté. Notamment, elle produit un exécutable validé par la maîtrise d'oeuvre et la maîtrise d'ouvrage, afin de montrer l'état d'avancement d'un projet.

Pour synthétiser, le processus unifié est une démarche méthodologique générique pour concevoir un système dans des technologies orientées objet. Il s'appuie sur le langage de modélisation standard UML. Les quatre phases successives, avec leurs préoccupations dominantes respectives, enrichissent et raffinent ses modèles. Les avantages de la démarche sont nombreux. L'architecture est élaborée pour répondre aux besoins de l'utilisateur final. La production d'un exécutable non jetable à chaque itération réduit le risque coûteux de retour arrière des démarches traditionnelles en cascade et permet à toutes les parties prenantes de valider l'avancement du projet, quelque soit leur niveau de connaissance dans ces technologies.

Critique de la démarche Processus Unifié

L'inconvénient majeur de la démarche est de fournir une décomposition trop fine du système et ainsi de compliquer l'intégration de composants préfabriqués. En effet, le découpage en cas d'utilisation permet d'avoir une bonne vision des fonctionnalités qui doivent être offertes par le système. Cependant, ce découpage ne doit pas être conservé une fois la phase de lancement terminée. Cela pour éviter une décomposition trop fonctionnelle. Un regroupement des objets s'opère alors pour organiser l'architecture interne du système. Le but étant d'en réduire la complexité en le compartimentant. Or, le processus, trop générique, n'offre pas de règle pour cela. Le composant est souvent intégré tardivement au moment de la conception. Les analystes ont produit des spécifications générales et une architecture objet qui n'offre pas forcément le découpage ad-hoc pour s'intégrer dans la structure des composants. L'intervention onéreuse d'experts est alors nécessaire comme par exemple l'architecte qui doit trouver les éléments architecturaux utilisables à partir de composants.Il nous semble important de sur-spécifier , afin d'offrir un cadre de conception plus structuré.

Une fois l'analyse et la conception finalisées, l'intégration d'un composant est souvent pressentie comme un surcoût de production, notamment à cause du maintien de la cohérence des différents livrables (spécifications, modèles, ...) fournis préalablement. Les spécifications propres du composant doivent être intégrées à celle du système car la traçabilité doit être assurée, et lors de l'évolution des besoins, l'impact sur le composant doit être gérée. De manière générale, l'intégration du composant consiste à intégrer ses propres modèles à ceux du système en développement. Ainsi, la charge d'intégration peut être distribuée à chaque rôle du processus.

2.4 Les démarches d'ingénierie à base de composants existantes

2.4.1 Catalysis

Catalysis [56] est une démarche de conception d'applications à base de composants, flexible et capable de s'adapter à tout type de projet d'ingénierie. Basée sur la notation UML 1.x, Catalysis étend le formalisme UML par des extensions pour représenter les composants au niveau logique, plutôt que la notion standard de composant UML qui est de niveau physique. L'approche ne fournit pas de mécanisme d'identification de composant et peut conduire comme le RUP à une décomposition trop fine du système. Par contre, l'usage systématique du langage de contrainte Object Constraint

Language [36] garantit une rigueur dans la spécification des composants. Les apports de Catalysis pour les démarches à base de composants sont utilisés et cités dans le chapitre suivant, notamment la définition et la réutilisation de frameworks. La démarche, reconnue complexe, n'offre pas de véritable cadre méthodologique pour tout le cycle de développement.

2.4.2 UML Component

Cheesman et Daniels proposent dans cet ouvrage [55] une méthode de spécification de composants intégrée dans une démarche d'ingénierie des systèmes d'information sans aborder tous les domaines. Le livre se concentre en effet sur la partie spécification d'un système à base de composants. Après une définition de la notion de composant, centrée sur la notion d'interface, ils présentent le formalisme utilisé pour spécifier un système à base de composants. Ce formalisme est UML 1.x avec l'usage de stéréotypes pour spécialiser la sémantique du langage graphique standard à l'usage présenté dans le livre. Issu de divers travaux sur les méthodologies orientées objets et orientées composants, l'ouvrage aborde la notion de modèle sans pour autant expliciter les interrelations entre les différents modèles présentés.

La démarche UML Component spécialise la démarche Catalysis [56] 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, UMLComponent 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 UMLComponent n'est pas MDA dans le sens où il ne donne pas une définition abstraite 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 UMLComponent est de limiter les problématiques d'intégration de préoccupations techniques dans le résultat d'une réflexion neutre.

> *Le développement à base de composants est différent des autres approches dans sa séparation des spécifications de composant de l'implémentation et la division de ses spécifications en interfaces.*

On déduit de cette définition que la connexion entre les composants est 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 de composant et séparer les spécifications de composant de l'implémentation. Par contre, l'objectif de UMLComponent 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.

UMLComponent définit des stéréotypes pour spécialiser UML 1.x 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.

2.5 La représentation de systèmes à base de composants

Il n'existe pas de standard pour modéliser un système à base de composants. Ce type de logiciel s'avère difficile à documenter de manière complète et consistante. La réutilisabilité s'en trouve dès lors réduite.

UML est un standard de modélisation de logiciels. Il est donc intéressant d'essayer de modéliser des systèmes à base de composants avec UML. Il y a deux types de documentation. La première est une documentation support de la conception et l'implémentation d'un système. La deuxième est la production de documentation en vue d'illustrer l'utilisation, les dépendances, ... d'un composant en vue de sa réutilisation.

L'ingénierie des systèmes à base de composants est censée augmenter la qualité, notamment le critère de réutilisation. Meyer [72] constate que ce critère a plutôt tendance à diminuer, par le fait qu'un composant approprié est difficile à trouver et qu'il est difficile d'évaluer son adéquation. Une fois trouvé, la documentation est souvent insuffisante ou inadéquate pour permettre au développeur d'intégrer les composants au reste du système. L'implémentation d'un composant diffère souvent de son usage spécifié, car il a été conçu dans un contexte différent. Enfin, l'effort réalisé au niveau implémentation est fait avec des technologies changeantes et des standard émergeants qui ne rencontrent pas forcément de réel succès.

Orientation

Exprimer la conception de composant d'une façon neutre vis-à-vis de l'implémentation dans le but de spécifier la réalisation complète d'un composant de manière à générer automatiquement une implémentation. L'initiative de l'OMG, **Model Driven Architecture**, tente d'imposer cette approche afin de bénéficier de spécifications de composants enfin réutilisables. Cette conception, neutre, peut servir de base à une documentation lors de la publication d'un composant, en vue de son exploitation dans une librairie. UML, standard de modélisation, pourrait être le formalisme neutre permettant d'atteindre l'objectif fixé par l'OMG.

2.5.1 Utilisation d'UML pour concevoir des composants

L'idée est de concevoir un composant avec UML dans l'attente d'un standard de modélisation à base de composants. Un composant peut être vu selon deux aspects, logique et physique. L'ensemble, vue logique et vue physique du composant en donne une définition complète.

La vue logique d'un composant

La vue logique de composant correspond à une abstraction logique, ses relations avec d'autres éléments logiciels et ses responsabilités.

L'abstraction logique peut être représentée par la notion de sous-système UML (cf figure 2.7).

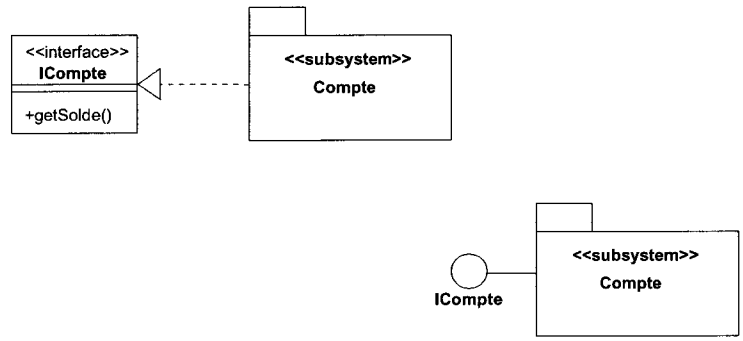


FIG. 2.7 – Le sous-système pour représenter un composant

Un **sous-système** est un paquetage stéréotypé. En cela, il peut contenir d'autres éléments UML. Ces éléments réalisent les opérations des spécifications du sous-système et sont encapsulés par celui-ci. Le contenu du sous-système est isolé du client qui ne voit pas le détail de l'implémentation du composant. Il y a séparation des spécifications du composant et de son implémentation. Le paradigme composant hérite cela du paradigme objet et de ses critères de **modularité** et d'**encapsulation**.

Les spécifications d'un composant sont exprimées par une interface (cf figure 2.7). Une interface peut être réalisée par plusieurs sous-systèmes et un sous-système peut réaliser plusieurs interfaces. Ainsi, une interface ne peut pas être contenue dans un sous-système. Un système qui réalise la même interface qu'un autre peut le remplacer.

L'intérieur d'un composant peut être représenté par des **collaborations**[2]. La notion de collaboration est l'une des plus importantes d'UML. Pourtant, c'est l'élément de modélisation dont on se sert le moins pratiquement. La collaboration permet de documenter la solution à un problème, par exemple, les spécifications d'un composant. En effet, la collaboration regroupe un ensemble de diagrammes UML et ainsi offre plusieurs vues :

- Les relations entre les éléments du sous-système et d'autres sous-systèmes sont représentées dans un diagramme de classes (cf figure 2.8).
- Les comportements les plus importants d'un sous-système sont précisés dans des diagrammes d'états et d'activités.
- Les détails d'implémentation des interfaces principales sont représentés par des interactions (cf figure 2.9).

[2]Nous revenons sur cette notion plus longuement dans le chapitre suivant

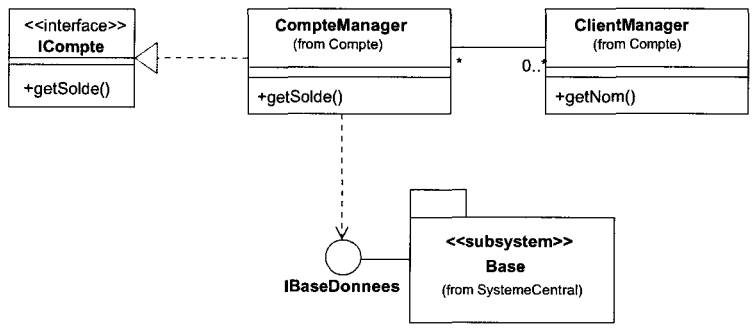


FIG. 2.8 – Vue statique d'une collaboration

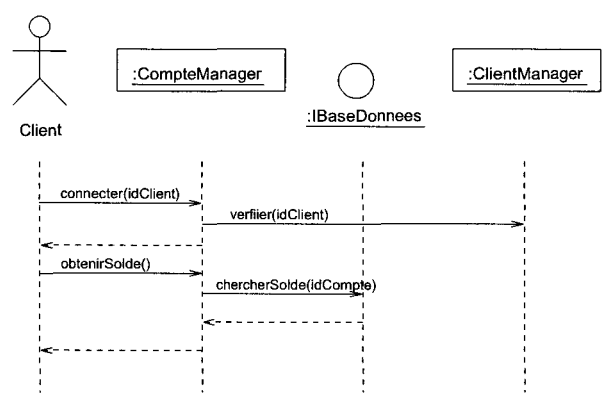


FIG. 2.9 – Vue dynamique d'une collaboration

La vue physique d'un composant

La représentation physique d'un composant se fait dans un environnement donné. L'élément UML **Component** permet de représenter l'aspect physique d'un composant en UML standard. Le mapping entre un sous-système et un composant UML n'est pas évident. Un composant UML est la réalisation physique d'un ou plusieurs sous-systèmes (cf figure 2.10). La projection dépend du langage d'implémentation et de l'environnement de développement, du modèle de composant choisi, de la manière de déployer le composant et du code source de celui-ci. Est-il développé, acheté ? ...

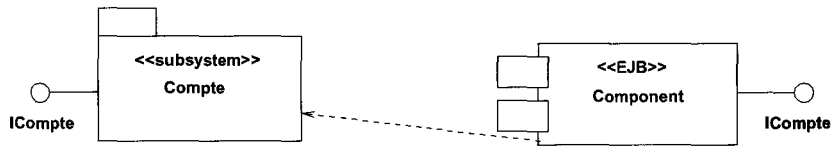


FIG. 2.10 – La projection vue logique et vue physique de composant UML

La représentation physique documente aussi la dépendance avec d'autres composants, notamment avec les composants de l'infrastructure. Le guide utilisateur d'UML [3] préconise de séparer l'aspect implémentation de l'aspect déploiement. Cependant, qu'en est-il du passage de la vue logique à la vue physique ? Qu'est-ce qui guide le développeur dans le choix des éléments d'une vue logique, pour concevoir les composants physiques ?

2.5.2 Le futur modèle de composants UML 2

Le standard UML 1.4 réduit la notion de composant à une entité physique (physical components) et intervient dans les dernières phases de construction du logiciel. Le futur standard UML 2.0 étend considérablement cette notion et introduit un modèle riche de composants logiques.

Un composant UML 2 [40, 20] expose un ensemble de ports qui définissent ses spécifications en terme d'interfaces fournies et d'interfaces requises. La vue interne de composant UML 2, montre ses propriétés privées qui sont des "classifiers" internes et leurs relations. Cette vue montre comment la vue externe est réalisée en interne. La correspondance entre la vue externe et la vue interne est réalisée par délégation des traitements à des connecteurs sur les ports, connectés à des parties internes (sous-composants ou classes). La figure 2.11 montre une partie du méta-modèle de composant du futur standard et des exemples ambigus de modèles.

L'introduction de composants logiques est une importante contribution de la future norme UML2.0. L'inconvénient est que le concept n'est utilisé que dans un diagramme de classes statique et n'apporte pas une définition complète d'un processus à base de composants. De plus, ce modèle riche montre des zones ambiguës. Par exemple, un *port* fournit ou requiert plusieurs interfaces, mais les composants sont connectés uniquement par leurs ports, la validité des connections entre composants reste floue. Un travail sur la sémantique des connections s'avèrerait utile dans ce modèle. La spécification n'est pas publique à l'heure de l'écriture de ce mémoire. Enfin, notre travail produit des résultats exploitables par des chercheurs et industriels. Ces derniers ne bénéficiant pas encore d'outils standards UML 2.0, notre travail exploite les versions antérieures du langage de modélisation.

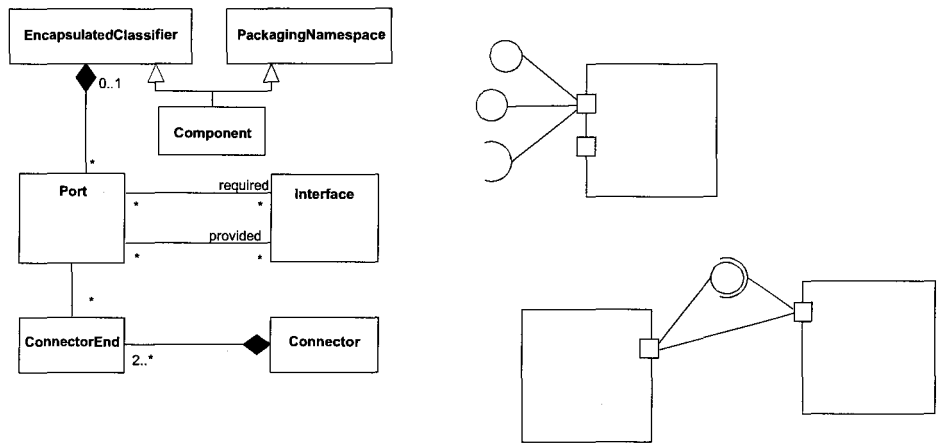


FIG. 2.11 – Extrait du méta-modèle UML2 et exemples de modèles

2.5.3 Les modèles de références de représentation de systèmes à base de composants

UML 1.4 est un standard de modélisation largement utilisé par des outils industriels de modélisation d'applications. Pourtant son usage pour supporter des démarches d'ingénierie orientées composant n'est pas facile. Alors que les concepts sont adéquats pour une décomposition orientée objet d'un système, une approche composant ne semble pas bénéficier des qualités du langage de modélisation. La documentation d'un composant est possible en utilisant des stéréotypes d'UML voire un profil, qui alors n'est plus standard. Et qu'en est-il de la démarche ? UML Component et Catalysis offrent tous deux un cadre de conception d'application à base de composants, avec une notation basée sur UML. Mais, leurs mises en oeuvre ne s'adaptent pas à tous les contextes dans le premier cas, et sont trop complexes à appliquer dans le deuxième cas.

La documentation des composants doit être garante lors de leurs publications d'une meilleure réutilisation. Cette documentation décrit comment le composant réalise ses interfaces et comment il doit être exploité. Cette problématique est abordée dans '"4+1 views" [65] selon plusieurs points de vue. La même approche peut être employée avec d'autres formats comme Reference Model of Open Distributed Processing [81]. Ces modèles de référence permettent d'identifier ce qu'il convient de décrire lorsque l'on documente un système, il convient d'adapter les modèles de références de représentations de systèmes d'information. Un composant pourra alors être considéré comme un système, et nous adopterons ces caractéristiques dans notre représentation de composant.

Modèle "4+1 vues" de composant

La figure 2.12 montre les 5 vues proposées par Kruchten [65] pour modéliser un système d'information en séparant les préoccupations des différentes parties prenantes d'un projet. En effet, Kruchten propose un modèle de description d'architecture d'un système logiciel, basé sur l'utilisation de vues multiples et concurrentes. La multiplicité des vues permet aux différentes parties prenantes d'un système d'adresser des préoccupations différentes. Chacune des vues est décrite avec une notation qui permet de capturer un ensemble de préoccupations. Voici une description succincte des quatre vues :

La **vue logique** supporte essentiellement les besoins fonctionnels, les services rendus par le composant à ses utilisateurs. Elle supporte des démarches orientées objet classiques.

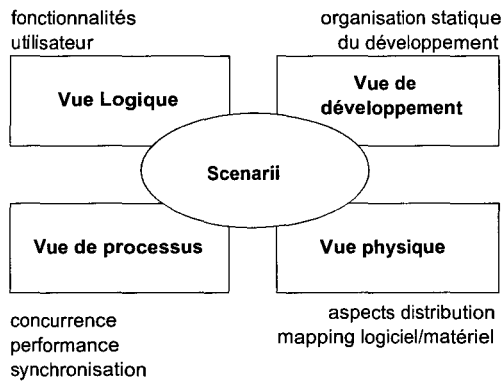


FIG. 2.12 – Modèle "4+1" vues

La **vue de processus** tient compte de services non fonctionnels tels que la performance, la disponibilité, la concurrence, la distribution ...

La **vue de développement** se concentre sur l'oganisation d'un logiciel en modules logiciels.

La **vue physique** décrit la correspondance entre les parties logicielles et matérielles d'un système.

Chaque vue adresse un ensemble de préoccupations. Kruchten décrit les dépendances entre les vues qui garantissent une certaine cohérence globale du système. Ces dépendances s'expriment en terme de règles de conception et d'heuristiques. Notamment, deux constats sont faits. Plus le système est large, plus la différence entre les vues de développement et logique est grande, de même pour les vues de processus et physique. Le constat de Kruchten confirme l'intérêt des démarches composants qui décomposent un système large en plusieurs sous-systèmes moins complexes. Les règles de correspondance entre les vues sont alors plus directes.

Modèle de référence des systèmes ouverts distribués

Le modèle de référence des traitements ouverts distribués [81] définit de manière abstraite, et donc adaptable à un contexte d'application, une architecture qui supporte les aspects distribution, réseaux, interopérabilité, portabilité. Ce modèle doit permettre d'organiser en un tout cohérent, les différentes parties d'un système, c'est-à-dire, la portabilité d'une application sur des plates-formes hétérogènes, l'échange d'informations et l'utilisation de fonctionnalités hébergées et offertes par différents systèmes ouverts dans un environnement distribué, et une transparence de la distribution des applications et utilisateurs.

Le modèle est décomposé en points de vue 2.13, adressant des préoccupations différentes.

- le **point de vue Entreprise** qui décrit l'objectif, la vision et les stratégies. Ce point de vue fait l'étude de l'organisation dans laquelle le système sera développé.
- le **point de vue Information** qui définit le vocabulaire et la manière dont est traitée l'information. Ce point de vue permet d'analyser un système en terme de schéma logique d'architecture, les relations entre les objets métiers qui contiennent l'information du système.
- le **point de vue Traitements (*Computational*)** offre une décomposition des fonctionnalités. Ce point de vue définit les objets contenant du comportement typique du métier.
- le **point de vue Ingénierie** décrit l'infrastructure nécessaire pour supporter la distribution

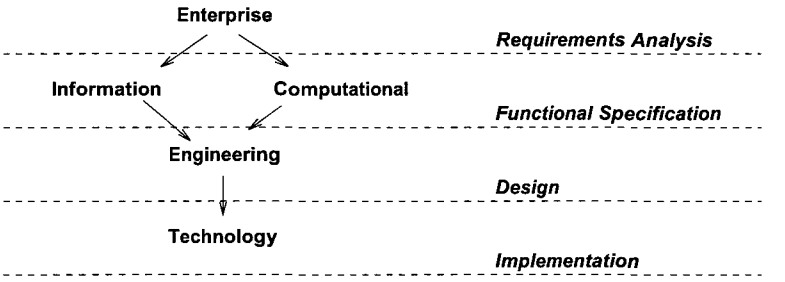


FIG. 2.13 – Points de vues RM-ODP

– le **point de vue Technologie** définit les choix technologiques d'implémentation

Ce modèle de référence définit un cadre de conception pour la représentation de systèmes distribués ouverts. Les cinq points de vue permettent de séparer des préoccupations.

Les systèmes sont de plus en plus complexes. Il est évident aujourd'hui d'adopter une démarche à base de composants qui permet de décomposer un système en sous-système de taille plus raisonnable. Une séparation des préoccupations en points de vue semble nécessaire afin de gérer un projet de construction de tel système en confiant des préoccupations à des équipes différentes. Cependant, il convient de mettre l'accent sur deux points cruciaux. La dépendance entre les composants issus d'un partitionnement fonctionnel du système, et orthogonalement, les interactions entre les préoccupations séparées dans des points de vues distincts. Le modèle doit offrir des vues distinctes tout en garantissant une cohérence globale entre ces vues. Dans le cadre de l'ingénierie des modèles et l'adaptation de méthodes telles que le processus unifié à une démarche à base de composant, nous considérerons ces critères de qualité du logiciel.

2.6 Travaux sur l'aspectisation de la modélisation

Nous l'avons vu, la notion de collaboration est au centre du processus. Elle permet de représenter le comportement d'une partie du système dans l'accomplissement d'une tâche, dans un but précis. Pourtant, elle est souvent oubliée dans les processus classiques. Notre contribution qui sera présentée plus loin, précise et exploite la notion de composant logique, issue des cas d'utilisation, des frameworks et collaboration qui correspond à une bonne unité de modélisation d'un système d'information. Cette notion participe à la réduction du fossé entre l'analyse objet grâce à UML et le développement à base de composants. Avant cela, il convient d'évoquer d'autres travaux connexes à la notion de composant logique et notamment les travaux qui concernent la séparation des préoccupations. Ces travaux étant aussi choisis pour illustrer les applications potentielles de notre travail dans d'autres disciplines.

Depuis ses débuts, les informaticiens ont tenté de mettre en place des bonnes pratiques qui ont fait leurs preuves dans d'autres domaines comme par exemple l'électronique pour les composants, la construction du bâtiment ou le travail à la chaîne pour la démarche projet. Dans le domaine du bâtiment, la séparation des préoccupations est opérationnelle depuis longtemps. Chaque partie prenante d'un projet de construction de bâtiment ne se préoccupe que de son corps de métier. Chacun a son propre plan, dans son propre formalisme, montrant uniquement les informations dont il a besoin afin d'accomplir sa tâche. Un maître d'oeuvre s'occupant de la cohérence globale, et de l'objectif final

qui est la construction d'un bâtiment correspondant aux besoins de la maîtrise d'ouvrage. Pour cela, les besoins et exigences de la maîtrise d'ouvrage sont séparés en préoccupations, par corps de métier, par le maître d'oeuvre. Cela, de manière à optimiser les coûts et la qualité, tout en prenant garde aux dépendances entre ces préoccupations. Par exemple, la construction des murs par la maçonnerie doit avoir lieu avant la pose des menuiseries. La difficulté vient du fait que des exigences, comme par exemple, l'étanchéité de l'ouvrage, dépend de l'exécution de corps de métier multiples : terrassement, maçonnerie, couverture, menuiserie, ...

Dans la construction logicielle, cette séparation des préoccupations est plus difficile à mettre en place car les préoccupations et leurs dépendances avec d'autres sont mal définies. Ainsi, la séparation des tâches dans l'ingénierie logicielle [77] demande l'identification des différentes propriétés du logiciel, qui sont abordées séparément. Ainsi, la construction de logiciels pourrait être réalisée de façon moins complexe et plus compréhensible, ce qui rendrait le logiciel plus réutilisable et maintenable. Le paradigme de séparation des préoccupations et l'assemblage de celles-ci une fois conçues ou développées posent de nombreuses difficultés aujourd'hui. Aucun standard dans ce domaine, ne semble voir le jour. Par contre, des approches et des solutions sont apparues comme, la programmation par aspect, la programmation par sujet, la programmation adaptative, la programmation structurée en contextes.

Programmation orientée Aspects (AOP)

Kiczales [64] introduit la programmation orientée aspect en 97, pour offrir un meilleur découpage du système en séparant les préoccupations. Les différentes fonctions du système sont développées indépendamment et assemblées par le mécanisme de **tissage**. Le tissage peut être statique (AspectJ), c'est-à-dire, par une pré-compilation, ou dynamique (JAC), c'est-à-dire, pendant l'exécution.

AspectJ [67] est un langage, ainsi qu'un compilateur qui a été développé par les chercheurs du XEROX PARC pour un usage industriel. Abouti, il permet la programmation d'aspects en Java et permet de les tisser pour concevoir un logiciel. Cependant, on peut lui reprocher qu'après tissage, il n'est pas possible de séparer du code final les différents aspects originaux pour les réutiliser. De plus, un ensemble de branchements (pointcuts) doit avoir été prévu, ou doit être ajouté afin de permettre le tissage.

La plate-forme **Java Aspect Component** [44] est un framework permettant la séparation des préoccupations distribuées en Java [78]. JAC permet de programmer ses aspects, comme AspectJ. En plus, il offre la possibilité de configurer des aspects déjà existants de manière à les utiliser dans des applications existantes. Il permet le tissage dynamique d'aspects distribués et déjà existants. La notion de composant d'aspect est une entité contenant une préoccupation transversale. Il permet notamment d'enlever, de "détisser" des aspects dynamiquement.

Programmation orientée sujet (SOP)

Ossher [75, 76] introduit la programmation orientée sujet, pour concevoir des systèmes orientés objets comme une composition de sujets. Un sujet étant un ensemble de classes. La composition de sujets combine les classes des sujets et produit de nouveaux sujets qui contiennent les fonctionnalités des sujets composés.

Clarke [14] fait le constat suivant :
Les cas d'utilisation expriment les spécifications du système. La décomposition orientée objet classique est constituée de classes, interfaces, sous-systèmes, ... Il n'y a pas de traçabilité garantie entre les besoins, exprimés en terme de caractéristiques, et de capacité du système à rendre un service, et la conception objet. Cela dégrade la compréhension et la traçabilité. Ainsi, il est difficile de concevoir, de réutiliser une partie du système et de l'étendre. Clarke propose une nouvelle décomposition qui supporte directement des modèles de conception en adéquation avec chaque besoin. Cette modélisation est qualifiée de "orientée sujet" (SOD, Subject-Oriented Design). Ce mécanisme permet de séparer les préoccupations et ainsi chaque modèle correspond à un besoin. Il propose ensuite, un mécanisme de **composition** permettant de réaliser une fusion des différents modèles d'un système. Il propose une extension d'UML pour modéliser des sujets et de la composition de sujets.

La programmation structurée en contextes

L'approche CROME [32, 31, 17, 10, 9, 91] propose un cadre de programmation par objets structurés en contexte. Elle permet la représentation d'objets selon des points de vues qui offrent différentes visions sur le système propre à chaque acteur qui attend une fonctionnalité différente du système. La représentation est organisée en schémas ou plans. Le **plan de base** contient les éléments communs à plusieurs points de vue, et **les plans fonctionnels** adaptent le plan de base à un contexte fonctionnel, une préoccupation. Il ajoute des éléments et des comportements propres à une fonctionnalité. Cette approche répond notamment au problème de crosscutting entre objets et fonctions et donc les cas d'utilisation. L'approche objet permet d'identifier et de découper le système en entités distinctes, mais, par un découpage trop fin, elle ne tient pas forcément compte du découpage fonctionnel du système. Le mécanisme de vues permet de découper le système en différentes fonctions indépendantes.

La programmation adaptative

Pour Lieberherr [73, 66], la décomposition à base de collaborations (cf. figure 2.14) qui considère les applications en terme de participants et de classes est primordiale.

	Classe Cl1	**Classe Cl2**	**Classe Cl3**
Collaboration C1	rôle Cl 1,1	rôle Cl 1,2	rôle Cl 1,3
Collaboration C2	rôle Cl 2,1	rôle Cl 2,2	
Collaboration C3		rôle Cl 3,2	rôle Cl 3,3
Collaboration C4	rôle Cl 4,1	rôle Cl 4,2	rôle Cl 4,3

FIG. 2.14 – Décomposition basée sur les collaborations

Son constat est qu'il manque une représentation adéquate pour les collaborations. Il propose une nouvelle construction de composant qui permet d'exprimer de manière isolée un comportement qui touche plusieurs classes, tout en gardant le schéma de classes classique mais en ajoutant un complément de description, et en supportant une granularité de décomposition qui est en adéquation avec la décomposition orientée objet. Pour Lieberherr,

> *the unit of reuse is generally not the class, but a slice of behavior affecting several classes.*

Il propose un modèle de décomposition multidimensionnel par une approche à base d'**hyperespaces**. La plate-forme *HyperJ* permet de mettre en oeuvre cette décomposition. Dans chaque dimension de l'hyperespace, deux préoccupations sont disjointes. Elles peuvent avoir des intersections si elles appartiennent à des dimensions différentes. Les dimensions sont structurées en modules qui contiennent un certain nombre de concepts, ainsi que des règles de composition, issues de la programmation par sujet. Les modules sont incomplets, et doivent être composés pour construire un système.

Les différentes propositions pour garantir la séparation des préoccupations couvrent rarement la totalité du processus de production d'applications. Elles sont soit focalisées sur la composition de comportements déjà développés dans des modules appelés "aspects", ce qui ne permet pas de capitaliser sur les modèles d'analyse et d'assurer une meilleure traçabilité. Ou alors, elles sont basées spécifiquement sur un système d'information donné (le schéma de base) et donc non prévues pour être réutilisable comme les schémas-vues de CROME [17]. Des travaux sont en cours pour donner une dimension de généricité et donc de réutilisation de ces schémas-vues [31]. Enfin, elles tentent de créer de nouvelles façons de concevoir.

Conclusion de l'état de l'art

Il n'existe pas de procédé parfait pour concevoir des applications à base de composants, avec des architectures nouvelles et les technologies à base de composants. Il est important, au vue de la complexité des systèmes d'information à large échelle, d'appliquer l'ingénierie dirigée par les modèles. Les démarches d'ingénierie classiques ne permettent pas l'identification précoce et la réutilisation de composants. Les démarches à base de composants ou les contributions sur la séparation des préoccupation mettent l'accent sur ces points mais ne sont pas complétement inscrits dans une démarche MDA. En effet, elles n'offrent pas de mécanisme de projection et une séparation nette entre les modèles PIM et les modèles PSM. Enfin, les représentations des composants doivent être complètes pour offrir des critères de courtage de composant. Pour cela, le modèle UML 2 offre une notion intéressante comme le composant logique, mais ce modèle est encore sous-spécifié.

La section suivante présente notre contribution qui se situe dans le contexte de l'analyse et conception à base de composants, et qui répond à la problématique de perte de traçabilité et de sous-spécification.

Chapitre 3

Cas d'utilisation, collaborations, frameworks et composants de modélisation : vers les composants logiques

Ce chapitre définit la notion centrale de la thèse : le composant de modélisation ou **Composant logique**. Le composant logique est issu d'une étude sur l'unité de réutilisation la plus propice dans le cadre du développement d'applications basées sur les besoins dans les environnements à base de composants présentés précédemment et permet de réduire le fossé entre un modèle d'analyse exprimé en UML et le développement d'applications sur des plates-formes technologiques à base de composants.

Tout d'abord, la section 3.1 présente les cas d'utilisation comme concept de base à un processus dirigé par les besoins. Les cas d'utilisation offrent un premier découpage du système qui n'est pas conservé lors du cycle de développement logiciel. Le passage de l'étude des besoins à l'analyse grâce aux collaborations et framework de conception contribue à la perte de la traçabilité globale du système qui lie un besoin aux composants d'une application déployée. La section 3.2 présente la notion de module de cas d'utilisation. La section 3.3 énonce la problématique : la perte de traçabilité et donc les choix complexes de l'analyse objet à l'implémentation à base de composant. Enfin, la section 3.4 présente notre proposition en donnant une définition de la "bonne unité de réutilisation" : les composants logiques.

3.1 Le processus unifié

Une démarche d'ingénierie telle que le **Processus Unifié**, est pilotée par les **cas d'utilisation**, c'est-à-dire qu'elle guide les développeurs vers la conception d'un système répondant aux besoins de ses utilisateurs. Ces besoins, par nature, difficiles à appréhender nécessitent un moyen de communication clair entre tous les intervenants d'un projet informatique. Le cas d'utilisation est un élément qui lie les besoins exprimés par l'utilisateur et les différents livrables du projet. Nous nous situons dans une démarche dirigée par les modèles, aussi, nous considérerons dans la suite que les principaux livrables sont des modèles. La **traçabilité** entre les cas d'utilisation et les différents modèles permet de préserver la cohérence globale du système et de la conserver lors d'évolution des besoins. La suite présente les différents modèles d'un système dans le cadre du Processus Unifié : le modèle de cas d'utilisation, le modèle d'analyse, le modèle de conception et le modèle de composants.

3.1.1 Le modèle de cas d'utilisation

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. L'enchaînement d'activités[1] de **capture des besoins** correspond à l'identification des besoins qui, une fois implémentés, apportent une plus-value 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. Un acteur interagit avec le système selon une séquence d'actions effectuées par le système pour produire un résultat satisfaisant pour l'acteur. Les interactions entre les acteurs et le système sont exprimées dans les cas d'utilisation [61].

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. Par exemple, dans un système de guichet automatique bancaire (GAB), le client de la banque est acteur de ce système. Il est représenté par un stick-man (cf figure 3.1). L'ensemble des acteurs définit l'environnement du système.

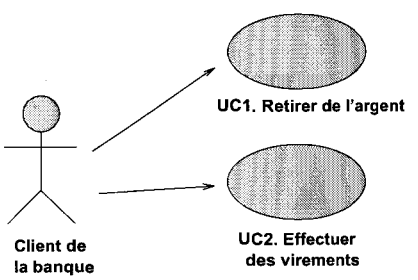


FIG. 3.1 – Diagramme de cas d'utilisation simplifié d'un GAB

[1] au sens "workflow" du Processus Unifié

Les cas d'utilisation spécifient le système. Ils sont représentés par des ovales (cf figure 3.1). En analysant toutes les utilisations potentielles des différents acteurs, l'ensemble des besoins fonctionnels est répertorié dans les cas d'utilisation. Selon Jacobson [62], 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. Il peut y en avoir plusieurs par cas d'utilisation. Un scénario particulier, dit "nominal", est choisi, car il représente le "cas où tout va bien", où l'acteur est satisfait. Les autres scénarii sont qualifiés d'alternatifs.

Description du cas d'utilisation Retirer de l'argent

Acteur principal Client de la banque

Scénario possible
1. Le client de la banque s'identifie
2. Le client de la banque choisit le compte sur lequel il veut retirer de l'argent
3. Le client indique le montant qu'il veut retirer
4. Le système déduit le montant du compte et délivre l'argent

3.1.2 Le modèle d'analyse

Après avoir décrit complètement un cas d'utilisation, c'est-à-dire, l'ensemble des chemins d'exécution, ou scénarii possibles, les éléments de l'architecture nécessaires pour réaliser le cas d'utilisation sont identifiés. Généralement, cela correspond à une structure de classes d'analyse et leurs relations. Chaque itération d'analyse réalise un ensemble de cas d'utilisation en créant de nouvelles classes ou en réutilisant des classes découvertes dans les itérations précédentes. Une classe d'analyse joue un ou plusieurs **rôles** dans la réalisation d'un cas d'utilisation. Un rôle correspond à une vue sur une classe, et spécifie les responsabilités de la classe qui permettent de trouver ses méthodes et attributs.

Un diagramme de séquence UML est utilisé pour représenter la réalisation d'un cas d'utilisation dans le modèle d'analyse (cf figure 3.2). Dans ce diagramme, figurent des objets correspondant à la notion de rôle. Le diagramme de classes UML (cf figure 3.3) permet de représenter les classes d'analyse et les relations issues de ce diagramme de séquence. L'ensemble des diagrammes de séquence permet de compléter le diagramme de classes. Le modèle d'analyse utilise les trois stéréotypes de classes de Jacobson [90] destinés à répartir les classes d'analyse selon trois catégories. Ce découpage étant jugé plus en adéquation avec le type d'applications développées aujourd'hui et surtout permettant de séparer l'interface homme-machine, sujette à de nombreuses et courantes modifications, le code de l'application qui est plus stable mais destiné à un usage particulier, et enfin, la partie la plus stable d'une architecture, c'est-à-dire, les données persistantes et les services utilisables par plusieurs applications d'un système. Les classes **frontières**, comme *InterfaceGuichet* (cf figure 3.2), modélisent une interaction entre le système et son environnement. Les classes de **contrôle**, comme *Retrait*, représentent un séquencement, une transaction ou une encapsulation du contrôle lié à un cas d'utilisation, comme ici. Les classes **entités**, comme *Compte*, correspondent à des informations durables, souvent persistantes. Par nature, ce sont les plus susceptibles d'être partagées par plusieurs réalisations de cas d'utilisation. Les flèches symbolisent les messages envoyés d'un rôle à un autre. Le rôle qui reçoit

un message a la **responsabilité** de répondre à ce message. Une classe est un regroupement de plusieurs rôles, et regroupe, par conséquent, toutes les responsabilités de ces rôles. Ainsi, l'ensemble des responsabilités permet de définir les interfaces du système. Et la séparation de trois niveaux entité, contrôle et frontière donne différents points de vue sur le système et permet de mieux partitionner celui-ci selon ces différentes préoccupations.

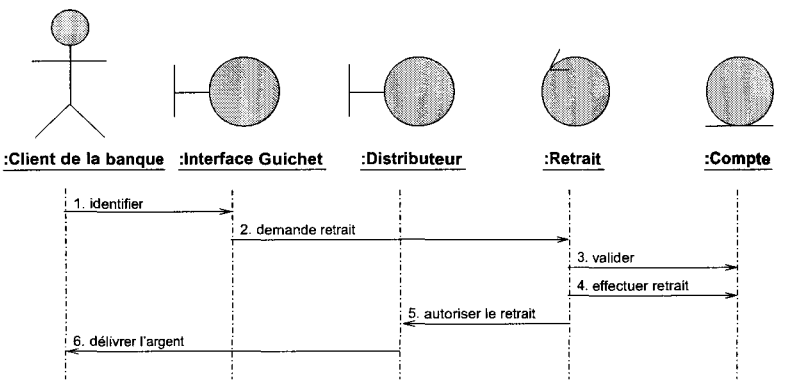


FIG. 3.2 – Diagramme de séquence

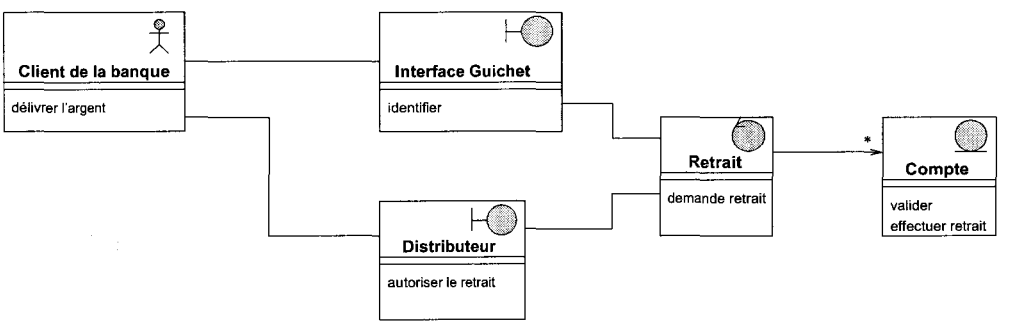


FIG. 3.3 – Diagramme de classes

La notion de collaboration

Dans les spécifications Unified Modeling Language (UML) de l'OMG [90], une collaboration décrit (cf figure 3.4) comment un cas d'utilisation est réalisé par un ensemble de classifieurs – **Classifier** – et les associations utilisées d'une façon spécifique. Elle définit un ensemble de rôles, joués par des instances, ainsi qu'un ensemble d'interactions qui définissent une communication entre les instances quand elles jouent un rôle. La collaboration inclut les concepts correspondant aux participants à un ensemble de buts donnés : – **ClassifierRoles** – et de – **AssociationRoles** –. Ils représentent les classifiers et associations qui prennent part à la réalisation du cas d'utilisation. La collaboration spécifie une vue (au sens restriction) du modèle.

Une – **Interaction** – est définie dans le contexte d'une collaboration. Elle définit la communication entre des rôles dans une collaboration – **ClassifierRole** –. Une interaction est décrite par un ensemble de messages – **Message** –, chacun spécifiant une communication. Par exemple, un signal envoyé par un émetteur – **sender** – et une invocation de méthode reçue par le récepteur – **receiver** –. Les collaborations peuvent être utilisées pour exprimer comment les cas d'utilisation sont réalisés.

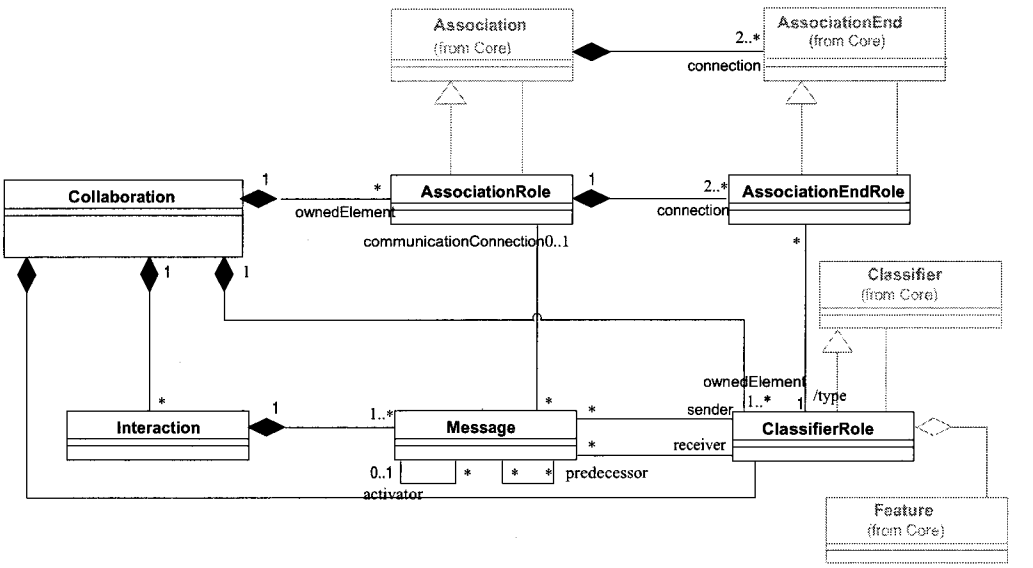


FIG. 3.4 – Extrait du méta-modèle UML 1.4

Dans Catalysis [56], les interactions entre les objets sont exploitées dans les collaborations. En effet, l'ensemble des interactions entre quelques objets peut donner lieu à la réalisation d'une opération de plus haut niveau. Dans la figure 3.5, le retrait d'argent est une collaboration entre les objets :Compte, :PorteurCarteCB et :GuichetAutomatique. Une manière d'exprimer le protocole de cette collaboration est le diagramme d'interactions (cf figure 3.2).

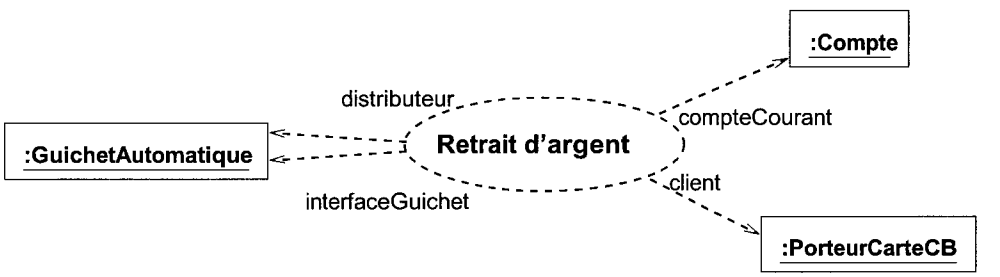


FIG. 3.5 – Exemple de collaboration

La notion de Framework de conception

Le concept de framework de Wills[2] [89] repose sur le postulat suivant :

> *Classes are not the best focus for object-oriented design. The most useful components in a design are frameworks*

[2]Nous nommerons ce type de framework "framework de conception" afin de ne pas être confondu avec des frameworks d'implémentation, librairies réutilisables et adaptables à un contexte technologique.

Wills décrit la notion de framework de conception comme une collaboration d'objets qui intéragissent dans un but donné. Cette notion de framework est exploitée dans la démarche Catalysis [56] afin de réutiliser un comportement d'un ensemble d'objets. Dans un système, un objet peut être participant à plusieurs frameworks de conception. Il joue un ou plusieurs **rôles** dans chaque framework. Pour chacun de ces rôles, l'objet a une **interface** qui spécifie un comportement attendu de cet objet. Avant d'implémenter un objet, il convient de composer l'ensemble de ses interfaces et donc de trouver ses rôles dans tous les frameworks où il participe. Ainsi, l'objet est le résultat final de la composition des interfaces de ses rôles (cf figure 3.6). Il est difficile d'anticiper l'ensemble des rôles qu'un objet peut tenir dans des frameworks de conception. Le framework de conception est un ensemble d'interfaces et d'interactions. Un cas d'utilisation peut être réalisé par un framework de conception en terme de responsabilités et de rôles.

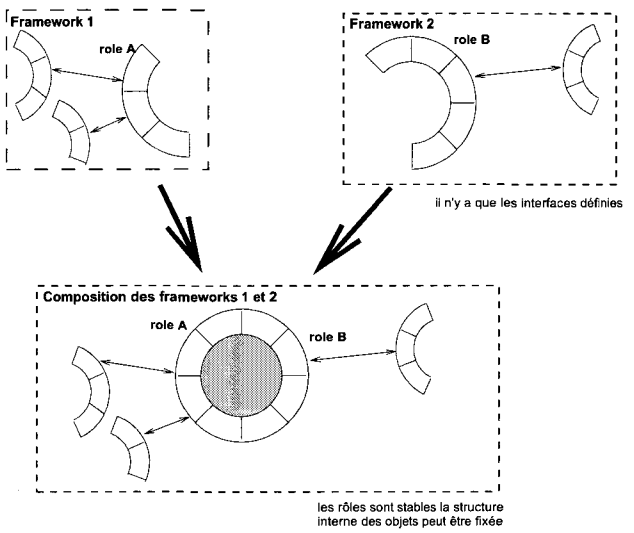


FIG. 3.6 – Composition de frameworks

Dans le contexte de projet d'ingénierie dirigé par les cas d'utilisation, l'enchaînement d'activités d'analyse consiste pour chaque cas d'utilisation à étudier les participants, **rôles**, à ce cas. Les cas d'utilisation définis par Jacobson servent de base à l'analyse des interactions entre les objets d'un système pour fournir les services décrits dans les cas d'utilisation aux acteurs. Étudier les rôles, consiste à définir leurs comportements, leurs responsabilités et donc définir leurs interfaces pour réaliser le cas d'utilisation. Une fois les principaux cas d'utilisation traités, les interfaces définies de chaque participant sont regroupées et permettent de définir les classes du système en composant ses interfaces. Ce travail peut utiliser l'analyse métier et notamment la définition des concepts métiers, inévitables dans le domaine d'application du système développé.

3.1.3 Le modèle de conception

Le modèle d'analyse est neutre vis-à-vis d'une technologie, il est issu du modèle de cas d'utilisation. La conception consiste à transformer les classes d'analyse et donc le modèle d'analyse en un modèle de conception qui contient les classes qui seront réellement implémentées. Par exemple, la classe frontière *Interface Guichet* deviendra dans le modèle de conception, trois classes : *Lecteur de carte*,

Ecran, *ClavierNumerique*. Les **sous-systèmes** regroupent logiquement les classes, et permettent de mieux appréhender le système. Chaque sous-système fournit et utilise un ensemble d'interfaces qui définissent le contexte.

3.1.4 Le modèle de composants

L'activité qui consiste à décrire le système dans le modèle de composants, consiste à regrouper les éléments contenus dans le modèle de conception en composants. C'est une notion de composant physique, c'est à dire, un élément de l'architecture telle quelle sera déployée et exécutée. Il n'y a pas de correspondance directe entre le modèle de conception et le modèle de composant. Cela dépend de l'infrastructure technologique sur laquelle sera déployée l'application.

3.2 Cas d'utilisation et modularité

Le **modèle de cas d'utilisation** a été défini initialement par Ivar Jacobson [60] en 87, comme une perspective externe au système, il est vu en "boîte noire". Son objectif est de modéliser les fonctionnalités du système, du point de vue de ses utilisateurs. Le modèle de cas d'utilisation offre un premier découpage fonctionnel du système : les fonctionnalités qu'attendent les utilisateurs et le client. Selon ce découpage, il est envisageable de concevoir un système qui serait capable d'évoluer dans le temps en ajoutant un composant au système chaque fois qu'un nouveau besoin serait exprimé sous la forme d'un cas d'utilisation. Cette façon de découper un système porte le nom de **modularité des cas d'utilisation**

Dans [23], Jacobson définit la modularité offerte par les cas d'utilisation.

> *If we could keep use cases separate, even while they cross several components, and maintain that separation all the way down through all lifecycle activities from requirements to test via analysis, design, implementation and testing, and, yes, also in runtime, we would get a system that was dramatically simpler to understand, to change, and to maintain.*

Si les cas d'utilisation sont gardés séparément, même si les fonctionnalités qu'ils décrivent sont transverses à plusieurs composants déjà définis, et que cette séparation est conservée au travers des différentes activités du cycle de développement logiciel, des besoins au test, en passant par l'analyse, la conception et le test unitaire, jusqu'à l'exécution, on peut obtenir un système considérablement simple à appréhender, faire évoluer et maintenir. En gardant les cas d'utilisation séparés, on obtient comme composants, des **modules de cas d'utilisation**.

3.3 Problématique : la perte de traçabilité

Les cas d'utilisation dirigent le processus. Les besoins sont exprimés sous la forme de cas d'utilisation sur lesquels s'appuient les responsables de projet pour planifier le développement. Les cas d'utilisation sont réalisés par des classes et sous-systèmes au cours de l'analyse et la conception. Ensuite les composants sont développés et intégrés. Les cas d'utilisation servent à concevoir des tests d'utilisation du système. Par composants, on entend ici les composants de déploiement, les exécutables qui sont l'aboutissement d'un processus d'ingénierie.

Si le processus est bien maîtrisé lors de l'analyse d'un système grâce aux notions de collaboration et de la composition de frameworks, le passage des classes d'analyse et sous-systèmes du modèle de conception aux composants n'est pas clairement défini. Il est souvent influencé par les choix d'infrastructure technique, qui peuvent mener à remettre en cause les choix des analystes et concepteurs, en terme de sous-systèmes, de dépendances et d'interfaces. De plus, la livraison classique d'un logiciel consiste souvent à fournir le code source et les exécutables. Dès lors, il y a perte de traçabilité entre les modèles livrés qui permettrait de définir l'impact d'une anomalie sur un besoin ou d'identifier les composants qui doivent être modifiés suite à une évolution des besoins. Sans une réelle rigueur dans l'application du Processus Unifié, qui de surcroît coûte cher, les qualités apportées par un processus dirigé par les cas d'utilisation ne sont plus exploitées.

Développons cette problématique pour ensuite donner des pistes de solutions.

Rupture de la traçabilité dans le cycle classique de développement logiciel

Le Processus Unifié, **dirigé par les cas d'utilisation** peut être basiquement schématisé en quatre étapes, correspondant à quatre enchaînements d'activités (cf figure 3.7). Après une brève description de celui-ci, nous discuterons de la faisabilité de la modularité de cas d'utilisation.

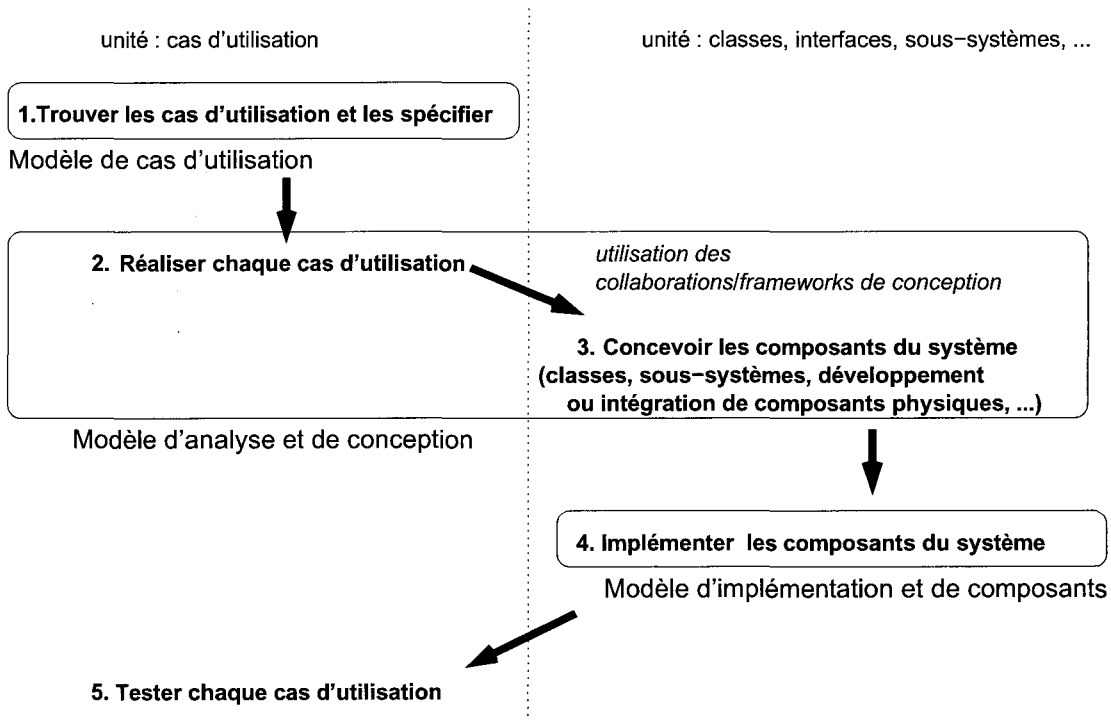


FIG. 3.7 – Enchaînements d'activités d'un processus dirigé par les cas d'utilisation

Chacun de ces 5 enchaînements d'activités est de la responsabilité d'un membre différent d'une équipe de projet. Les enchaînements d'activités 1., 2., et 5. sont basés sur les cas d'utilisation. Les enchaînements d'activité 3. et 4. ne sont pas basés sur les cas d'utilisation. En effet, la conception

et l'implémentation des classes et sous-systèmes sont libres par rapport au découpage de l'architecture induit par les cas d'utilisation. Ils utilisent comme source d'information la description des cas d'utilisation, cependant ils ne capitalisent pas sur ce partitionnement. Ainsi, **le développement de système ne capitalise pas sur l'aspect modulaire des cas d'utilisation**. L'évolution des besoins, le changement et la correction d'anomalies détectées dans l'enchaînement d'activités de test, ou lors de l'utilisation par les utilisateurs finaux demeurent complexes et difficiles à tracer.

Plus précisément, lors du passage de l'enchaînement "2. Réalisation des cas d'utilisation" à "3. Analyse conception des composants du système", il existe deux effets de décomposition [23] qui participent à la rupture de la traçabilité.

1. *Un composant ne contient pas que le code pour réaliser une partie d'un cas d'utilisation, mais aussi le code pour réaliser plusieurs autres cas d'utilisation (cf. figure 3.8).*

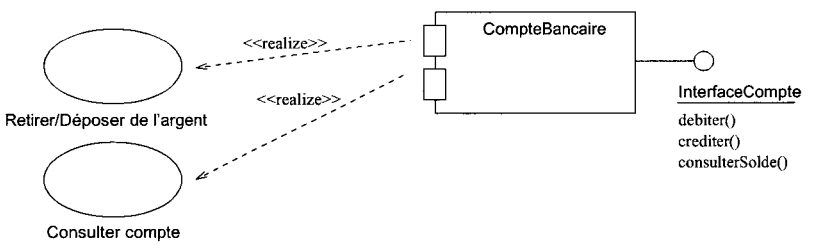


FIG. 3.8 – Enchevêtrement du code de composant

2. *Un cas d'utilisation est souvent réalisé par le code de plusieurs composants inter-connectés (cf. figure 3.9).*

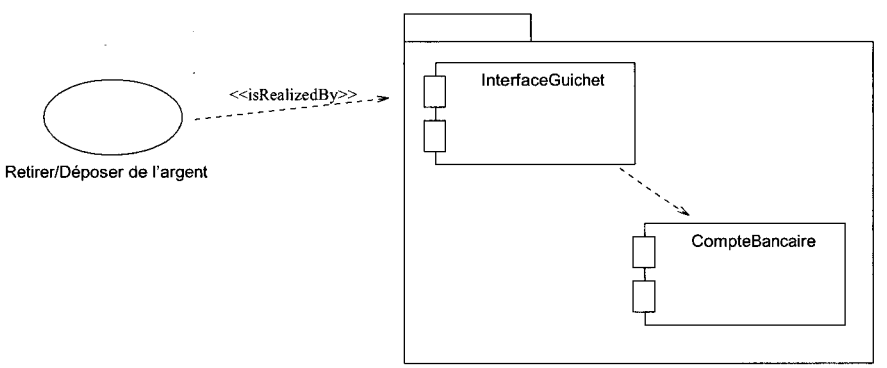


FIG. 3.9 – Eparpillement d'un cas d'utilisation sur plusiques composants

Cela implique que lors de l'évolution des besoins, il est difficile de faire évoluer un système conçu à partir de composants de granularité fine, issus de la décomposition initiale par les cas d'utilisation : la modularité de cas d'utilisation. La modularité de Jacobson semble ne pas être la bonne unité de réutilisation. Le postulat de Jacobson, bien que censé, n'est pas réaliste. Les cas d'utilisation ne sont pas la bonne unité de réutilisation, par contre, ils sont une source d'information et d'organisation pour le travail d'analyse et conception à base de composants.

La problématique se situe donc au passage de l'étape (2.) à l'étape (3.) d'un processus dirigé par les cas d'utilisation, c'est-à-dire, la recherche des collaborations des éléments de l'architecture pour réaliser les cas d'utilisation. Un cas d'utilisation est réalisé par un ensemble d'objets, qui jouent des rôles au sein de collaborations. Certains objets peuvent alors jouer différents rôles puisqu'ils interviennent dans la réalisation de plusieurs cas d'utilisation. Pour améliorer la traçabilité, nous préconisons de descendre au niveau classes et associations, c'est-à-dire l'analyse des collaborations, pour remonter en abstraction à la notion de composants (Passage de l'étape 2 à l'étape 3 du schéma) en intégrant le mécanisme de composition de frameworks de conception. Ce procédé est complexe et nécessite de conserver une cohérence globale entre des dépendances de cas d'utilisation et des relations entre objets qui réalisent ces cas d'utilisation. Un processus classique se focalise uniquement sur les relations nécessaires à la réalisation d'un cas d'utilisation et non sur les relations entre cas d'utilisation. A ce niveau, le processus n'offre plus de cadre de travail. A part le maintien rigoureux de la traçabilité grâce à la réalisation laborieuse d'une matrice de correspondance entre les besoins (cas d'utilisation) et l'ensemble des éléments de l'architecture, le processus ne garantit pas que l'impact d'une évolution des besoins soit aisément détecté. Afin de garantir cette traçabilité, la section suivante donne des pistes de solution à cette problématique en exploitant la modularité des cas d'utilisation lors du passage des activités dirigées par les cas d'utilisation à des activités dirigées par les éléments architecturaux (classes, sous-systèmes, interfaces, composants physique, ...).

Pour construire un système depuis les collaborations, il faut, pour chaque objet, faire l'inventaire des participations de cet objet à différents rôles, pour cela, il faut mettre en commun le vocabulaire du changement d'état. Nous introduisons la notion de composant logique dont la granularité correspond aux frameworks de Wills. Le composant logique n'est pas un framework de conception. En effet, un morceau de framework peut être spécifié dans plusieurs cas d'utilisation [60, 45]. De plus, l'ajout de la notion de données aux frameworks permet de rendre le composant logique plus autonome.

3.4 Notre proposition : les composants logiques

Les cas d'utilisation permettent d'exprimer les spécifications d'un système. La décomposition orientée objet classique, c'est-à-dire par classes, interfaces et méthodes, ne garantit pas une bonne traçabilité entre les besoins, exprimés en terme de caractéristiques et de capacités, et la conception et l'implémentation orientée objet [14]. Il en résulte une dégradation de la compréhension du système et la traçabilité. Ainsi, il est difficile de réutiliser des parties du système et d'étendre le système. L'objet n'est pas la bonne unité de réutilisation. UML, langage de modélisation objet, n'est pas adapté, par nature, à la conception de systèmes à base de composants et par conséquence, n'offre pas de cadre pour cela.

Dans une véritable démarche à base de composants, l'identification des composants doit être précoce afin de capitaliser sur un découpage de qualité. Les cas d'utilisation sont souvent les premiers livrables d'un projet. Ils donnent une vue globale des fonctionnalités du système et offrent un découpage fonctionnel de celui-ci qui permet de gérer les premières phases du projet. Notre proposition tente de capitaliser sur l'aspect modulaire des cas d'utilisation tout en minimisant les inconvénients liés à l'enchevêtrement de plusieurs cas d'utilisation dans un composant et l'éparpillement des fonctionnalités décrites dans un cas d'utilisation sur plusieurs composants. Pour cela, nous proposons dans le modèle de cas d'utilisation, de regrouper un ensemble de cas d'utilisation au sein d'un composant.

Afin de trouver quels cas d'utilisation doivent être regroupés dans un même composant, nous utilisons les collaborations.

La notion de médium [7, 8, 87] est une notion, elle aussi, issue de l'étude des collaborations. Les média de communication et le processus sous-jacent intègrent la notion de données aux collaborations, une interaction est alors auto-contenue et spécifiée indépendamment de tout contexte d'utilisation. Ces collaborations sont réalisées par des frameworks de conception. Les frameworks sont souvent utilisés pour définir les interfaces des éléments de l'architecture, aussi, l'ajout de données à ceux-ci va nous permettre de définir de véritables unités de réutilisation.

La notion de composant logique correspond, dans notre proposition, à un ensemble de cas d'utilisation réalisés par un ensemble de collaborations corrélées, et de frameworks issus de ces collaborations, puis contenant les données nécessaires pour exister en dehors de tout contexte. Enfin, pour être complet, les interactions entre composants logiques non prises en charge par les parties du composant doivent pouvoir exprimer que le fonctionnement du composant n'est garanti que s'il est connecté à d'autres composants qui ont la responsabilité attendue. Il faut pouvoir catégoriser les services rendus par un composant dans des interfaces multiples. Ces catégories doivent correspondre aux réalisations de cas d'utilisation et donc aux collaborations.

Le composant logique

Un composant logique est une unité de modélisation réutilisable qui contient un ensemble de modèles offrant chacun un point de vue sur son architecture (des cas d'utilisation aux problématiques d'assemblage). Un système est partitionné en composants logiques. Ce partitionnement doit être consolidé et conservé tout au long du cycle de vie logiciel. Ce qui garantit une meilleure traçabilité et améliore la gestion de l'évolution des besoins.

Un composant logique est composé de cas d'utilisation. Les cas d'utilisation sont réalisés par un ensemble de collaborations. L'objectif est d'atteindre une certaine qualité du partitionnement en composants logiques. Celle-ci réside dans le fait que de nombreux objets participent à plusieurs collaborations d'un même composant logique et peu dans des collaborations d'autres composants logiques. On réduit ainsi les dépendances entre composants. La structure statique du composant logique est issue du rassemblement des rôles joués par les éléments du composant logique dans les collaborations. Une collaboration correspondant à un framework, la structure statique d'un composant logique peut être trouvée par composition de frameworks, comme dans la démarche Catalysis. Ainsi, ce mécanisme permet de définir les deux types de parties du composants :

- Les parties internes *de contrôle* pour la gestion des interactions nécessaires au fonctionnement du composant, au contrat fonctionnel.
- Les parties internes *entités* pour la gestion des données.

Un composant logique offre des comportements catégorisés dans un ensemble d'interfaces requises. Et les rôles de collaborations réalisés dans d'autres composants logiques sont accessibles par connexion d'interfaces requises aux interfaces fournies des autres composants. L'assemblage de composants logiques permet de résoudre le fait que les collaborations font intervenir des rôles externes.

La figure 3.10 synthétise la notion de composant logique et donne une idée du processus sous-jacent. Après modélisation des fonctionnalités du système d'information grâce aux cas d'utilisation, l'objectif est l'identification des collaborations. L'étude des éléments enrôlés dans chaque collaboration permet d'identifier les limites des composants.

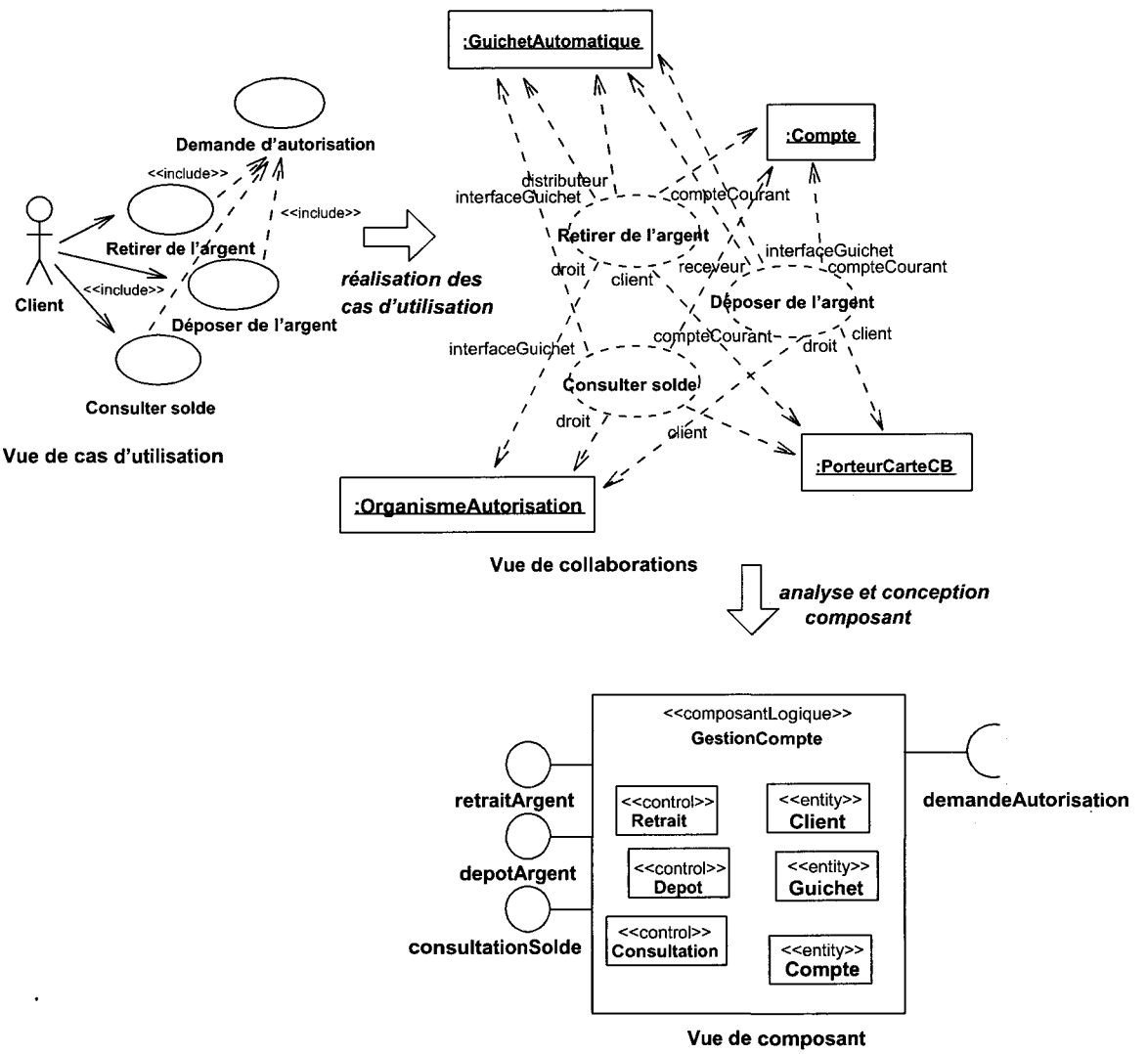


FIG. 3.10 – Des cas d'utilisation aux composants logiques

3.5 Synthèse

Dans l'ingénierie du logiciel, exploiter un modèle d'analyse exprimé en UML, pour développer sur des plates-formes à base de composants est complexe. Nous avons vu que une cause peut être la perte de traçabilité entre l'expression des besoins et leur réalisation et une analyse de composants basée uniquement sur un modèle de classes. L'importance des collaborations n'est pas suffisamment prise en compte. Notre travail s'inspire des divers travaux présentés. Notre problématique est de réduire le

fossé entre la modélisation orientée objet, et l'implémentation sur des plates-formes technologiques à base de composants. Aussi, pour ce cas, nous avons contribué à réduire le fossé entre analyse objet et conception à base de composants grâce à la notion de **composants logiques**, que nous avons défini dans ce chapitre. Le fait est que les démarches d'ingénierie actuelles reposent essentiellement sur l'"expérience" des participants aux projets, et sur un maintien "manuel" de la traçabilité. La notion de collaboration a été trop négligée, à la fois dans les approches (elle est présente mais en arrière plan) et dans les formalismes, ou notations utilisées (il y a un manque dans la représentation des collaborations). La recherche et les solutions au principe de séparation des préoccupations sont des réponses au problème de manque de visibilité globale des différentes fonctionnalités du système, de l'évolution, réutilisation et de facilité de l'analyse et conception des préoccupations. Afin de passer d'un modèle d'analyse objet, exprimé en UML, à un développement sur des plates-formes à base de composants, il faut faire de nombreux choix de conception, et aucun cadre de travail n'a encore été défini, que ce soit dans les travaux de Catalysis qui n'apportent pas de véritable méthodologie, ou dans le Processus Unifié, qui fédère les bonnes pratiques du génie logiciel sans offrir un véritable cadre de gestion du cycle de développement du logiciel. Ils laissent libres les parties prenantes du projet, qui peuvent prendre leurs propres décisions. Aussi, plus le projet est large, plus les décisions prises peuvent être contradictoires et mener à des incohérences.

Notre contribution est présentée dans la suite de ce mémoire. Elle propose une décomposition du système selon les "comportements qui affectent plusieurs classes" de Lieberherr, c'est-à-dire, les collaborations. Notre proposition est de monter en abstraction très tôt dans le processus, et de passer à une approche composant avant de décomposer le système en classes. Ainsi, il serait possible d'identifier de manière précoce, les différentes parties d'un système ainsi que leurs dépendances. Le modèle d'un système est découpé en plusieurs vues, correspondant aux préoccupations dominantes d'un projet d'ingénierie. Le modèle proposé peut ensuite être projeté vers un modèle spécifique à une technologie. Le même découpage est respecté et il y a correspondance directe de concepts présents dans les deux formalismes. Enfin, la conception de type framework du modèle cible permet le développement sur une plate-forme spécifique. Notre contribution est la formalisation d'un modèle indépendant d'une plate-forme technologique (PIM) reposant sur notre notion de composant logique et une proposition de démarche. Un prototype de projection vers la plate-forme EJB est proposé pour valider le modèle.

Chapitre 4

Le modèle de composants logiques

Cette partie du mémoire présente notre proposition de démarche pour répondre à la problématique présentée précédemment. Le Processus Unifié à base de Composants doit permettre de guider le développeur dans le développement d'applications en identifiant suffisamment tôt les composants du système et en modélisant ceux-ci dans un formalisme riche mais indépendant d'une plate-forme technologique. Ainsi, il y a capitalisation sur le travail d'analyse/conception. Ce formalisme doit permettre de manipuler la notion de composant logique selon différents points de vue. Cette notion de composant logique doit être omniprésente ce qui améliore la traçabilité et la gestion du changement.

Le modèle support de la démarche Component Unified Process, dénommé **modèle CUP** est un modèle de composants logiques. Il est conçu pour formaliser la description d'un système d'information à base de composants en introduisant de nouvelles notions afin de répondre aux problématiques soulevées dans la première partie de ce document. Dans un contexte d'ingénierie dirigée par les modèles, il vise à simplifier le passage complexe de l'analyse à la conception à base de composants en assurant un découpage du système structurant et pérenne tout au long des phases du cycle de production du système. Dans cette partie, nous décrivons ce formalisme de modélisation de systèmes à base de composants logiques en détaillant son méta-modèle qui s'inspire du méta-modèle Unified Modeling Language 1.4 [90]. Afin d'être mieux appréhendée, la démarche supportée par ce formalisme est présentée, dans sa mise en oeuvre grâce à la modélisation d'une étude de cas simpliste qui met en valeur les points clés du processus unifié à base de composants **Component Unified Process**[1].

Le méta-modèle CUP présenté dans l'article [46] est riche et indépendant d'une plate-forme technologique. Lors de l'étude de la formalisation d'un système à base de composants, des concepts ont été identifiés et exploités dans le but de capitaliser sur des notions présentes dans diverses technologies actuelles. Ainsi, un formalisme riche et indépendant d'une technologie, tout en étant en concordance avec les middlewares dits à base de composants est proposé. En outre, la notion de **composant logique** est bien distincte des composants dits technologiques tels que Enterprise Java Beans, Corba Component Model, ... Mais, par un mécanisme de projection, le modèle de composants logiques doit permettre la génération de tels composants. La section 4.1 présente les points clés de l'approche. Les sections suivantes décrivent le méta-modèle et la démarche sous-jacente.

[1] Adaptation du Processus Unifié vers une approche à base de composants

4.1 Le Processus Unifié à base de Composants

Le **Component Unified Process** est une *démarche d'ingénierie* (cf 4.1.1) itérative, basée sur les besoins, et orientée composant, destinée à modéliser et développer une application. Il s'inscrit dans l'initiative *Model Driven Architecture* (cf 4.1.2) 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* (cf 4.1.3), 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. Cette section présente les points clés du Processus Unifié de développement d'application à base de composants.

4.1.1 Une démarche d'ingénierie

Dans la première partie de ce document, la notion de démarche d'ingénierie est présentée comme une séquence d'itérations qui raffinent et enrichissent les livrables (modèles) d'un projet. Le Component Unified Process s'inscrit dans cette définition.

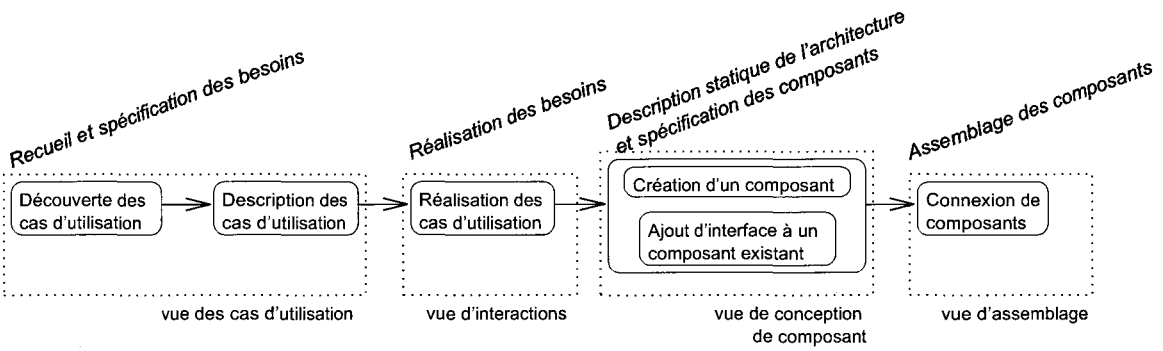


FIG. 4.1 – Démarche d'ingénierie à base de composants

La démarche CUP est basée sur les besoins

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 (cf. chapitre 3). L'analyste de cas d'utilisation et les utilisateurs finaux, de concert, 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, 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 (cf. chapitre 3), 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.

La démarche CUP est itérative

Effectuer les activités d'ingénierie présentées ci-dessus successivement, serait trop restrictif et conduirait à un rejet de la démarche dans le cadre de projet d'ingénierie. En fait, l'introduction d'un nouvel élément peut être fait dans n'importe quelle vue, dans pratiquement n'importe quel ordre. Cependant, cela implique la réalisation d'autres activités et donc un enrichissement et raffinement des autres vues du modèle du système. Dans la suite de ce document, les activités sont présentées les unes à la suite des autres dans un but didactique. Cependant, les contraintes de dépendance exprimées plus loin permettent de s'abstraire de l'ordre de présentation des tâches du processus unifié à base de composants CUP et maintenir une cohérence globale.

4.1.2 Un processus inscrit dans la Model Driven Architecture

Comme nous l'avons vu dans la première partie, un processus MDA [33] inclut une première phase qui consiste à définir un modèle indépendant d'une plate-forme spécifique, ce que propose le formalisme CUP [2] (PIM - Platform Independent Model) (cf figure 4.2 étape **1**). Ce modèle PIM est raffiné et enrichi (cf figure 4.2 étape **2**). Le modèle est suffisamment riche pour permettre une projection vers une plate-forme spécifique (cf figure 4.2 étape **3**). Le résultat de cette projection est enfin optimisé par diverses expertises techniques (cf figure 4.2 étape **4**).

[2] dans la terminologie MDA de l'OMG

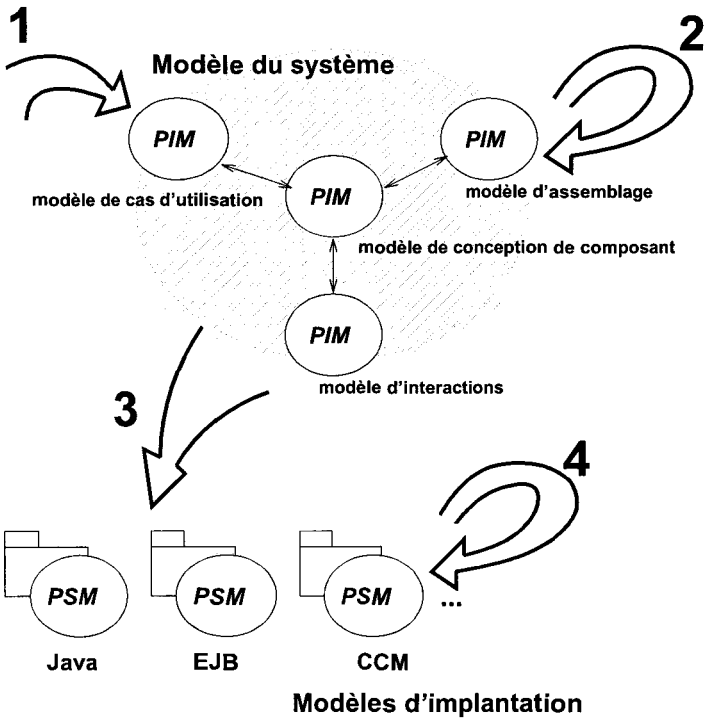


FIG. 4.2 – Modèles d'un système découpé en vues

4.1.3 Travail coopératif, séparation des préoccupations et cohérence de l'architecture

Séparation des préoccupations

Le processus d'ingénierie CUP décrit des activités permettant à chaque intervenant d'un projet d'accomplir un certain nombre de tâches dans le but de construire une application. Chaque intervenant a une préoccupation, et n'a pas le même point de vue sur l'architecture. Le méta-modèle intègre cette séparation des préoccupations, en offrant un formalisme pour décrire une architecture de systèmes à base de composants découpée selon quatre points de vue prédominants : les **vues**. Chaque sous-partie du méta-modèle est décrite dans un paquetage (cf figure 4.3). Cette compartimentation entre paquetages au sein du méta-modèle est une distinction par rapport au méta-modèle UML 1.4, qui est un "méta-modèle plat", et dans lequel ne transparaissent pas les usages potentiels du langage. Le formalisme de modélisation à base de composants est articulé autour de ces quatre vues. Chacune de ces vues est décrite par une partie du méta-modèle CUP (cf figure 4.3), contenant les concepts mis en jeu et leurs relations.

- **La vue de cas d'utilisation** d'un système décrit les besoins des utilisateurs finaux et les **dépendances entre composants** au niveau fonctionnel. Cette vue doit être compréhensible par toutes les parties prenantes du projet et contient donc essentiellement des documents en langage naturel.
- **La vue d'interactions** des éléments d'un système donne une représentation plus fine des dépendances entre composants. Notamment, elle décrit les messages échangés et ainsi permet de définir les interfaces des éléments de l'architecture. Cette vue permet de représenter les collaborations discutées dans la section 3.
- **La vue de conception de composant** décrit les structures interne et externe des composants dans un formalisme orienté objet (classes, interfaces, associations, ...).

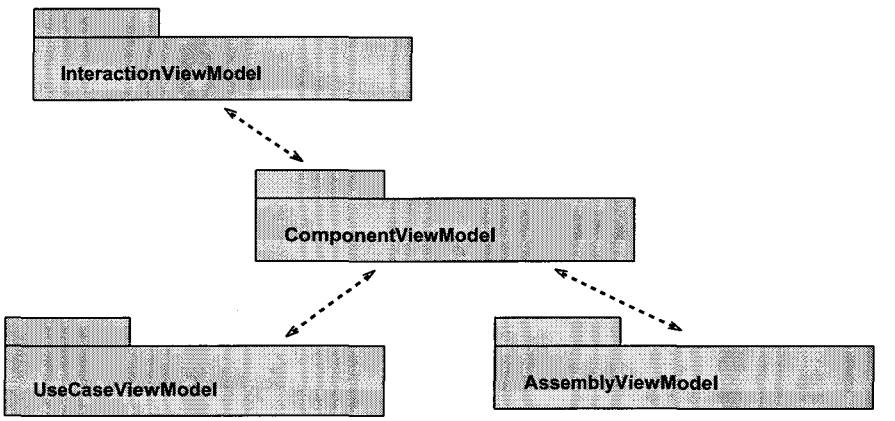


FIG. 4.3 – Méta-modèle CUP structuré en paquetage

- **La vue d'assemblage des composants logiques** décrit le modèle global du système comme un assemblage de composants inter-connectés.

Travail coopératif

La conception de système à base de composants guidée par notre méthodologie CUP consiste à construire une représentation indépendante d'une technologie dans un formalisme facilitant sa mise en correspondance avec une plate-forme de composants spécifique. Cette représentation se compose des 4 modèles PIM qui résultent de la réalisation d'un des 4 workflows[3] de notre démarche.

Voici une description succincte de ces workflows :

- **Le recueil et la spécification des besoins** consistent à exprimer les fonctionnalités que le système doit offrir aux utilisateurs finaux. Pour cela, les utilisateurs finaux sont catégorisés en acteurs et leurs interactions avec le système en cas d'utilisation. *Le modèle de cas d'utilisation* représente les frontières entre le système et son environnement. Le formalisme utilisé est décrit dans le **méta-modèle de cas d'utilisation**.
- **La réalisation des besoins** consiste à modéliser le comportement interne du système lors de l'exécution de chaque cas d'utilisation. Le comportement interne du système correspond aux interactions entre des éléments de l'architecture logicielle. Un élément peut intervenir dans plusieurs cas d'utilisation, mais il n'y joue pas forcément le même rôle. **La description statique de l'architecture** construit le *modèle statique d'analyse* au fur et à mesure que les cas d'utilisation sont réalisés. Ainsi, lorsqu'un élément de l'architecture n'existe pas, il est créé dans ce modèle. Les différents éléments de l'architecture logicielle, leur structure et leurs relations supportant les interactions du modèle d'interactions, sont décrits grâce au **méta-modèle de conception de composant**.
- **La spécification des composants** permet de représenter les notions de composant et de connexion de composants dans *le modèle statique de composant.* Ce formalisme est décrit dans le **méta-modèle de conception de composant**
- **L'assemblage des composants** utilise les éléments décrits dans le méta-modèle d'assemblage.

[3]Le RUP introduit la notion de workflow pour décrire une séquence d'activités menant à un résultat observable.

Cohérence de l'architecture

Les différentes vues sont destinées à différents intervenants sur le projet, avec leurs propres préoccupations. Cependant, ils coopèrent afin de construire un modèle complet du même système à l'étude. Or, les travaux doivent être cohérents entre eux. Par exemple, un risque est qu'un concept puisse être modélisé dans plusieurs vues sans qu'il soit détecté que deux artefacts ont été créés. Les vues sont cohérentes entre elles grâce à la structure du méta-modèle, à des contraintes et à l'omniprésence de la notion de composant logique. Toute activité a des impacts sur l'architecture du système et implique la réalisation d'autres activités pour maintenir la cohérence. Cette cohérence est garantie par des contraintes entre les vues du modèle. Les contraintes sont soit exprimées dans le méta-modèle, soit ajoutées sur le méta-modèle grâce au langage de contraintes UML : Object Constraint Language [92, 36].

Les sections suivantes du document présentent les vues, sous-parties du méta-modèle CUP, et leurs mises en œuvre dans une proposition de démarche supportée par le méta-modèle. Chacune des sous-parties sont décrites par l'objectif visé, la partie du méta-modèle correspondante, les concepts mis en jeu et leurs mises en oeuvre dans le cadre de la conception d'une étude de cas.

4.2 Les vues du modèle CUP

4.2.1 La vue de cas d'utilisation

Énoncé de l'étude de cas

Une société de location offre des services de location de matériels. Destinée aux employés de la société (les clients n'utilisent pas le système informatique), l'application offre les services suivants :
- Saisie d'une location d'un produit par un client
- Saisie d'un retour d'un produit par un client
- Consultation et modification des informations d'un client
- Consultation et modification du stock et des produits disponibles en stock
- Consultation et modification des prix de location journalière des produits
- Consultation et modification des recettes pour les produits
- Sauvegarde et restauration du stock, des prix, des clients et des recettes

La maquette écran 4.4 présente un écran de saisie d'une location d'un ou plusieurs produits. Ainsi, l'application permettra de consulter les produits disponibles, ainsi que le prix de location afin de louer divers produits. Le système d'information de la société de location comprend deux sites principaux, une agence, et un siège. Le siège contient les bases de données responsables des tarifs et de la recette pour la comptabilité. L'agence contient le côté commercial de la société et reçoit les clients. Chaque agence gère son propre stock de matériel et ses fichiers client (cf. figure 6.13). A première vue, une problématique est que sur un même écran, on doit afficher des informations en provenance d'un serveur local à une agence comme le nombre de matériel d'un type en stock, mais aussi des informations centralisées sur le serveur du siège, comme les tarifs de location.

Objectifs

Les spécifications d'un système s'expriment sous la forme de cas d'utilisation. Nous offrons dans la vue des cas d'utilisation un formalisme permettant de décrire les spécifications d'un système en

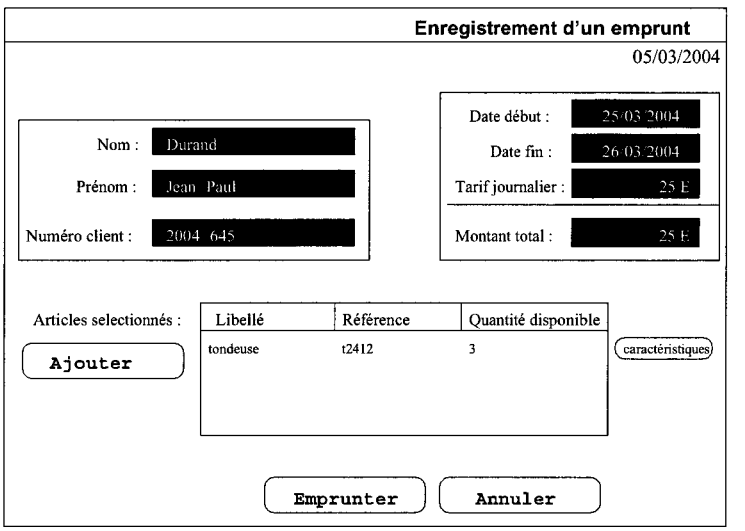


FIG. 4.4 – Maquette écran de l'étude de cas

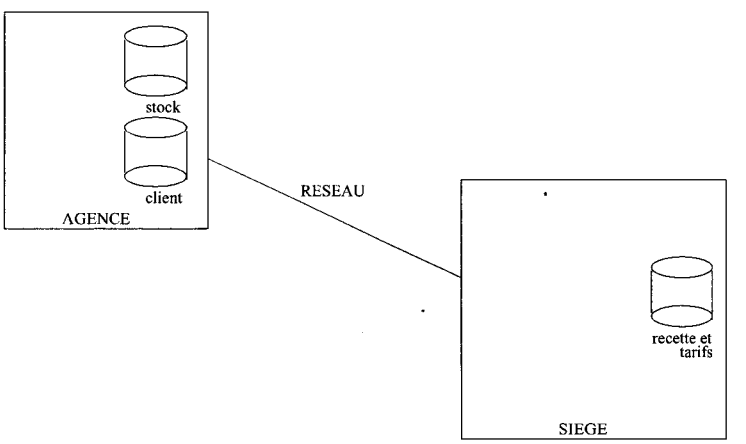


FIG. 4.5 – Architecture physique du système d'information Location de Matériel

terme de cas d'utilisation. La modélisation des cas d'utilisation du système en cours de construction est découpée en composants. Ainsi, la dépendance entre les composants du système est détectée de manière précoce, ce qui autorise la réutilisation des spécifications de composants préfabriqués.

Comme nous l'avons vu dans le chapitre 3, une activité du génie logiciel sur laquelle se basent les autres, consiste à définir les besoins des utilisateurs finaux dans une forme compréhensible par le plus grand nombre. Il n'est pas question, ici, de langage de programmation, les besoins sont exprimés en langage naturel. Les spécifications d'un système sont classiquement découpées en cas d'utilisation. Chaque cas d'utilisation définit une séquence d'interactions entre un ou plusieurs acteurs et le système. Le contenu des cas d'utilisation sert à la fois de base aux activités d'analyse et conception et permet de concevoir les scénarii de test du système. Notre proposition dans cette vue, est d'introduire un découpage du système plus "grossier" que le paradigme orienté objet, en regroupant certains cas d'utilisation dans des **composants logiques**. Ainsi, le problème initial est divisé en problèmes plus simples. De plus, ce découpage permet une véritable réflexion sur les dépendances entre les composants du système. Ce qui est souvent sous-estimé et est répercuté dans les coûts d'intégration. Enfin, les spécifications des composants logiques sont réutilisables pour définir les spécifications d'autres systèmes. Notre modèle définit, dans cette vue habituellement utilisée dans les premières phases d'un processus d'ingénierie, la notion de *composant logique* comme un ensemble de *cas d'utilisation*. Nous considérons, en effet, qu'un composant logique réalise un ensemble de cas d'utilisation et que sa granularité correspond à la notion de modèle (cf. chapitre 3).
Grâce aux cas d'utilisation, le service rendu par le composant peut ainsi être exprimé dans un langage naturel et explicite, ce qui le rend plus facile à réutiliser et lisible par l'utilisateur final. Le modèle CUP oblige la publication de composant logique avec ses documents de spécifications. Ainsi, dès les premières étapes du cycle de développement d'un système à base de composants, il est possible d'identifier des composants préfabriqués qui peuvent répondre aux besoins exprimés.

Description du méta-modèle des cas d'utilisation

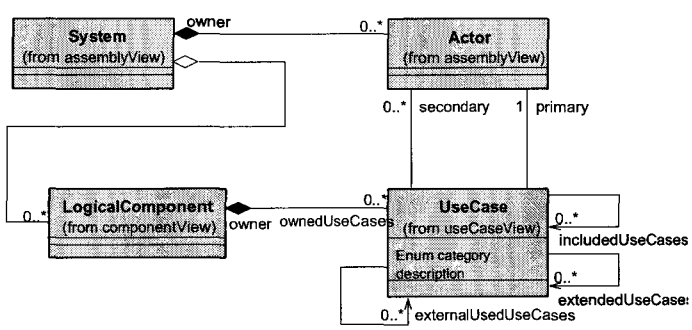


FIG. 4.6 – Méta-modèle de cas d'utilisation

La vue de cas d'utilisation spécifie l'utilisation du système –*System*– par les utilisateurs finaux et d'autres systèmes. Le concept d'acteur –*Actor*– catégorise les utilisateurs finaux et les autres systèmes coopérant avec le système. Un système contient plusieurs acteurs. Les fonctionnalités offertes aux différents acteurs sont catégorisées en cas d'utilisation –*UseCase*– correspondant à la définition donnée dans le chapitre 2. Pour rappel, le cas d'utilisation représente un type d'interaction entre un ou plusieurs acteurs et le système. Un cas d'utilisation a une propriété catégorie –*category*– qui correspond à

un critère permettant le courtage de composants à partir de ses spécifications. Les catégories sont par exemple : Créer, Modifier, Obtenir, Supprimer, ... La propriété de description *–description–* est l'ensemble des documents qui doivent être fournis avec un cas d'utilisation. Ces propriétés garantissent une meilleure documentation. La description d'un cas d'utilisation est développée plus loin dans cette section.

Une première partie de la définition d'un composant logique *–LogicalComponent–* correspondant à cette vue est un ensemble de cas d'utilisation. En effet, un composant logique est un module de modélisation réutilisable qui rend un ensemble de services spécifiés par les cas d'utilisation du composant logique. Les spécifications fonctionnelles d'un système, ensemble de composants logiques, sont exprimées par les cas d'utilisation de ces composants logiques. Un composant logique n'est pas une sous-partie d'un unique système, il peut être réutilisé dans d'autres systèmes. La partie du méta-modèle correspondante (cf figure 4.6) définit les relations entre cas d'utilisation. En plus de relations standards d'inclusion *–includedUseCases–* et d'extension *–extendedUseCases–*, l'ajout de la relation CUP d'utilisation externe *–externalUsedUseCase–* permet d'exprimer qu'un cas d'utilisation utilise un cas d'utilisation d'un autre composant logique. Les deux relations standards sont différenciées. La relation d'inclusion indique qu'un cas d'utilisation inclut les séquences d'interactions d'un autre cas d'utilisation. La relation d'extension étend le comportement du système exprimé dans un autre cas d'utilisation. Le choix de ne pas créer deux relations externes, vient du fait que dans les deux cas, il y a réutilisation des interactions exprimées dans un cas d'utilisation. Et ce qui est intéressant est de montrer le besoin d'un service externe, que ce soit pour étendre un comportement, on inclure un autre comportement. Cela permet de combler le manque des langages de modélisation standards, tels qu'UML, en exprimant la dépendance entre deux composants logiques dans la vue de cas d'utilisation dans un formalisme compréhensible par un utilisateur final.

Ainsi, on exprime au niveau des spécifications du système, les dépendances entre sous-parties de ce système. Pour bénéficier de cette apport, la granularité des composants logiques est forcément celle d'un sous-système.

Contraintes

Nous présentons ici les contraintes liées au méta-modèle CUP. Ces contraintes sont également spécifiées à l'aide de règles OCL pour éviter toute ambiguïté.

- Un acteur n'est contenu que par un seul système (relation UML de composition). En effet, la notion d'acteur est rôle dans le contexte d'un système. Par contre, un composant logique peut appartenir à plusieurs systèmes (relation UML d'agrégation). Il peut même être partagé par plusieurs systèmes.

```
Context Actor inv :
  self.owner.size()=1
```

- Un UseCase n'a qu'un seul acteur primaire. Celui à l'initiative du cas. Il peut par contre avoir zéro ou plusieurs acteurs secondaires. Ils servent à l'exécution du cas d'utilisation.

```
Context UseCase inv :
  self.primary.size()=1
```

- La relation externalUsedUseCases relie des cas d'utilisation de composants différents.

```
Context UseCase inv :
  self.externalUsedUseCases (ext | ext.owner!=self.owner)
```

- La relation includedUseCase relie des cas d'utilisation du même composant.

```
Context UseCase inv :
  self.includedUseCases (ext | ext.owner=self.owner)
```

- La relation extendedUseCase relie des cas d'utilisation du même composant.

```
Context UseCase inv :
  self.extendedUseCases (ext | ext.owner=self.owner)
```

Vue des cas d'utilisation de l'étude de cas

L'activité de recueil et expression des besoins consiste à identifier les acteurs, les fonctionnalités offertes aux acteurs par le système, décrire les cas d'utilisation de manière textuelle, c'est-à-dire, la totalité des scénarii d'utilisation. Dans l'étude de cas, les acteurs identifiés sont les *Vendeurs* et les *Administrateurs*. Les fonctionnalités offertes à ces acteurs sont décrites par les cas d'utilisation de la figure 4.7. La représentation des diagrammes est empruntée à UML et enrichie d'éléments tels que le conteneur de cas d'utilisation qui représente un composant logique. Un profil UML pourrait être défini plus formellement à ce niveau afin d'obtenir un outil compatible UML [79]. A partir des cas d'utilisation, on identifie les composants logiques, regroupement logique de cas d'utilisation (cf fig. 4.7) basé sur la notion de collaboration (cf. chapitre 3). C'est une identification précoce par les services exprimés dans les cas d'utilisation. Dans l'exemple, on identifie les composants logiques *Magasin* et *Siege*.

La figure 4.7 est une représentation de la vue de cas d'utilisation de l'étude de cas :

Description des cas d'utilisation

Les cas d'utilisation doivent être décrits dans un langage naturel. La description de cas d'utilisation doit être simple. Elle contient un ensemble de scénarii, qui sont les chemins d'exécution du cas d'utilisation. Les pré-conditions sont des gardes qui garantissent que l'état du système permet de commencer le cas d'utilisation. Les post-conditions spécifient l'état du système après les interactions décrites dans le cas d'utilisation. Ces dernières permettent de vérifier que le système a fonctionné correctement. Des scénarii d'exception permettent d'exprimer un arrêt du cas d'utilisation qui ne s'est pas déroulé juste qu'au bout. Voici, un exemple de description de cas d'utilisation.

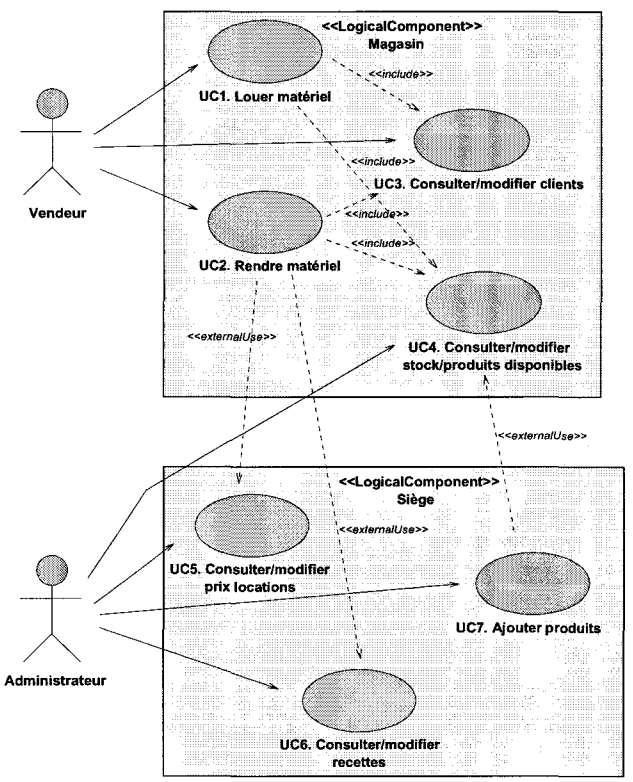


FIG. 4.7 – Vue de cas d'utilisation

Description du cas d'utilisation Rendre matériel

Acteur principal Vendeur
Partie prenante Client

Préconditions
...

Scénario nominal
N1. Le vendeur saisit le client
N2. Le système affiche la liste de matériel emprunté
N3. Le vendeur indique le matériel à rendre
N4. Le système indique le prix de la location
N5. Le vendeur indique que la location est payée
N6. Le système modifie le stock et la fiche client
N7. Le système modifie la recette

Postconditions
Le quantité de cet article est incrémenté dans le stock
La recette est incrémentée du tarif de la location

Scénario alternatif
A1 : Le matériel à rendre n'est pas emprunté
(commence à l'étape N4 du scénario nominal)
A1-1. Le système indique que le matériel à rendre n'est pas emprunté
(le scénario nominal reprend à l'étape N3.)

Scénario d'exception
E1 : La location n'est pas payée
(commence à l'étape N5 du scénario nominal)
E1-1. Le vendeur indique que la location n'est pas payée
(le scénario s'arrête)

L'ensemble des scénarii des cas d'utilisation d'un composant servira à analyser les interactions entre les éléments de son architecture, afin de répondre correctement aux besoins des utilisateurs. Les interactions sont modélisées dans la vue d'interactions.

4.2.2 La vue d'interactions

Objectifs

Un cas d'utilisation correspond à un ensemble de scénarii qui décrivent les interactions entre les acteurs et le système dans le cadre de ce cas d'utilisation. La vue d'interactions permet de représenter

les **collaborations**, c'est-à-dire, les interactions entre les éléments du système qui participent au scénario. En terme objet, cela permet de représenter les messages et donc les appels de méthodes entre les objets, et ainsi de définir les interfaces des classes définissant le système. L'activité d'analyse des interactions consiste à modéliser le comportement interne du système pendant l'exécution de chaque cas d'utilisation. Le comportement interne du système est exprimé par les interactions entre les éléments de l'architecture logicielle. Un cas d'utilisation peut être exécuté de plusieurs manières : les scénari. Chaque cas d'utilisation est réalisé par un ensemble d'éléments de l'architecture. Le modèle d'interactions permet de représenter les rôles que jouent les éléments de l'architecture dans les scénarios. Un élément de l'architecture joue un rôle dans plusieurs cas d'utilisation, mais il ne joue pas forcément le même rôle.

Description du méta-modèle d'interactions

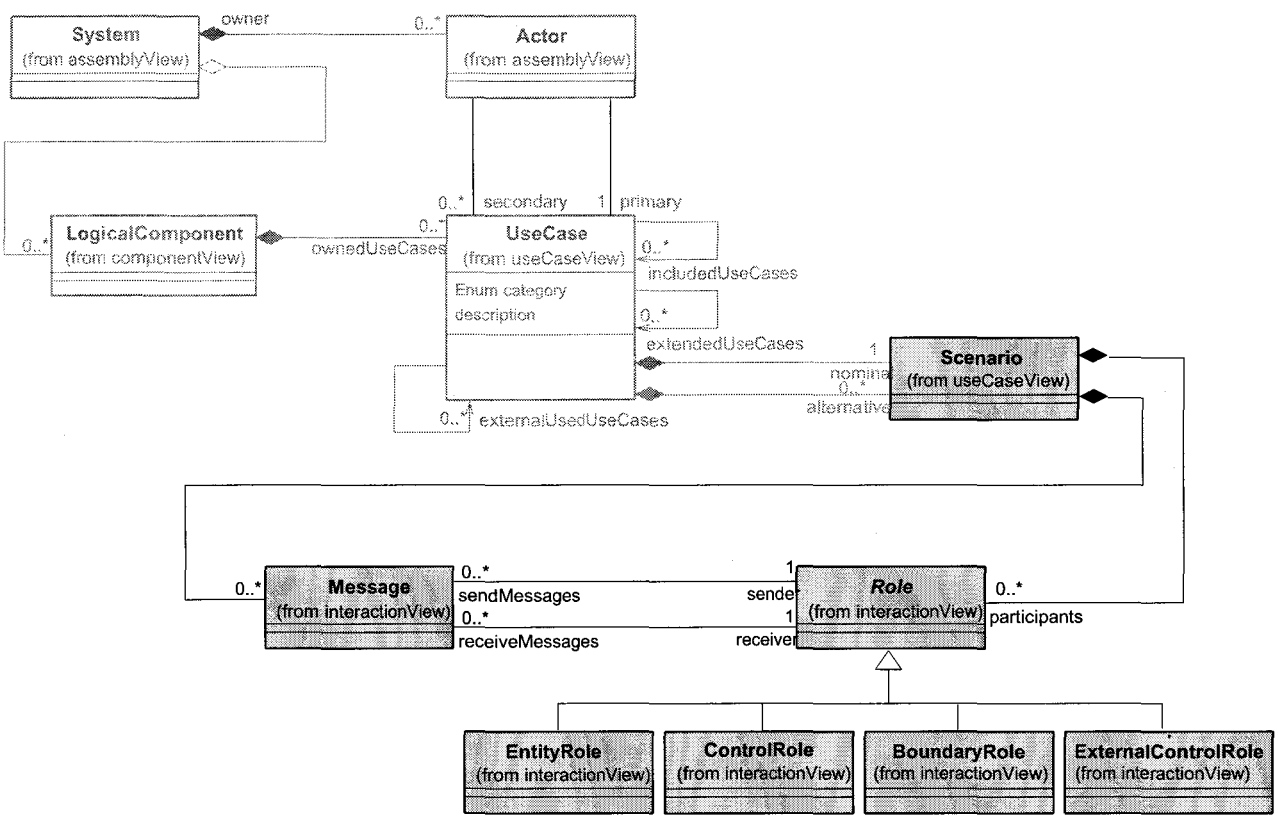


FIG. 4.8 – Méta-modèle d'interactions

Un cas d'utilisation est spécifié par un certain nombre de scénarii *–Scenario–*. Il y a un unique scénario nominal *–nominal–*. Il décrit la séquence d'interactions permettant à l'acteur d'atteindre l'objectif décrit par le cas d'utilisation. Les autres scénarii sont des scénarii alternatifs *–alternative–* au cas nominal. Un scénario est une séquence d'interactions, qui sont décomposées en une séquence de messages *–Message–* entre les éléments du système appelés rôles *–Role–*. La notion de rôle correspond au rôle tenu par un élément du système dans le contexte d'un scénario. Un élément du système peut ainsi être participant à des scénarios différents, au même titre qu'il peut participer à plusieurs collaborations

(cf. chapitre 3). La notion de rôle est abstraite et il existe plusieurs types concrets de rôle, les trois premiers étant standards. Le rôle entité *–EntityRole–* correspond au rôle tenu par un élément dont la durée de vie est plus longue que celle de l'application. Le rôle de contrôle *–ControlRole–* correspond au rôle tenu par un élément dont la durée de vie est de la durée d'une transaction, celle d'un scénario. Le rôle frontière *–BoundaryRole–* représente les points d'entrée dans le système via lesquels les acteurs interagissent avec le système. Nous proposons l'ajout du rôle de contrôle externe *–ExternalControlRole–* pour exprimer dans une interaction qu'un rôle est nécessaire à un scénario mais n'appartient pas au composant. Cette interaction nécessite alors la collaboration d'un autre composant.

Cette vue consolide les choix de découpage du système en composants, ainsi que les dépendances entre les composants exprimées grâce à la relation externalUse. Les dépendances sont exprimées ici par des envois de message et les récepteurs de message ont une granularité d'objet. On applique ici les principes de "bon découpage" présentés dans le chapitre 3. Les rôles interagissent par envoi et réception de messages. Un message reçu par un rôle est une responsabilité de celui-ci vis-à-vis de l'expéditeur du message. Il correspond à une méthode dans le type du rôle récepteur du message. Ainsi, cette vue favorise la définition des interfaces. Les rôles de contrôle externes ne sont typés que par des interfaces requises (définies dans la vue suivante), puisque leurs responsabilités correspondent aux responsabilités d'un élément externe au composant. Parmi les types de rôle d'analyse standards (cf fig. 4.8), l'ajout du rôle *externalControlRole* permet de donner une description plus fine et dynamique de la dépendance entre composants au niveau méthode. Un message envoyé à ce type de rôle est en fait un appel de service externe au composant, et définit une partie de l'interface exprimant les services **requis** de ce composant.

Contraintes

– Un cas d'utilisation n'a qu'un seul scénario nominal. Il correspond au scénario le plus pertinent. Par contre, il peut avoir plusieurs scénarii alternatifs.

```
Context UseCase inv :
  self.nominal.size()=1
```

– L'utilisation d'un rôle de contrôle externe implique qu'il existe une relation externalUse entre le composant et un autre, et que le rôle est issu d'un scénario du cas d'utilisation en question.

```
Context UseCase inv :
  self.externalUsedUseCase->forAll(
    nominal.participants->select(r |
      r.oclIsTypeOf(ExternalControlRole))->notEmpty()
    or
    alternatives->participants->select(r |
      r.oclIsTypeOf(ExternalControlRole))->notEmpty()
  )
```

– Un rôle est une classe abstraite.

```
Context Role inv :
  self.isAbstract=true
```

Vue des interactions de l'étude de cas

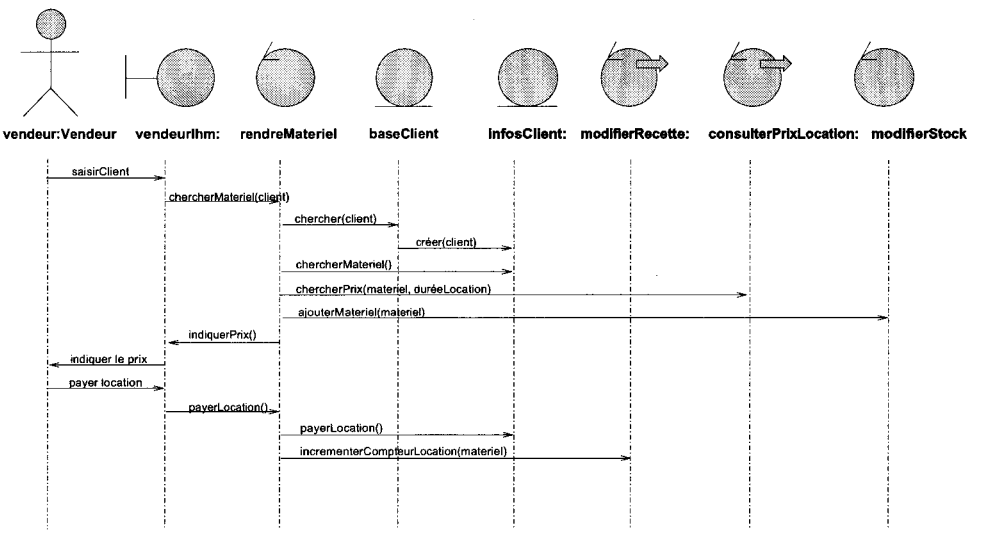
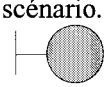


FIG. 4.9 – Vue partielle des interactions

On utilise des diagrammes de type diagrammes de séquences UML (cf fig. 4.9) pour représenter les interactions entre les rôles pendant l'exécution du scénario nominal du cas d'utilisation *Rendre matériel*. De gauche à droite sur la figure 4.9, les symboles correspondant aux différents rôles concrets sont présentés :

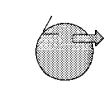
- l'acteur *Vendeur* n'est pas un rôle, mais il figure sur le diagramme pour illustrer les messages échangés entre le système et son environnement dans le cadre du scénario.
- le rôle frontière *vendeurIhm* modélise l'interface homme-machine de l'acteur vendeur qui lui permet d'interagir avec le système.

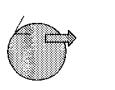


- le rôle de contrôle *rendreMateriel* encapsule le traitement nécessaire pour exécuter le scénario.

- les rôles entités *baseClient* et *infosClient* contiennent des données et traitements métier utiles au scénario.

- les rôles de contrôle externe *modifierRecette* et *consulterPrixLocation* permettent des appels de services à d'autres composants.





Les flèches modélisent les messages échangés entre rôles et disposés de manière chronologique de haut en bas.

La réalisation des cas d'utilisation est découpée en deux points : rechercher les éléments de l'architecture et les relations entre eux, provenant de l'analyse du métier et les instancier pour jouer un rôle dans chaque scénario de cas d'utilisation pour renseigner leurs caractéristiques (propriétés et traitements). La dépendance entre composants est précisée, elle est exprimée par des messages qui guident la définition d'interfaces aux composants. Ainsi, l'ensemble des messages reçus par un rôle contrôle

externe peut constituer une interface requise du composant. La vue suivante permet de représenter les interfaces issues de cette vue, et le type des éléments de l'architecture des composants.

4.2.3 La vue de conception de composants

Objectifs

La conception de composants permet de représenter de manière statique la structure interne et la structure externe des composants. C'est une vue en boîte blanche. Ainsi, il est possible d'exprimer les services rendus par le composant et les services dont il a besoin, tout en restant indépendant d'autres composants. Ainsi, il est plus réutilisable. Le méta-modèle introduit dans la vue de conception de composants les éléments d'architecture et leurs structures. Leurs relations supportent les interactions de la vue d'interaction. Un composant offre des services et a besoin de services offerts par d'autres composants. Le composant fournit des interfaces qui exposent les méthodes métiers. Ils définissent les interfaces fournies. Ils définissent un ensemble de comportements. Un composant possède plusieurs interfaces, qui permettent d'organiser les méthodes métiers selon les points de vue des clients du composant.

Description du méta-modèle de conception de composants

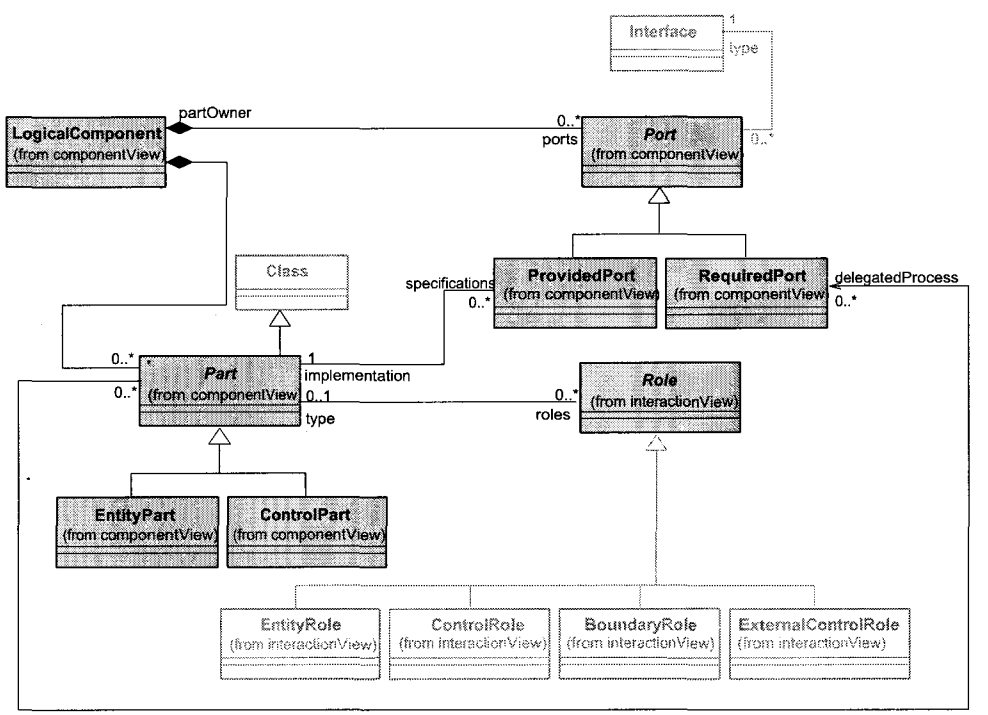


FIG. 4.10 – Méta-modèle de composants

Cette vue donne une nouvelle facette à la définition d'un composant logique *–LogicalComponent–*. La structure interne d'un composant logique est constituée de parties[4] *–parts–*. La structure interne d'un composant logique est modélisée par la notion abstraite de parties. Une partie a deux définitions concrètes. Elle est entité *–EntityPart–* lorsqu'elle a une durée de vie indépendante de l'application. Elle contient le traitement métier et les informations contenues par le composant. Une partie est de contrôle *–ControlPart–* quand son cycle de vie est dépendant d'une application, ou d'une transaction (séquence d'interactions). Elle peut implémenter les interfaces fournies. Le concepteur de composant peut choisir si une interface fournie est implémentée par une ou plusieurs parties contrôles.

La définition externe de composant permet grâce à deux types d'interfaces de modéliser les services offerts du composant et les services qu'il requiert pour fonctionner. Un port *–Port–* est une notion abstraite qui correspond à une interface de composant. Un port est fourni *–ProvidedPort–* quand il définit les services rendus par le composant, et requis *–RequiredPort–* quand il définit les services nécessaires au fonctionnement du composant et offerts par d'autres composants.

Un rôle de collaboration, défini dans la vue d'interactions, est typé par une partie *–type–*. La relation entre partie et rôle est une relation de type type-instance. Étant donné que les langages utilisés dans la plupart des plates-formes technologiques cibles sont des langages de classes, la méta-classe d'une partie étend la méta-classe *class* et hérite donc des relations d'associations, héritage, ... Une partie peut typer un ou plusieurs rôles *–roles–* participants à plusieurs collaborations et donc présents dans plusieurs vues d'interactions. Les rôles de contrôle sont typés par des parties de contrôle, et les rôles entité, par des parties entité. La cohérence entre les spécifications du composant (vue de cas d'utilisation et documents associés) et sa définition (vue de composants) est assurée par la mise en jeu de ses parties et ses ports dans les scénarios de cas d'utilisation sous la forme de *Role*.

Contraintes

– Un part est une classe abstraite

```
Context Part inv :
  self.isAbstract=true
```

– Un composant logique contient au moins un port fourni.

```
Context LogicalComponent inv :
  self.ports->select(p | p.oclIsTypeOf(ProvidedPort))->notEmpty()
```

– Un port fourni doit être directement relié par une partie de contrôle

```
Context ProvidedPort inv :
  self.implementation.oclIsTypeOf(ControlPart)
```

– Les rôles entités sont typés par des parties entités.

```
Context EntityRole inv :
  self.type.size()=1 implies self.type.oclIsTypeOf(EntityPart)
```

[4]Bien que le méta-modèle CUP utilise des termes qui seront employés par la future norme UML2.0, qui doit introduire également des composants logiques, CUP et UML2.0 restent deux formalismes distincts

– Les rôles de contrôle sont typés par des parties de contrôle.

```
Context ControlRole inv :
  self.type.size()=1 implies self.type.oclIsTypeOf(ControlPart)
```

Vue de conception de composants de l'étude de cas

Cette vue de conception de composant décrit les structures internes et externes du composant *Magasin*. Il possède deux ports fournis *LouerRendreMateriel* et *AjouterRetirerProduit* et deux ports requis *ConsulterPrixLocation* et *GestionRecette*. Le port *LouerRendreMateriel* est implémenté par la partie de contrôle *LocationManager*, le port *AjouterRetirerProduit*, par la partie de contrôle *StockManager*. *GestionnaireMateriel* est une partie de contrôle spécialisée dans l'ajout et retrait de matériel du stock. Les deux parties entités gèrent les informations concernant les clients. Enfin, du traitement est délégué à d'autres composants, qu'il est nécessaire de trouver et de connecter. Ce besoin de services du composant est modélisé par deux ports requis *ConsulterPrixLocation* et *GestionRecette*. Le lien de délégation vient du fait que l'on délègue du traitement. La gestion de la cohérence globale du modèle est assurée par la vérification des contraintes concernant les relations entre parties et ports de cette vue et les rôles de la vue d'interactions.

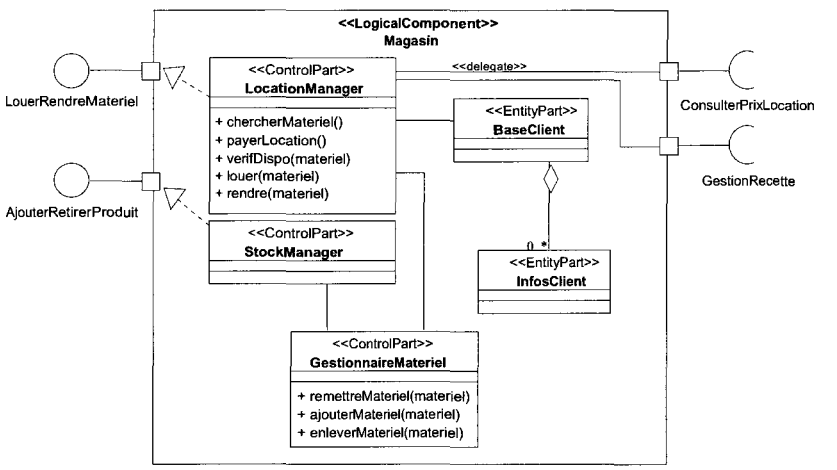


FIG. 4.11 – Vue de conception de composant

4.2.4 Vue d'assemblage du système

Objectifs

La vue d'assemblage permet de concevoir globalement un système par assemblage de ses composants. Les composants sont alors vus en boîtes noires, et ils sont reliés par des connexions.

Description du méta-modèle d'assemblage de composants

Le méta-modèle de description d'assemblage définit la notion de système –*System*–. Un système est modélisé par l'assemblage des composants logiques le constituant. Il peut y avoir plusieurs assemblages possibles. Un composant logique –*LogicalComponent*– est vu en boîte noire, seule figure dans

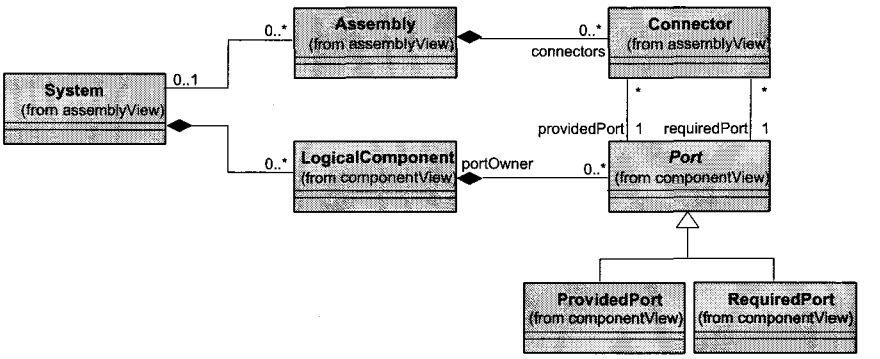


FIG. 4.12 – Vue d'assemblage

cette vue sa structure externe –*port*–. Deux composants sont connectés par la connexion de leurs ports. Un système est construit selon des assemblages de composants –*Assembly*–, c'est-à-dire l'ensemble des connecteurs –*Connector*– du système. Un connecteur relie logiquement deux ports de nature différentes : port fourni –*providedPort*– et port requis–*requiredPort*–. Dans un souci de maintien de la cohérence entre les vues, on convient que :

- un port requis et un connecteur correspondent à un rôle de contrôle externe dans la vue d'interaction.
- une partie de contrôle et un port fourni correspondent à un rôle de contrôle et un rôle frontière dans la vue d'interaction.

Contraintes

- Un port est une classe abstraite.

```
Context Port inv :
  self.isAbstract = true
```

- Un connecteur relie un port requis à un port fourni.

```
Context Connector inv :
  self.providedPort.size()=1 and self.requiredPort.size()=1 and
  self.providedPort.oclIsTypeOf(ProvidedPort) and
  self.requiredPort.oclIsTypeOf(RequiredPort)
```

- Un connecteur ne peut relier un port fourni à un port requis d'un même composant

```
Context Connector inv :
  self.providedPort.portOwner <> self.requiredPort.portOwner
```

- Pour un même assemblage, un port ne peut être relié qu'une seule fois

```
Context Assembly inv :
  self.connectors->forAll (c1, c2 | c1 <> c2 implies
    c1.providedPort <> c2.providedPort and
    c1.requiredPort <> c2.requiredPort
  )
```

Vue d'assemblage de l'étude de cas

Pour ses traitements, le composant peut avoir besoin de services offerts par d'autres composants. Dans l'étude de cas, la consultation ou modification de recette : connecteur *Magasin : :ConsulterRecette-Siege : :ConsulterRecette* est un exemple. Le rôle de contrôle externe de la vue d'interaction permet l'identification de services délégués à d'autres composants.

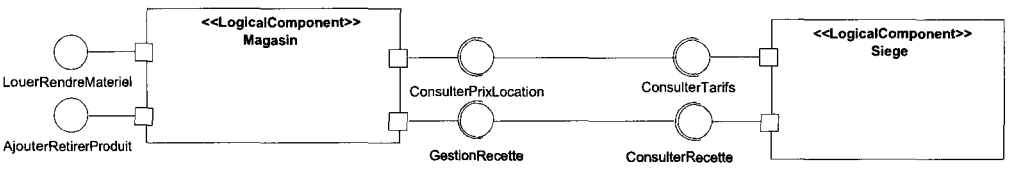


FIG. 4.13 – Vue d'assemblage

4.3 Outil de modélisation à base de composants logiques : CUPTool

CUPTOOL est un atelier de conception à base de composants et un générateur de composants vers des plates-formes technologiques actuelles [16]. Son architecture a été conçue à partir du méta-modèle CUP. Il supporte la démarche CUP. Pour cela, il permet d'utiliser le formalisme à base de composants logiques défini par le méta-modèle CUP, de type *Platform Independent Model* , permettant de modéliser un système à base de composants avec le maximum de caractéristiques tout en restant indépendant d'une plate-forme technologique. Ensuite, il permet de générer un modèle *Platform Specific Model* sur la plate-forme technologique EJB. Il permet de capitaliser sur l'analyse et améliore la qualité du logiciel.

Développé et exploité dans l'environnement open-source ECLIPSE[5], CUPTool utilise le plugin ECLIPSE MODELING FRAMEWORK[6]. Ce plugin est un framework de modélisation et de génération de code basé sur un modèle de données structurées. Cet outil, permet rapidement, et de manière évolutive, de générer la fonctionnalité de modélisation de l'atelier CUPTool. Le concepteur de systèmes à base de composants a ainsi à sa disposition, un formalisme précis, contraint par des règles exprimées dans le méta-modèle et vérifiées lors de la modélisation. La figure 4.14 montre l'ajout d'un élément à un composant. En sélectionnant un élément *Component*, l'outil propose au concepteur une liste d'éléments qui correspondent aux liens de composition entre la méta-classe `Component` et les méta-classes de ces éléments (fig. 4.14). Enfin, l'outil autorise le travail collaboratif de concepteurs ayant des préoccupations différentes et travaillant sur le même système. La cohérence des différents travaux est maintenue à chaque insertion par les contraintes dans un fichier XMI qui fait office de référentiel.

L'outil CUPTool offre les vues offertes par le méta-modèle CUP, aux différents types d'intervenants du projet.

[5] *http ://www.eclipse.org*
[6] *http ://www.eclipse.org/emf*

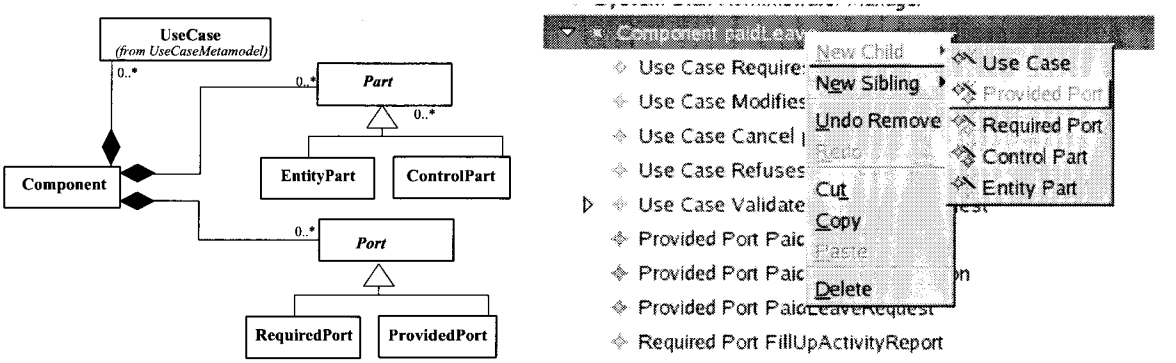


FIG. 4.14 – Extrait du méta-modèle de composant et vue de l'atelier correspondante

4.3.1 Le standard de la Méta-modélisation : MOF 2.0

Définition de MOF 2.0

Le MOF ou Meta Object Facility [29] est le standard défini par l'OMG pour la méta-modélisation. C'est un framework dédié à la spécification, conception, échange et intégration de méta-données[7], dirigés par les modèles. Il existe de nombreuses sources de méta-données et de nombreux méta-modèles. Le MOF offre un langage pivot pour garantir l'interopérabilité, la fédération d'outils, de sources de données, de faciliter le développement de méta-modèles, outils, middlewares. La figure 4.15 représente les transformations du modèle d'un système exprimé en UML vers un fichier XMI pour le sérialisé, ou en utilisant JMI afin de manipuler les méta-data du modèle, ou en IDL pour son exploitation dans un environnement de développement distribué.

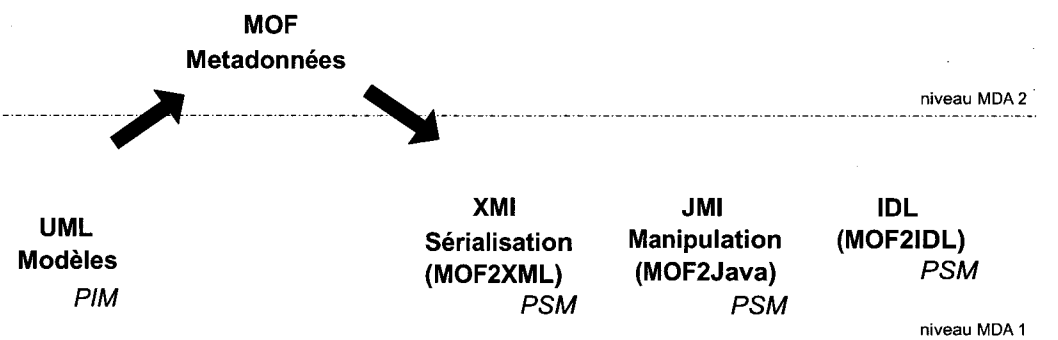


FIG. 4.15 – MOF 2.0

Essential MOF

Essential MOF (EMOF) est une restriction des spécifications complètes MOF 2.0 utilisée pour définir des méta-modèles simples en utilisant des concepts simples. EMOF utilise des concepts orientés objets comme les classes, attributs, ... MOF 2.0 et UML 2.0 partageant le diagramme de classes, il est possible de définir des méta-modèles MOF avec les outils UML du marché.

[7]Le terme méta-donnée correspond au concept d'information sur l'information.

L'architecture EMOF est décrite dans la figure 4.16.

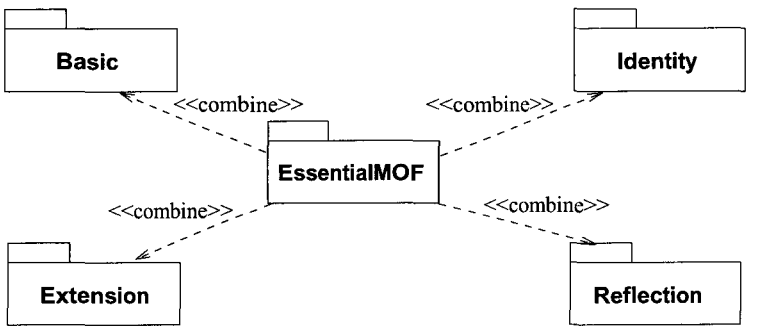


FIG. 4.16 – Architecture EMOF 2.0

Le paquetage EMOF est la fusion[8] des paquetages MOF 2.0 :
- **Basic**, éléments basiques de UML 2.0.
- **Identity** qui fournit les mécanismes permettant d'identifier des objets sans ambiguïté.
- **Reflection** qui permet la manipulation et découverte de méta-objet sans connaissance d'un méta-modèle a priori.
- **Extension** qui permet l'ajout dynamique des informations à des méta-données.

EMOF est une partie de MOF et en cela, un modèle EMOF peut être sérialisé, manipulé, ...

Concrètement, nous utilisons cette technologie grâce à l'environnement Eclipse et notamment le plugin **Eclipse Modeling Framework**.

4.3.2 Eclipse Modeling Framework

L'outil de développement Eclipse [22], Open-source, supporté par IBM est aujourd'hui le plus populaire. Son architecture à base de plugins permet de l'étendre et de le spécialiser pour divers langages et divers usages. Le plugin EMF étend l'éditeur Eclipse pour la génération de code dirigée par les modèles.

Ce framework open-source permet de construire des applications Java par génération de code à partir de modèles. EMF fournit son propre méta-modèle, nommé **Ecore**, pour décrire des modèles de données d'applications, appelés **core models**, un sérialiseur XMI pour stocker des modèles, un outil pour transformer des modèles écrits en UML, XML Schema ou interfaces Java en Modèle Ecore, enfin, un outil de génération qui permet de générer du code de qualité depuis une description de modèle en Ecore afin de manipuler ce modèle.

EMF est un outil qui s'inscrit dans l'initiative **Model Driven Architecture** de l'OMG. Il prend en entrée des modèles et génère du code exécutable. Son méta-modèle Ecore, conçu à partir de MOF et quasi-équivalent [58], est proche de **EMOF** (Essential MOF), sous-ensemble du standard MOF 2.0.

[8]La relation de dépendance stéréotypée *combine* correspond à l'union des concepts présents dans les différents paquetages combinés

Le modèle Ecore (cf figure 4.17) évolue en parallèle du MOF et tend à converger prochainement avec MOF 2.0. De plus, des outils existent qui permettent la création de gestionnaires de métadonnées MOF. Notamment, l'outil RAM3 [26] permet en plus l'administration et le prototypage et permet de manipuler tous les niveaux de méta-modélisation.

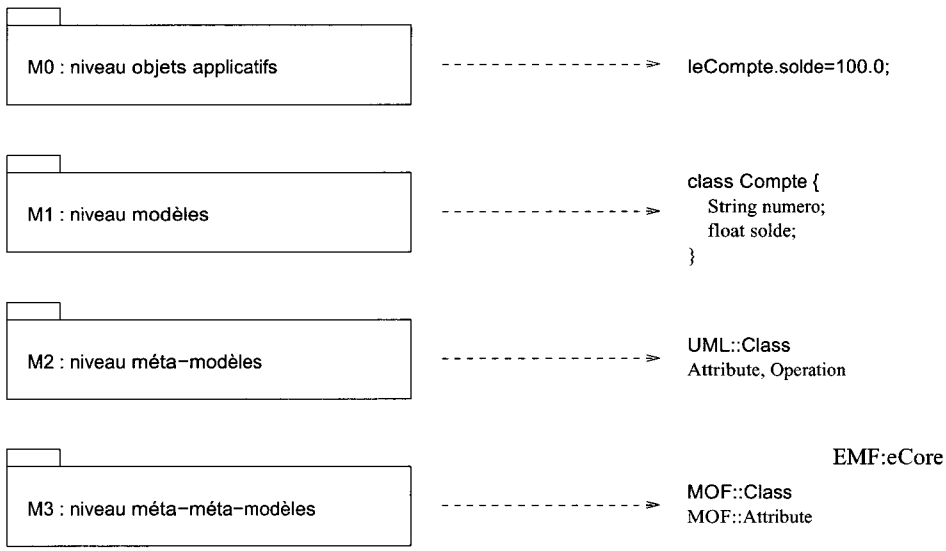


FIG. 4.17 – eCore

Le framework

EMF est un framework java et un mécanisme de génération de code permettant de construire des outils et autres applications basées sur un modèle structuré. Ce modèle peut être décrit en UML, XML ou à l'aide d'interfaces Java. Les modèles EMF peuvent être définis grâce à des outils UML. Ces modèles sont importés dans EMF qui ajoute automatiquement du code de manipulation. Ainsi, cette approche permet de bénéficier de la modélisation et de la génération de code. Les modèles peuvent aussi être importés dans un format XML compatible avec le formalisme EMF Ecore. Et inversement, l'outil EMF génère des documents XMI pour sérialiser et stocker les modèles.

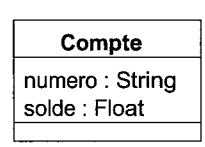


FIG. 4.18 – Exemple de concept

Sérialisation de modèles

EMF utilise le format d'échange XML Metadata Interchange (XMI 2.0) pour sérialiser les modèles Ecore au format XML. Le framework EMF permet l'utilisation d'autres formats de sérialisation.

```
<ecore:EPackage xmi:version=''2.0'' xmlns:xmi=''http://www.img.org/XMI''
  xmlns:xsi=''http://www.w3.org/2001/XMLSchema-instance''
```

```
  xmlns:ecore=''http://www.eclipse.org/emf/2002/Ecore''
  name=''banque'' nsURI=''http://banque.xmi'' nsPrefix=''banque''>
  <eClassifiers xsi:type=''ecore:EClass'' name=''Compte''>
    <eAttributes name=''numero''
      eType=''ecore:EDataType
      http://www.eclipse.org/emf/2002/Ecore#//EString''/>
    <eAttributes name=''solde''
      eType=''ecore:EDataType
      http://www.eclipse.org/emf/2002/Ecore#//EInt''/>
  </eClassifiers>
</ecore:EPackage>
```

Manipulation de modèles

JMI (Java(TM) Metadata Interface) est une projection du OMG MOF 1.4 vers le langage Java. JMI est orienté et optimisé pour les annuaires de méta-données. La projection EMF vers Java est optimisée pour réduire les besoins en mémoire.

```
public interface Compte extends EObject
{
  String getNumero();
  void setNumero(String value);

  float getSolde();
  void setSolde(int value);
}

public class CompteImpl extends EObjectImpl implements Compte
{
  ...
  protected static final float SOLDE_EDEFAULT;
  public float getSolde()
  {
    return solde;
  }
  public void setSolde(float newSolde)
  {
    float oldSolde=solde;
    solde=newSolde;
    if (eNotificationRequired())
      eNotify(new ENotificationImpl(this, ..., oldPages, pages));
  }
  ...
}
```

4.3.3 Le modeleur CUP

Le modeleur CUP a été conçu sur ce principe grâce à EMF. Le méta-modèle a été fourni à l'outil sous forme de schéma de classes. Puis la génération du code de manipulation a été faite, enfin la génération d'un plugin proposant une interface de manipulation de modèle dans Eclipse.

Fourniture du méta-modèle CUP

La première étape est la fourniture du modèle, en UML, XML Schema ou interfaces Java. Puis, grâce à un assistant, il est transformé en modèle core et en modèle générateur. Ainsi, il est possible d'utiliser le générateur EMF pour modifier le modèle générateur et générer le code. A ce moment, il est possible de modifier le code Java et le modèle. EMF peut reconnaître certains changements dans le code et les répercuter sur le modèle.

Le modeleur CUP permet de créer des modèles CUP. Ce modeleur est généré dans un plugin Eclipse. Un exemple sera développé dans cette page, afin de présenter la modélisation CUP.

Dans chaque paquetage CUP, il y a une classe Factory qui contient les méthodes de fabrique pour tous les éléments contenus dans ce paquetage. Par exemple, voici la création d'un composant :

```
component.Component comp =
   component.ComponentFactory.eINSTANCE .createComponent();
comp.setName("conges");
syst.getComponents().add(comp);
```

En plus, une étape permet de récupérer une collection de composants du système et d'y ajouter le nouveau composant. Par exemple, voici l'ajout de cas d'utilisation à un composant :

```
useCase UseCase uc1 = useCase.UseCaseFactory.eINSTANCE.createUseCase();
uc1.setName("Requires paid leave request");
comp.getComponents().add(uc1);

useCase UseCase uc2 = useCase.UseCaseFactory.eINSTANCE.createUseCase();
uc2.setName("Validates paid leave request");
comp.getComponents().add(uc2);

...
```

Un éditeur de type manipulation d'arbre XML est généré par défaut dans le plugin. Il permet de manipuler un modèle CUP dans Eclipse. Une vue graphique du modeleur CUP est en cours de développement. Cet outil fournira une notation graphique respectant la même sémantique.

4.4 Synthèse du processus Unifié à base de composants logiques

Voici une liste des critères de qualité améliorés par le processus unifié à base de composant :

- **Lisibilité** : Les différentes parties d'un système sont clairement représentées. Les frontières des composants logiques sont représentées dans chaque vue. Cela réduit l'espace de travail et permet une meilleure lisibilité du modèle du système. Chaque intervenant a une vue sur le système.
- **Réutilisabilité** : Le concept de connecteur améliore la réutilisabilité des composants du système. La réutilisation est ainsi explicite.
- **Traçabilité** : La traçabilité est assurée par le méta-modèle, ainsi que les règles présentées.

- **Cohérence de l'architecture** : L'architecture est cohérente grâce à l'omniprésence des composants logiques, conteneurs des différents éléments de l'architecture. La notion de composant est omniprésente dans toutes les vues. Cela favorise la cohérence des vues.
- **Évolution du logiciel** : L'évolution du logiciel, et notamment l'ajout d'un service peut être réalisé soit par modification d'un composant soit par connexion d'un nouveau composant responsable de ce service. En l'occurrence, l'amélioration de l'adaptabilité d'un système est garantie par l'identification du composant à modifier, et donc l'évolution a des répercutions plus localisées.
- **Courtage de composant** : La définition de composant est plus précise. Chaque intervenant d'un projet a une définition en cohérence avec son point de vue, y compris l'utilisateur final. Ainsi, plusieurs qualités d'intervenant peuvent intervenir dans la recherche d'un composant préfabriqué ou la décision d'implémenter un nouveau composant.

Le processus CUP est dirigé par les modèles et à base de composants. Les sous-modèles sont liés par le concept central de composant logique. La granularité du composant logique est celle d'un modèle plutôt que celle d'une classe enrichie. Le modèle de composants logiques omniprésents améliore la traçabilité. L'identification des composants est précoce. La définition des composant basée sur les cas d'utilisation améliore la détection et la réutilisation de composants logiques préfabriqués. L'avantage de notre approche est qu'elle n'impose pas un processus strict. Par contre, elle fournit un ensemble de règles de dépendance entre les vues du modèle d'un système. Pour cela, nous avons défini le méta-modèle CUP qui fournit un formalisme pour chaque vue et offre un cadre de travail différent à chaque intervenant tout en maintenant une cohérence globale.

Un outillage a été développé afin de mettre en oeuvre la modélisation de systèmes dans le formalisme CUP. Cela permet de valider, à la fois le modèle et son utilisation pour améliorer le développement d'applications de gestion dans des technologies à base de composants. De plus, le choix et l'exploitation de cette technologie permettent de concevoir un outil extensible, comme par exemple, l'extension graphique de l'outil. Le chapitre suivant développe l'extension permettant de valider un prototype de projection vers une plate-forme spécifique. Il démontre expérimentalement que le passage du modèle abstrait CUP reste complexe, tout en étant plus direct qu'en partant directement d'un modèle UML standard. La nécessité d'un modèle intermédiaire est prouvée expérimentalement.

Chapitre 5

Projection vers une plate-forme technologique : application aux EJB

Un processus Model Driven Architecture est la construction d'un système par la production d'un modèle indépendant d'une plate-forme et une ou plusieurs projections sur une plate-forme technologique spécifique. Cette dernière opération peut être automatisée grâce à un mécanisme de transformation du modèle PIM en un ou plusieurs modèles spécifiques. Cette démarche permet de capitaliser sur les activités d'analyse et de conception. Le système d'information devient plus facilement évolutif. La sur-spécification du système est ainsi amortie par des projections vers différentes technologies. Cependant, les réalisations MDA sont actuellement peu nombreuses. Ce chapitre propose une implémentation MDA, la démarche CUP. CUP consiste en la projection d'un modèle PIM, exprimé dans le formalisme CUP selon la démarche sous-jacente, vers un modèle PSM, la technologie choisie étant les Enterprise Java Beans. La contribution présentée dans ce chapitre est une chaîne de production d'applications à base de composants selon une approche MDA.

Les choix d'implémentation d'un modèle à base de composants indépendant d'une plate-forme (PIM - Platform Independent Model) projeté vers une plate-forme spécifique (PSM - Platform Specific Model) sont multiples et complexes. Nous avons choisi la plate-forme Enterprise Java Bean (EJB) pour montrer qu'une projection automatique sur cette plate-forme n'est possible qu'en tranchant ces choix d'implémentation. L'ensemble de ces choix doit être consigné dans un **cadre de conception**[1] ainsi qu'une démarche. Une partie du cadre de conception exploite le découpage et l'architecture fournis par Component Unified Process. Afin, de raffiner ce modèle PIM pour le projeter sur la plate-forme EJB, nous proposons le framework EJB++ qui fournit un modèle de composants évolués intermédiaire, plus riche que le modèle EJB. Il introduit des concepts de plus haut niveau, qui correspondent à ces choix. La projection est rendue plus directe et la génération de code à partir d'un modèle PIM sur la plate-forme EJB semble plus réalisable et intéressante. Il propose, par exemple, en plus des associations, le mécanisme de connexion, qui permet aux composants EJB de catégoriser leurs méthodes métiers en facettes et réceptacles. Ces deux concepts correspondent directement aux interfaces requises et fournies du composant CUP. Dès lors, la projection d'un modèle CUP vers la plate-forme EJB pourrait être guidée par le modèle intermédiaire EJB++.

[1] Le terme cadre de conception pourrait traduire le terme anglais framework

La section 5.1 présente plus précisément le framework EJB. Sur la base de ce framework, la section 5.2 présente son extension, le framework EJB++. Enfin, la section 5.3 présente la partie de l'outil CupTool qui intègre un générateur de code vers le framework EJB++. Cette implémentation montre l'intérêt de modèles intermédiaires, à la fois PIM comme le modèle CUP et PSM comme le modèle EJB++ et fournit une véritable chaîne de production et d'exploitation de composants.

5.1 La plate-forme Enterprise Java Bean

5.1.1 Les composants Enterprise Java Beans

Les Enterprise Java Beans sont des *composants logiciels distribués*. Ils permettent la conception de la partie métier des applications orientées objet distribuées dans un environnement spécifique Java. Un composant EJB typique est constitué de méthodes qui contiennent la logique liée à un métier. Ils ne possèdent pas d'interface graphique contrairement aux Java Beans et sont destinés à être déployés sur un serveur d'application, dans un hébergeur d'EJB : le *conteneur*[2].

Un client n'accède pas directement à un EJB par appel de méthodes à distance. Les spécifications *EJB* décrivent la notion de conteneur comme un fournisseur de services autres que métier, aux Enterprise JavaBeans. Ces services sont d'ordre technique, par exemple : la transaction, la persistance et la gestion du cycle de vie des instances d'un bean. Tant qu'un bean respecte les spécifications, il est déployable et exécutable dans n'importe quel conteneur de n'importe quel fournisseur. Un conteneur d'EJB simpliste gère l'interface d'objets métiers distants. C'est à lui que s'adressent les clients pour invoquer les méthodes de ces objets. Le comportement des conteneurs EJB assure de nombreux services qui simplifient grandement la programmation d'applications 3-tiers. Ainsi, les programmeurs n'ont plus à se soucier des détails techniques concernant la gestion des transactions et des états, la concurrence, la sécurité, les unités d'exécution multiples, le regroupement des ressources et autres API complexes de bas niveau.

5.1.2 Architecture des Enterprise JavaBeans

L'architecture EJB est constituée des éléments suivants (fig. 5.1) :

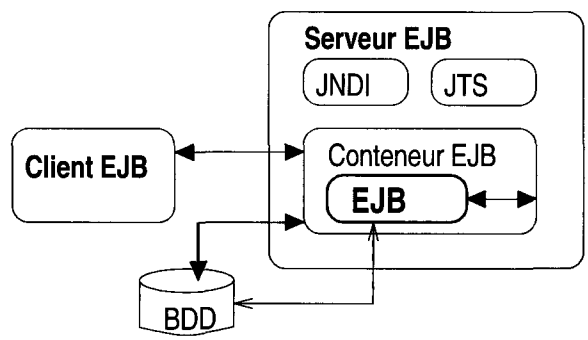


FIG. 5.1 – Architecture Enterprise JavaBeans

Un serveur d'application ou **serveur EJB** doit fournir l'ensemble des éléments suivants correspondant aux spécifications EJB. Le **conteneur EJB** héberge des EJB. Il fait "vivre les EJB". Le conteneur EJB est contraint à trois services : la sécurité, les transactions, la persistance. Un conteneur est responsable de la mise à disposition des classes EJB aux clients. Le **client EJB** est un programme qui accède aux EJB. Il peut être une servlet, une applet, une page Java Server (JSP), une application JAVA. L'**EJB** est un composant orienté métier. Il contient des méthodes et des données de l'entreprise. Une instance d'EJB est gérée par un conteneur. Tout accès à l'EJB passe par un conteneur. Le service **Java Naming**

[2]en anglais *container*

Directory Interface (JNDI) est une sorte d'annuaire qui permet de retrouver les références distantes des EJB à partir d'un nom logique. Le service **Java Transaction Service (JTS)** permet au conteneur de gérer l'ensemble des transactions de la façon spécifiée dans le descripteur de déploiement de chacun des EJB déployés.

Séparation des données et traitements

Plus précisément, nous distinguons deux catégories d'EJB[3] (fig. 5.2) spécialisés soit dans des traitements d'ordre applicatif : les EJB de session ; soit dans la gestion des informations pérennes : les EJB entités. Cette distinction est commune à de nombreux environnements à base de composants.

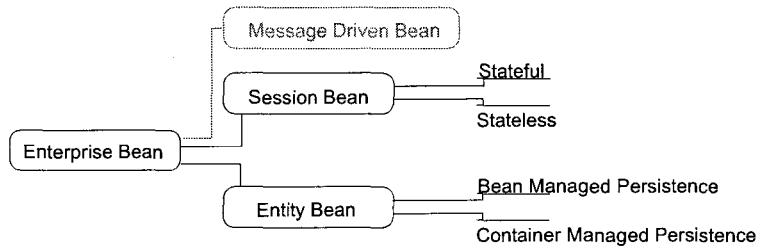


FIG. 5.2 – Lesdifférents types d'EJB

- **Les EJB de session**
  Un bean de session a une durée de vie assez courte, limitée au temps d'une session client. Il est créé en réponse à la requête d'un seul client. Il utilise le service de transaction du conteneur. Il est détruit en cas de panne du serveur et le client doit instancier un nouveau bean pour continuer les opérations. Il ne représente pas les données devant être stockées dans une base de données. Le conteneur offre un environnement d'exécution ajustable pour exécuter simultanément un grand nombre de beans session. Généralement les beans session sont utilisés pour des données transitoires pouvant être perdues.
  Un bean de session peut être **sans état** (*stateless*) ; il ne garde pas trace des informations échangées d'un appel de méthode à un 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.
  Un bean de session 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 bean session 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 EJB entité**
  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 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-

[3]Dans ce mémoire, nous n'abordons que les traitements synchrones. Pour cela, nous ne considérerons pas la notion de Message Driven Bean proposé par les spécifications EJB

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[4] spécifie que le bean doit lui-même contenir les primitives de sauvegarde et restauration vers une base de données persistante. Les méthodes **ejbLoad()** et **ejbStore()** 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 conteneur évite d'avoir à implémenter les méthodes **ejbLoad()** et **ejbStore()**. Cela rend le code du bean CMP[5] portable sur n'importe quel environnement puisque la génération s'effectue en fonction de l'environnement opérationnel sur lequel le bean est déployé. Dans le cadre de notre prototype, les composants générés utiliseront ce service. Le développeur pourra, ensuite, retravailler le code et passer en BMP

La gestion du mapping objet-relationel

Le modèle de composants d'entreprise EJB (*Enterprise JavaBeans*) [84, 74] est un modèle bien conçu pour exploiter des données persistantes via des Systèmes de Gestion de Bases de Données Relationnelles (SGBDR). La gestion de la persistance, ainsi que l'intégrité des données, considérées comme des services (au même titre que la sécurité, le nommage et la gestion des transactions) sont assurées par la notion de conteneur, le développeur de composants EJB ne se préoccupant, alors, que du "code métier". Des composants de type **EntityBean** permettent d'accéder et de modifier ces données via les méthodes du composant. Le rôle du conteneur EJB consiste à charger/sauver et assurer la cohérence des contraintes d'intégrité (les autres rôles du conteneur tels que la sécurité et la transaction ne sont pas traités ici). Les composants EJB entité peuvent également être reliés à d'autres EJB entités en utilisant le mécanisme de relations **relationship**. Un schéma relationnel de base de données est donc parfaitement projetable vers un schéma de composants EJB. Le mapping objet-relationnel est pris en charge par le conteneur selon les spécifications EJB.

De manière abstraite, un composant EJB peut être représenté par la figure 5.3.

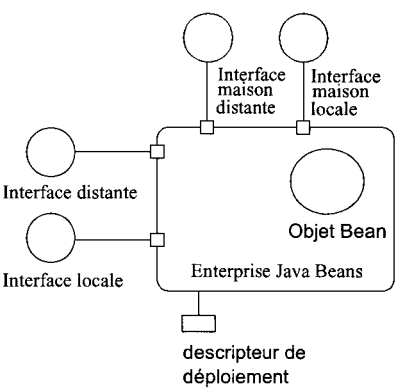


FIG. 5.3 – Modèle abstrait de composant EJB

[4] *BMP - Bean-Managed Persistence*
[5] *CMP - Container-Managed Persistence*

5.1.3 Le développement de composants EJB

Le développeur d'un EJB doit fournir trois classes :

- **la classe du bean** qui implémente l'interface **javax.ejb.SessionBean** pour un composant de session, **javax.ejb.EntityBean** pour un composant entité. Cette classe contient l'implémentation des méthodes métiers, ainsi que l'implémentation de la méthode **ejbCreate()**. En plus pour le composant entité, il faut que cette classe contienne l'implémentation des méthodes **ejbPostCreate()** correspondant à chaque méthode **ejbCreate()** implémentée, **ejbFindByPrimaryKey()** pour la persistance gérée par bean, ainsi que d'autres méthodes de recherches.
- **l'interface home**[6] qui étend l'interface **javax.ejb.EJBHome**. Cette interface contient les signatures des méthodes de création. Pour le composant entité, elle contient les signatures des méthodes de recherche **findByPrimaryKey()**, et autres.
- **l'interface remote**[7] qui hérite de l'interface **javax.ejb.EJBObject**. Elle présente les signatures des méthodes métiers.

5.1.4 Le déploiement et exécution sur un serveur d'applications

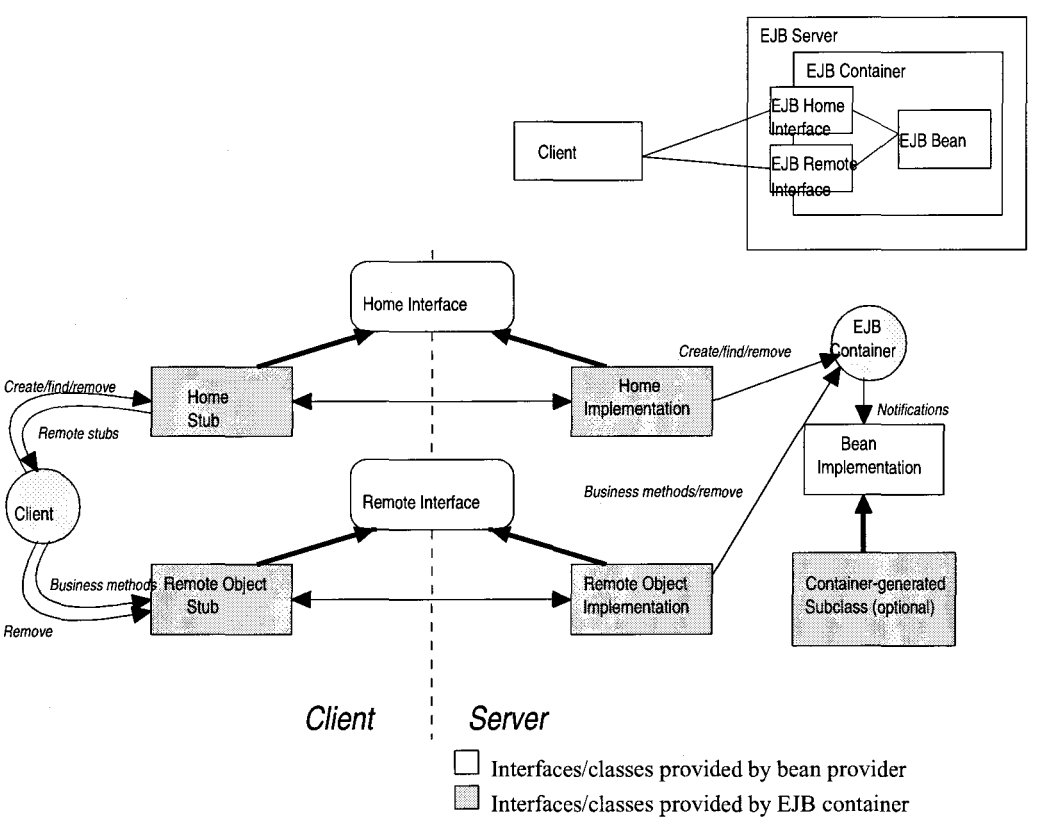


FIG. 5.4 – Communications entre éléments des EJB

[6]Le client accède directement à l'interface **home** qui est utilisée pour créer ou trouver des objets EJB d'un certain type.

[7]Le client obtient alors une référence vers le **stub** qui implémente l'interface remote qui contient les signatures des méthodes métier.

Un serveur d'applications est nécessaire pour déployer et exécuter une application conçue à l'aide de composants EJB. Il permet d'administrer les composants. Un bus logiciel[8] sous-jacent permet d'administrer des données et d'offrir des services aux composants EJB tels que la sécurité, l'administration, la transaction. L'application cliente ne voit (cf figure 5.4) que les interfaces **Home** et **Remote** de l'EJB. Le client accède à l'interface **Home** pour créer une instance de l'EJB. Il reçoit une souche de l'interface **Remote** lui permettant en local d'appeler les méthodes de l'EJB. Par RMI ou IIOP, les paramètres et résultats de ces méthodes sont sérialisés, ce qui permet l'interaction entre le côté serveur et le côté client.

Dans la suite, nous considérerons la plate-forme EJB comme schématisée dans la figure 5.5. L'interface **Home** est l'application du pattern *Fabrique* et permet de créer et de rechercher des instances du composant. L'interface **Remote** spécifie les méthodes métiers proposées aux clients de ce composant. Enfin, la classe **Bean** contient l'implémentation des méthodes du composant (le code métier).

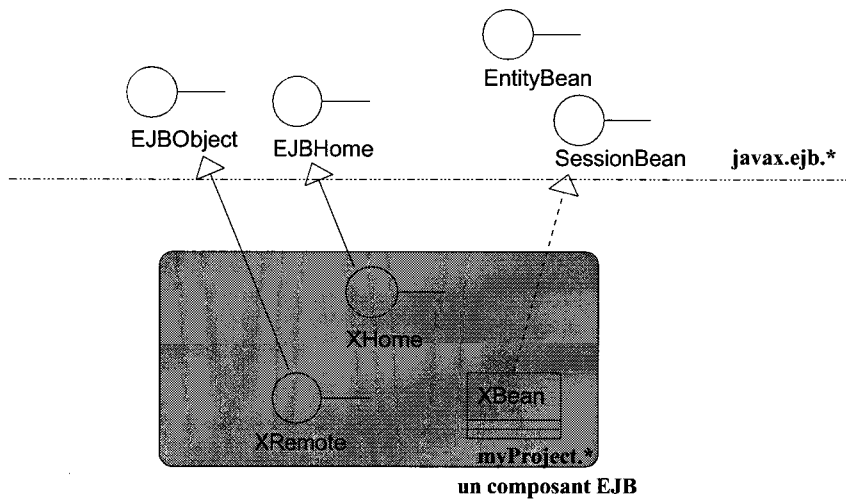


FIG. 5.5 – Synthèse du framework EJB

Après avoir brièvement présenté la plate-forme spécifique Enterprise JavaBeans et ses propriétés dans le cadre du développement d'applications à base de composants, nous présentons notre réponse à la complexité de choix de conception, lors du passage d'un modèle d'analyse, indépendant d'une plate-forme, à un modèle de conception, spécifique à une plate-forme technologique.

5.2 Définition du modèle intermédiaire EJB++

5.2.1 Motivations

Le choix de la plate-forme EJB, plutôt que CCM permet de tirer pleinement partie du développement d'applications dites "3-tiers" dans lequel elle s'est spécialisée. En effet, les EJB permettent de déployer des objets Java sur n'importe quel serveur d'applications accueillant les EJB, sur des servlets pour les applications web. Il est donc possible de séparer les objets contenant du traitement propre à

[8]en anglais *Object Request Broker*

une application et des objets métiers contenant des données persistantes et exploitées plus largement dans un système d'information.

La conception du framework EJB++ pour réaliser une projection du modèle CUP d'un système vers la plate-forme spécifique EJB, a été motivée notamment par les manques de concepts à base de composants dans la plate-forme spécifique EJB. Les concepteurs et développeurs sont obligés d'utiliser des artifices, ce qui rend la tâche difficile et anéantit la traçabilité. Le modèle EJB++ comble des lacunes de la plate-forme EJB parmi les suivantes :

- L'ajout du traitement ou des propriétés à un bean est possible par délégation de ces nouveaux services à d'autres EJB. Cette caractéristique correspond à la notion de connexion dans le modèle CUP, qui n'a pas d'équivalent en EJB.
- Le modèle EJB est plus simple que d'autres modèles plus riches comme CCM. Pourtant, l'absence de la notion de ports qui permettent de relier des composants EJB qui ne se connaissent pas nuit à une bonne réutilisation de ces composants et une meilleure spécification des services fournis et requis.
- Enfin, le passage de CUP à EJB reste difficile du fait d'un trop grand écart entre ces modèles et donc des nombreux choix de conception à prendre pour passer de l'un à l'autre.

Un composant EJB a une structure très simple. Il dispose d'une interface Java permettant de décrire l'ensemble des méthodes disponibles sur ce composant. Grâce au mécanisme d'héritage multiple d'interfaces Java, il est possible de structurer en catégories l'ensemble de ces méthodes (en fonction des cas d'utilisation par exemple). Cependant, du point de vue du client d'un composant EJB, il n'existe pas de mécanisme simple d'introspection qui permet de découvrir ces catégories de méthodes. Seule l'utilisation du mécanisme lourd de réflexion Java dans l'implémentation de l'application cliente le permettrait.

Les composants peuvent être reliés par association à d'autres composants (fig. 5.6), ce qui permet la traduction rapide et efficace de schémas relationnels des systèmes de gestion de base de données en composants EJB et garantit l'intégrité référentielle des liens d'une association en fonction des cardinalités de l'association définie lors du déploiement. Cependant, le type des composants métiers reliés par association doit être connu avant toute compilation et déploiement de ces composants. En effet, les associations sont définies dans le descripteur de déploiement. Grâce à celui-ci, le conteneur génère le code nécessaire pour gérer ces associations. Les associations ne sont plus modifiables après le déploiement. Il n'y a donc pas de possibilités de **connexion dynamique par association**.

Dans un système d'information à base de composants, les mêmes entités jouent des rôles fonctionnels destinés à différents usages, différents utilisateurs et leur implémentation implique des compétences différentes [27]. Dans le cadre des bases de données, ces aspects fonctionnels sont traités par les vues et schémas-vues. Dans le cadre de la conception à base de composants, des implémentations existent pour disposer de la programmation par vues [27, 10]. Le modèle EJB, distingue deux types de composants : entité, et session. Un composant EJB entité encapsule une donnée persistante, alors que le composant session encapsule un traitement. Cette séparation données et traitements va à l'encontre du modèle objet, mais semble intéressante pour la conception à base de composants. Bien que cette différenciation pourrait permettre d'appliquer des traitements différents à un même composant entité, le modèle EJB n'intègre pas directement cet aspect dans la conception d'application à base de composants.

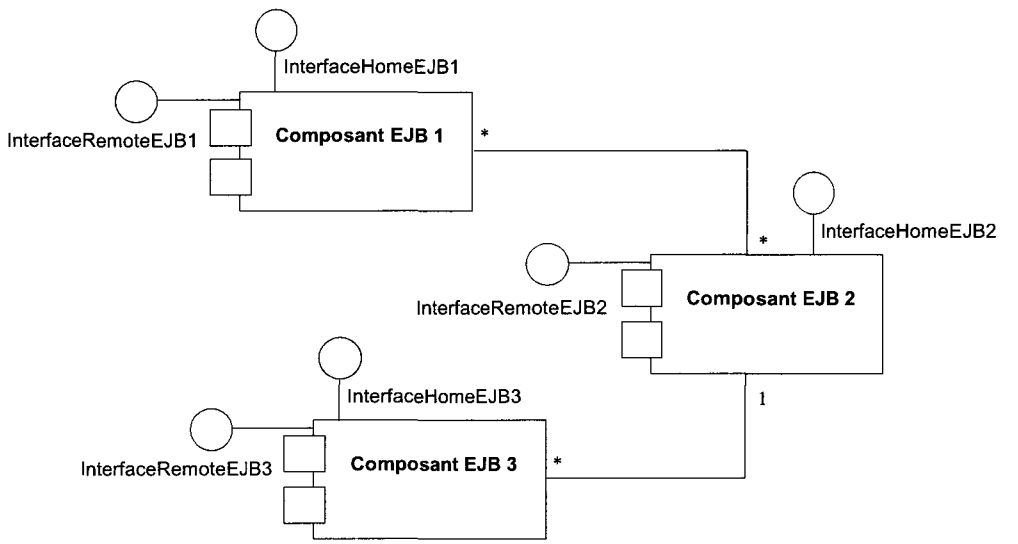


FIG. 5.6 – Association de composants EJB

5.2.2 Concepts

Dans l'article [11], nous avons défini le modèle EJB++. Ce modèle de composants simple apporte un certain nombre de caractéristiques intéressantes pour le développement de composants de plus haut niveau et comble les lacunes de plates-formes telles que les EJB. Le principe de base est que l'implémentation logicielle du modèle de composants EJB++ soit une implémentation purement extensive. Un composant EJB++ dispose de toutes les caractéristiques d'un composant EJB. De cette manière, les composants produits restent compatibles avec les outils les plus connus et utilisés dans le marché des serveurs d'applications. Ainsi, les résultats de nos expérimentations sont directement exploitables par des industriels cherchant à définir des composants sur l'étagère. Nous avons développé une couche logicielle qui s'intègre dans la plate-forme EJB.

Nous nous sommes inspirés du modèle abstrait de composants de CORBA pour introduire de nouveaux concepts à la plate-forme EJB, afin de proposer un modèle de composants intermédiaire pour projeter un modèle PIM à base de composants sur la plate-forme EJB. Nous introduisons notamment la notion de ports de composants et de connexions entre les ports. Nous avons sélectionné, dans ce modèle [34, 70] les concepts pertinents en vue de disposer de composants EJB réutilisables. En plus de toutes les spécificités du modèle EJB, un composant EJB++ peut fournir un ensemble de méthodes structurées en catégories, nommées **facettes** du composant. Chaque composant dispose de fonctions simples d'**introspection** qui permettent à une application cliente du composant, ou un autre composant de découvrir et de manipuler ces facettes. Pour qu'un composant puisse fonctionner, il est possible qu'il ait besoin d'être connecté à un ou plusieurs composants. Un composant peut donc avoir un ou plusieurs **réceptacles**. Ce sont des interfaces requises pour ce composant. Des mécanismes d'introspection existent et permettent la découverte simple de ces réceptacles. Facettes et réceptacles sont identifiés par un nom. Les connexions via ces ports s'établissent dynamiquement entre facettes et réceptacles. Chaque composant dispose automatiquement de fonctions de connexions.

Dans le cadre d'une structuration fonctionnelle à l'aide de composants EJB++, associations et connexions entre composants ont des rôles spécifiques et complémentaires. Les associations ont pour rôle de décrire le système d'information utilisé que ce soit au niveau d'un contexte fonctionnel ou au niveau du plan de base [17]. Les connexions entre composants permettent de relier ces composants entre un contexte fonctionnel et un plan de base.

5.2.3 Le framework EJB++

Ainsi, un composant EJB++ est muni de ports : les ports fournis permettant de catégoriser les méthodes métiers ; les ports requis exprimant dans des interfaces les besoins du composant ; d'un mécanisme de connexion qui vient s'ajouter au mécanisme d'association des EJB 2.0 ; d'un mécanisme d'introspection d'interfaces permettant, à l'exécution d'explorer les interfaces du composant. Pour bénéficier de ces avantages, le framework EJB++ vient étendre le modèle EJB (cf figure 5.5), par une abstraction de la notion de composant.

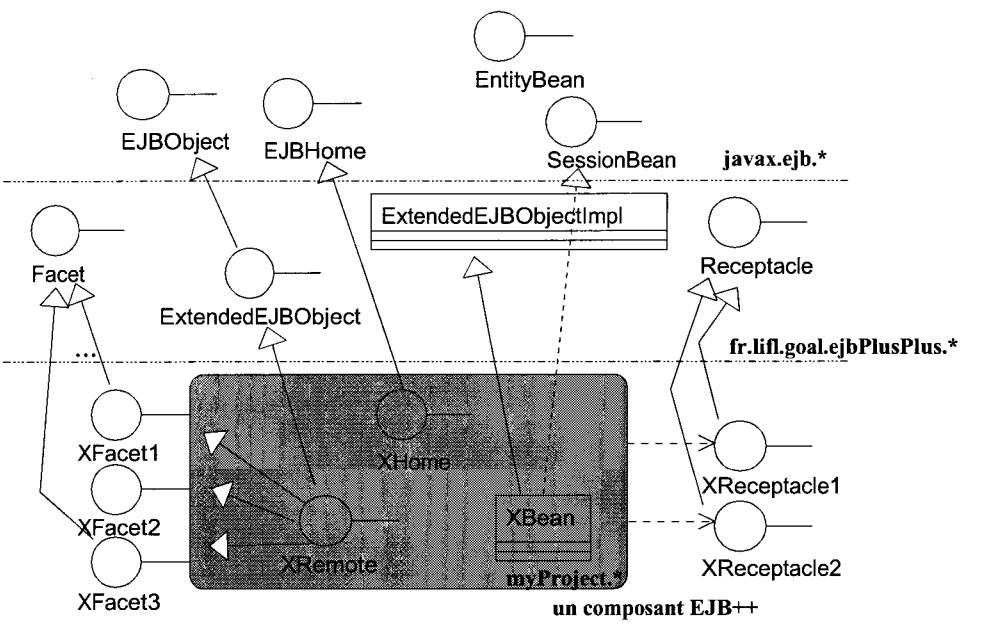


FIG. 5.7 – Synthèse du framework EJB++

Le framework EJB++ consiste en une couche logicielle qui vient s'interfacer entre la couche logicielle du framework EJB et les composants EJB. La figure (cf figure 5.7) décrit un composant EJB++ **X** disposant de trois facettes (**XFacet1**, **XFacet2**, **XFacet3**). Toutes les facettes sont exploitées par l'interface **Remote** du composant EJB, ce qui permet la stricte compatibilité avec la norme EJB. En outre, le composant EJB++ **X** dispose de deux réceptacles (**XReceptacle1**, **XReceptacle2**).

Implémentation du framework EJB++

L'annexe A contient tout le code du framework EJB++.

- **L'interface ExtendedEJBObject** permet l'ajout des méthodes étendues au composant EJB. Le composant EJB++ dispose ainsi des fonctionnalités étendues telles que les mécanismes de connexion

et de déconnexion, ainsi que des méthodes d'introspection sur les facettes et réceptacles du composant, et enfin une méthode permettant de vérifier la configuration du composant.

```
public interface ExtendedEJBObject extends javax.ejb.EJBObject {

  public void connect (String receptacleName, Facet facet)
    throws RemoteException, UnknownPortException,
           AlreadyConnectedException;
  public void disconnect (String receptacleName)
    throws RemoteException, UnknownPortException;

  public Facet getFacet (String facetName)
    throws RemoteException, UnknownPortException;
  public Vector getFacetNames ()
    throws RemoteException;
  public Vector getReceptacleNames ()
    throws RemoteException;

  public Facet getFacet (String facetName)
    throws RemoteException, UnknownPortException;
  public Vector getFacetNames ()
    throws RemoteException;
  public Vector getReceptacleNames ()
    throws RemoteException;

 public void configuration_complete()
    throws InvalidConfigurationException;
}
```

L'interface **ExtendedEJBObject** dispose donc de toutes les fonctionnalités de l'interface **EJBObjet**. Les fonctionnalités suivantes sont ajoutées. La méthode **connect** permet de connecter deux composants EJB++. Pour cela, le développeur qui veut connecter un composant client à un autre, dit serveur, doit donner le nom du réceptacle du composant client et la référence d'une facette du composant serveur. Le composant client ne peut se connecter au composant serveur que s'il n'est pas connecté. Dans le cas contraire, une exception **fr.lifl.goal.ejbPlusPlus.AlreadyConnectedException** est levée. L'exception **fr.lifl.goal.ejbPlusPlus.UnknowPortException** est levée dans le cas où le composant client ne possède pas de réceptacle qui porte ce nom.
La méthode **disconnect** permet la déconnexion de deux composants. Le développeur s'adresse au composant client en spécifiant le réceptacle qu'il veut déconnecter.
Les méthodes d'introspection permettent à partir de la référence d'un composant d'obtenir une référence de facette à partir d'un nom logique, dans le but de connecter ce composant à un autre. Les méthodes **getFacetNames()** et **getReceptacleNames()** permettent respectivement d'obtenir les noms des facettes et des réceptacles d'un composant.
La méthode **configuration_complete()** est la seule méthode qui doit être implémentée par le développeur. Cette méthode peut vérifier que le composant est correctement configuré pour avoir le comportement désiré. Par exemple, cette méthode peut vérifier que tous les réceptacles sont connectés.

- La classe **ExtendedEJBObjectImpl** contient l'implémentation de l'interface **ExtendedEJBObjet**.

- Les interfaces **Facet** et **Receptacle** sont des interfaces vides qui permettent de typer les interfaces d'un composant EJB++ comme fournies ou requises.

- Un certain nombre d'exceptions spécifiques sont à la disposition des développeurs afin de garantir un comportement stable d'un système développé avec le framework EJB++.

Implémentation d'un composant EJB++

L'annexe B contient tout le code du composant xComponent qui illustre nos propos.

Le développement d'un composant EJB++ commence par la définition des services qu'il fournit et des services qu'il requiert. Cela est fait par la programmation des interfaces du composant. Le composant **X** dispose de trois facettes. Supposons que la facette **XFacet1** propose deux méthodes. Les autres facettes seront développées de la même façon.

```
package xComponent;

import fr.lifl.goal.EJBPlusPlus.Facet;
import fr.lifl.goal.EJBPlusPlus.UnknownPortException;

public interface XFacet1 extends Facet{
  public String getName() throws java.rmi.RemoteException;
  public Integer plus (Integer arg1, Integer arg2)
    throws UnknownPortException, java.rmi.RemoteException;
}
```

Notons que le composant **X** correspond au paquetage **XComponent**.

Le composant **X** dispose de deux réceptacles développés comme suit :

```
package xComponent;

public interface XReceptacle extends fr.lifl.goal.EJBPlusPlus.Receptacle {
  public Integer add(Integer arg1, Integer arg2)
    throws java.rmi.RemoteException;
}
```

Conformément au modèle EJB, le composant EJB++ a une interface **Remote** qui par héritage contient les méthodes de toutes les facettes du composant et les méthodes présentées plus haut, fournies par le framework : les mécanismes de connexion et de déconnexion, ainsi que celui d'introspection sur les facettes et réceptacles du composant.

```
package xComponent;
import fr.lifl.goal.EJBPlusPlus.ExtendedEJBObject;
```

```
public interface XRemote extends ExtendedEJBObject,
  XFacet1, XFacet2, XFacet3 {}
```

A la différence des EJB, l'application cliente ne doit pas accèder par l'interface Remote, mais par les facettes du composant. Ceci n'est pas vérifié afin de conserver la complète comptatibilité avec les applications utilisant les composants EJB++ comme des composants EJB classiques.

Tout comme la classe du bean d'un EJB, la classe bean EJB++ doit étendre soit **javax.ejb.EntityBean** soit **javax.ejb.SessionBean** selon qu'il est un composant entité ou de session. Le développement d'un composant EJB++ est alors équivalent au développement d'un composant EJB. La classe bean du composant EJB étend en plus la classe abstraite **ExtEJBObjectImpl** et doit donc implémenter les méthodes abstraites comme la méthode **configuration_complete**. De plus, dans la classe du bean, des services peuvent être nécessaires au comportement développé dans une méthode, une partie de ce comportement étant délégué à un autre composant. Le framework permet grâce aux réceptacles d'appeler ce comportement sans nécessité de référence sur un autre composant.

```
package xComponent;

import java.rmi.RemoteException;
import javax.ejb.SessionBean ;
import javax.ejb.SessionContext;
import java.rmi.RemoteException ;
import fr.lifl.goal.EJBPlusPlus.*;
import fr.lifl.goal.EJBPlusPlus.ExtendedEJBObjectImpl;

public class XBean extends ExtendedEJBObjectImpl implements SessionBean {
  SessionContext ctx;
  public Integer plus (Integer arg1, Integer arg2)
    throws UnknownPortException, UnknownMethodException, RemoteException {
    Object[] args = {arg1, arg2};
    Class[] argTypes = {Integer.class, Integer.class};
    return (Integer)invokeMethod("XReceptacle", "add", argTypes, args);
  }
  public Integer sub (Integer arg1, Integer arg2) throws RemoteException {
    return new Integer(arg1.intValue()-arg2.intValue()) ;
  }
  public String getName() {
    return "tartempion";
  }
  public void ejbCreate () throws java.rmi.RemoteException {
    this.setReceptacleNames();
  }
  public void ejbPostCreate() {
    this.setReceptacleNames();
  }
  public void setSessionContext (javax.ejb.SessionContext ctx) {
    this.ctx=ctx;
  }
  public javax.ejb.EJBContext getEJBContext(){
```

```
    return ctx;
  }
  public void configuration_complete()
    throws InvalidConfigurationException {
    try {
      getReceptacle("XReceptacle");
    } catch (UnknownPortException e) {
      throw new InvalidConfigurationException();
    }
  }
  public void setReceptacleNames(){
    this.receptacleNames.addElement("XReceptacle");
  }
  public void ejbRemove() {}
  public void ejbActivate() {}
  public void ejbPassivate() {}
}
```

Cycle de vie d'un composant EJB++

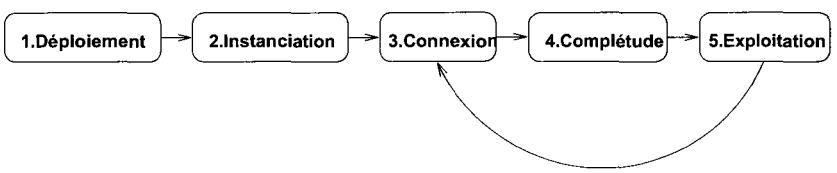


FIG. 5.8 – Cycle de vie du composant EJB++

1. Déploiement
   Le déploiement est en tout point identique à celui des EJB.
2. Instanciation
   Comme un composant EJB, l'instanciation d'un composant EJB++ est réalisée grâce au service de nommage JNDI, en récupérant la référence de l'interface *Home* et en créant des instances du composant EJB++. Ensuite l'accès aux méthodes métiers doit être fait par le biais des facettes.

```
xComponent.XRemoteHome unComposantHome=null ;
xComponent.XRemote unComposant ;*
ctx= new javax.naming.InitialContext();  .
Object ref1= ctx.lookup("xcomp") ;
unComposantHome =
  (xComponent.XRemoteHome)javax.rmi.PortableRemoteObject
  .narrow(ref1,xComponent.XRemoteHome.class );
unComposant = unComposantHome.create() ;
xComponent.XFacet1 uneFacetteDuComposantX =
  (xComponent.XFacet1)unComposant.getFacet("xComponent.XFacet1");
System.out.println(uneFacetteDuComposantX.plus(new Integer(5),
                                                new Integer(1)));
...
```

3. Connexion
   La connexion se fait par un appel de méthode. L'EJB++ offre, en effet, ce service. On lui passe la référence de la facette du composant que l'on souhaite connecter et le nom du réceptacle. La connexion d'un réceptacle n'est possible que s'il n'est pas déjà connecté. Le cas échéant, il est possible à l'exécution de déconnecter un réceptacle et de le connecter à une autre facette, ou une facette d'un autre composant.

```
...
unComposant.connect("XReceptacle",
    unComposantY.getFacet("yComponent.YFacet"));
...
```

4. Complétude et exploitation
   La méthode *configuration_complete* permet au développeur de faire les contrôles de connexion nécessaires pour garantir le contrat fonctionnel du composant. Cette méthode est implémentée librement par le développeur. Soit il vérifie que toutes les connexions sont effectives avant de donner son accord pour un fonctionnement correct. Soit il ne vérifie rien. Dans ce cas, soit il y a une erreur d'exécution lorsqu'un service délégué est appelé, soit le comportement est simplement différent.

```
try {
  xComponent.configuration_complete();
  yComponent.configuration_complete();
  // code d'exploitation des deux composants
  ...
} catch (InvalidConfigurationException e) {
  ...
}
yComponent.YFacet facette = (yComponent.YFacet)unComponentY
                            .getFacet("yComponent.YFacet");
xComponent.connect("XReceptacle1",(Facet)facette);
```

Deux composants connectés ont un contrat mutuel. Un réceptacle du composant client doit être conforme à une facette du composant serveur. Ce réceptacle et cette facette ont alors la même interface. Le pattern **Adaptor** [57] est utilisé pour définir un connecteur entre composants lorsque les réceptacles et facettes ne sont pas conformes. Dans l'architecture EJB, cet adaptateur est un composant EJB de type session et adopte la structure générique suivante :

```
public class AdaptorXFacet1YReceptacle
  extends SessionBean, YReceptacle {
  private XFacet1 aFacet;
  public ejbCreate (XFacet1 aFacet) {
    this.aFacet = aFacet;
  }
  public String obtenirLeNom() {
    return aFacet.getName()
    ...
  }
  public Integer additionner(Integer arg1, Integer arg2) {
```

```
        return aFacet.plus(arg1, arg2));
        ...
    }
    ...
}
```

Le code de ces adaptateurs peut être entièrement généré à partir de la description des facettes et réceptacles à adapter.

Tableau comparatif EJB / EJB++

Caractéristique	EJB	EJB++
interfaces fournies	1 interface Remote	plusieurs interfaces Facettes
interfaces requises	NEANT	plusieurs interfaces Réceptacles
introspection	à développer dans le client	méthodes d'introspection dans composant
connexion	NEANT	méthode de connexion/déconnexion dans composant
association	relationship EJB entité	idem
services non fonctionnels	conteneur	idem

Les apports du modèle EJB++

Dans le cadre de l'ingénierie logicielle dans lequel on souhaite réutiliser des composants, ce modèle apporte un bénéfice considérable par rapport au modèle EJB standard. En effet, les concepts de facettes et de réceptacles délimitent finement les frontières de ces composants et en donnent une définition plus précise. Un composant EJB peut désormais être étendu par un autre, en utilisant le mécanisme de connexion. Le modèle EJB++ est plus riche que le modèle EJB et peut favoriser ou faciliter le portage d'objets métiers en composants. Notamment, la démarche du Processus Unifié introduit les objets d'analyse frontières, de contrôle et entités. Avec ce modèle plus riche, il semble plus aisé de porter ces objets en composants EJB++ session et EJB++ entité. L'une des caractéristiques intéressantes de ce modèle est qu'elle favorise une structuration en contextes fonctionnels de composants (cf figure 5.9). Les multiples recherches dans la conception par contextes [52, 1, 18, 17, 10], sujets [75, 76], ou même aspects [64] disposent ainsi d'une plate-forme expérimentale qui leur permettrait de transiter d'objets de conception structurés en contextes à des composants fonctionnels exploitables. La possibilité de préserver à l'exploitation la structuration des objets de conception favorise ainsi la notion importante de traçabilité [10].

L'exemple de la figure 5.9 montre l'intérêt de ce modèle pour une structuration fonctionnelle de composants. Un composant entité $Vehicule$ dispose de différentes propriétés (marque, couleur, ...) accessibles par la facette $CaracteristiquesFacet$. Ce composant de base dispose, en outre, d'associations qui montrent les relations entre une voiture et d'autres entités génériques du domaine métier de l'automobile (constructeur, moteur, ...).

Dans le contexte fonctionnel *vente*, le composant $Produit$, connecté au composant $Vehicule$ dispose de toutes les propriétés et comportements du plan de base (méthodes déléguées au composant connecté) et également de propriétés et comportements propres au métier de ce plan fonctionnel. Il dispose de plus, de relations avec des entités ou classes fonctionnelles propres à ce contexte métier.

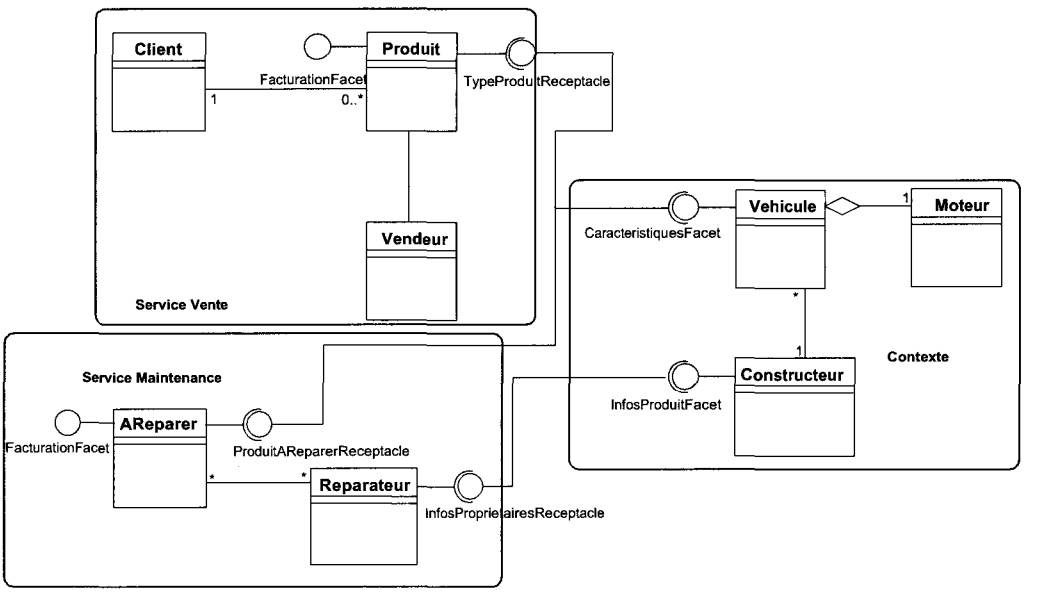


FIG. 5.9 – Exemple d'application supportée par le framework EJB++

Le service *maintenance* gère le composant *AReparer* qui nécessite également d'être connecté à un composant qui lui fournit les informations de contexte , ici *Vehicule*.

Dans cet exemple simpliste, le composant *Vehicule* est utilisable pour les applications liées au service *vente* et est également utilisable pour les applications liées au service *maintenance*. De plus, les fonctionnalités de chacun des contextes fonctionnels pourraient être génériques et réutilisables avec un autre plan de base (cf figure 5.10).

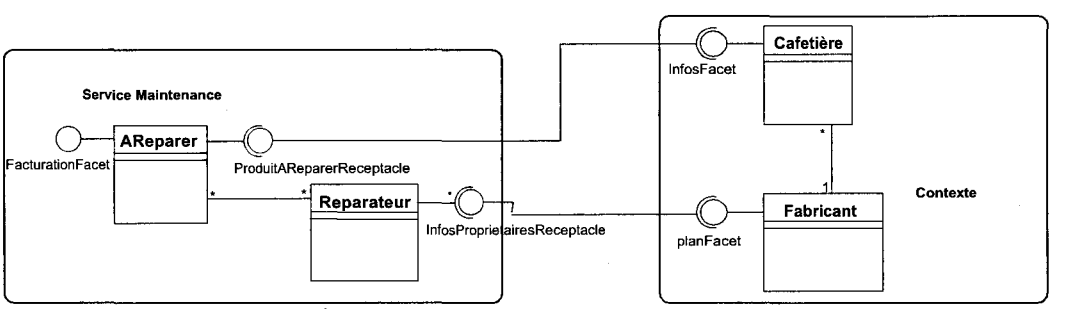


FIG. 5.10 – Autre exemple d'application supportée par le framework EJB++

Dans ce cadre d'une structuration fonctionnelle à l'aide de composants EJB++, association et connexion entre composants ont chacune un rôle spécifique et complémentaire. Les associations ont pour rôle de décrire le système d'information utilisé que ce soit au niveau d'un contexte fonctionnel ou au niveau du plan de base. Les connexions entre composants permettent de relier ces composants entre un contexte fonctionnel et un plan de base.

Cette étude présente deux avantages. Tout d'abord, nous disposons d'un modèle de composants plus riche que le modèle EJB. Il intègre le modèle EJB 2.0, notamment le concept de relations, inconnu dans d'autres modèles comme CORBA Component Model. Certains concepts sont définis dans d'autres modèles comme les ports du modèle CCM. Ce framework paraît être une brique essentielle pour des démarches de conception fonctionnelles, puisque des mécanismes de connexion favorisent la structuration de composants en contextes.

5.3 La génération de CUP vers EJB++

5.3.1 La projection CUP vers EJB

Des choix d'implémentation des composants CUP en composants EJB++ sont faits pour que la projection soit plus directe (fig. 5.11).
Le tableau suivant résume ces choix de transformation.

CUP Model	EJB++ Model	EJB Model
Component	archive JAR	archive JAR
ProvidedPort	Facet	EJB Interface Remote
RequiredPort	Receptacle	Interface Java
ControlPart	SessionBean ++	Session Bean
EntityPart	EntityBean ++	Entity Bean

TAB. 5.1 – Projection CUP-EJB++-EJB

Un composant CUP est transformé en une archive (.jar) contenant un ensemble de composants EJB++. Un composant Session EJB++ applique le pattern façade, en proposant les facettes correspondantes aux interfaces fournies du composant CUP. Le composant EJB++ est muni de réceptacles correspondant aux ports requis CUP. Les parties entités du composant ont généré un composant EJB++ entités relié par association. La projection exploite la complémentarité des notions d'association et de connexion. En effet, les relations entre parties de composants CUP sont implémentées par des associations, tandis que les connexions CUP sont directement projetées vers les connexions EJB++.
Il est à noter que ces transformations pourraient être par la suite paramétrables, selon des critères techniques, telles que l'occupation mémoire du serveur d'applications ou la performance, ...Nous ne nous préoccupons pas des critères non fonctionnels. Par contre, une optimisation par des experts techniques est souhaitable afin que cette implémentation par défaut correspondent d'avantage aux exigences techniques.

5.3.2 Choix de conception lors de la projection

Le **port fourni** CUP est une interface offrant une vue sur le composant. Il contient un certain nombre de méthodes correspondant aux comportements du composant accessibles par ce port fourni. La projection d'un port fourni donne une **facette** dans le modèle EJB++.

```
package MagasinComponent ;
import fr.lifl.goal.EJBPlusPlus.*;

public interface LouerRendreMaterielFacet extends Facet {
  public Materiel chercherMateriel(String reference) ;
  public void payerLocation(int montant) ;
```

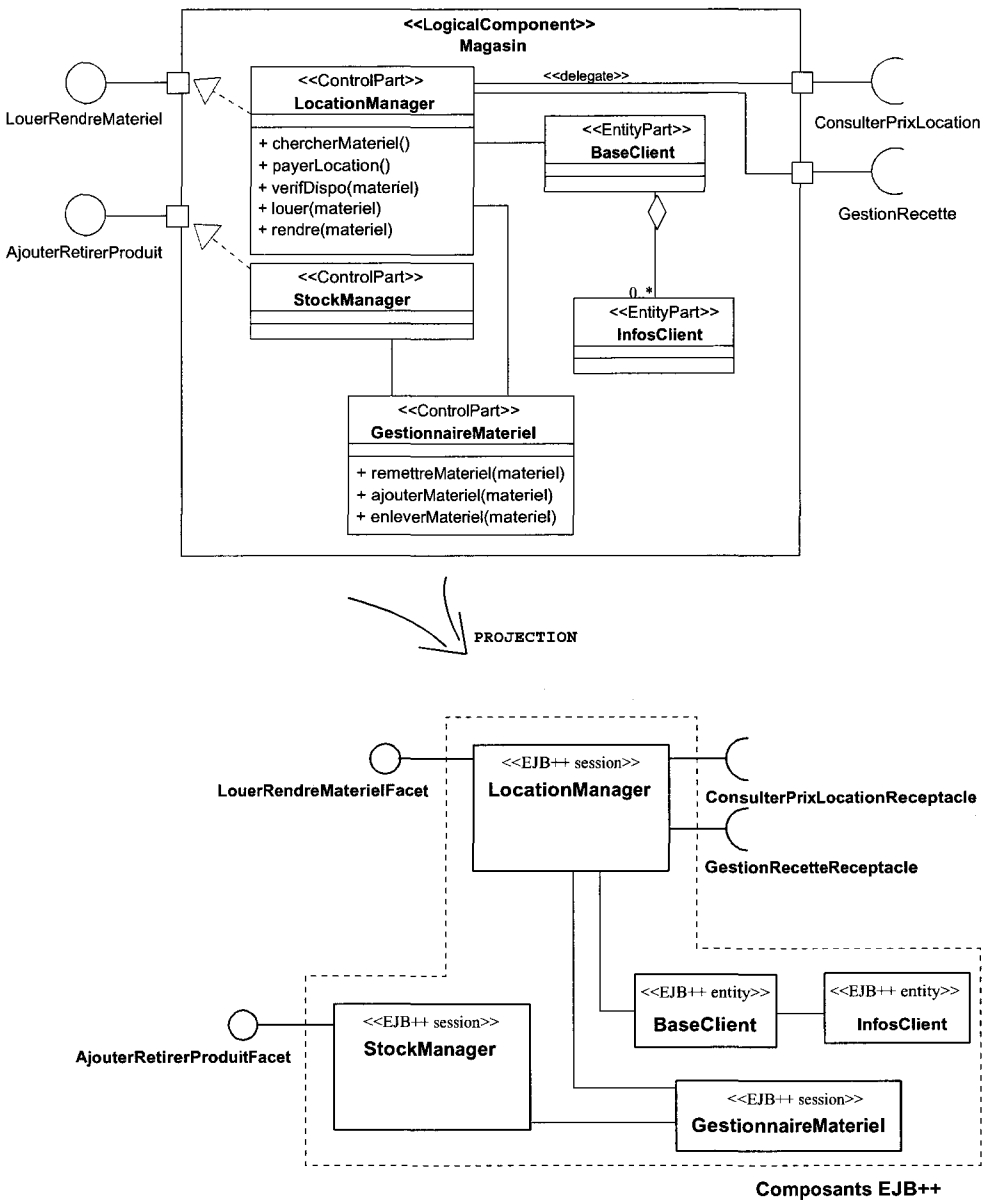


FIG. 5.11 – Exemple de projection d'un composant CUP vers la plate-forme EJB++

```
  public String verifierDisponibilite(String reference) ;
  public louer(String reference) ;
  public rendre(String reference) ;
}
```

Une **partie de contrôle** de composant CUP correspond au concept de classe. Elle contient des méthodes ainsi que le corps des méthodes et des données temporaires nécessaires à une session. Chaque partie de contrôle est projetée en un composant **EJB++ session**.

Interface Remote de l'EJB++

```
package MagasinComponent ;
import fr.lifl.goal.EJBPlusPlus.ExtendedEJBObject;

public interface LocationManagerRemote
  extends ExtendedEJBObject  , LouerRendreMaterielFacet  {}
```

Interface Home de l'EJB++

```
package MagasinComponent ;

import java.rmi.RemoteException;
import javax.ejb.CreateException;
import javax.ejb.EJBHome;

public interface LocationManagerRemoteHome  extends EJBHome {
  public LocationManagerRemote create()
    throws CreateException, RemoteException;
 }
```

Bean de l'EJB++

```
  package MagasinComponent ;

  import java.rmi.RemoteException;
  import java.util.Date;
  import javax.ejb.SessionBean ;
  import javax.ejb.CreateException;
  import javax.ejb.SessionContext;
  import javax.ejb.EJBException ;
  import java.rmi.RemoteException ;
  import fr.lifl.goal.EJBPlusPlus.*;

  public abstract class LocationManagerBean
    extends ExtendedEJBObjectImpl implements SessionBean {

    EntityContext ctx;

    public Materiel chercherMateriel(String reference) {
      ...
    }
    public void payerLocation(int montant) {
```

```
    ...
  }
  public String verifierDisponibilite(String reference) {
    ...
  }
  public louer(String reference) {
    ...
  }
  public rendre(String reference) {
    ...
  }

  public void ejbCreate () throws java.rmi.RemoteException {
    this.setReceptacleNames();
  }
  public void ejbPostCreate() {
    this.setReceptacleNames();
  }
  public void setSessionContext (javax.ejb.SessionContext ctx) {
    this.ctx=ctx;
  }
  public javax.ejb.EJBContext getEJBContext(){
    return ctx;
  }
  public void configuration_complete(){
    ...
  }
  public void setReceptacleNames(){
    this.receptacleNames.addElement("ConsulterPrixLocation");
    this.receptacleNames.addElement("GestionRecette");
  }
  public void ejbRemove() {}
  public void ejbActivate() {}
  public void ejbPassivate() {}
}
```

Une **partie entité** de composant CUP correspond au concept de classe. Elle contient des méthodes ainsi que des données persistantes. Un **EJB++ entité** correspond à la projection d'une partie entité sur le modèle EJB++.

Un **port requis** est une interface qui contient les méthodes appelées par des parties du composant, et qui ne sont pas implémentées par des parties de ce même composant. Un **réceptacle EJB++** sera la projection d'un port requis.

```
package MagasinComponent ;
import fr.lifl.goal.EJBPlusPlus.*;

public interface GestionRecetteReceptacle  extends Receptacle {
  ...
```

}

Les associations entre parties d'un composant sont projetées en références d'EJB++. La relation d'implémentation de port fourni par une partie de contrôle est une relation *implement* d'un EJB++ session. Et l'association entre une partie de contrôle et un réceptacle est codée dans l'implémentation du EJB++ session correspondant.

En effet, les choix de conception sont ici directs, et il reste peu de chose à faire pour obtenir une application déployable.

5.3.3 Conclusion de la projection

Un modèle UML 1.4 ne guide pas le concepteur dans les choix d'architecture permettant de faire correspondre ce modèle sur la plate-forme EJB. On justifie, par cette étude, le fait qu'un modèle intermédiaire est nécessaire pour améliorer la projection.

L'avantage de l'approche que nous proposons réside dans la démarche employée qui se décompose en deux temps. La première phase consiste à exploiter le méta-modèle PIM (Platform Independent Model) proposé par CUP. Il est riche de caractéristiques permettant sa projection vers des plates-formes technologiques à base de composants. La projection vers la plate-forme EJB demande un cadre de conception, qui offre une correspondance directe entre concepts contenus dans CUP et exploitant la technologie des EJB. Cette phase offre divers points de vue à différents intervenants, tout en maintenant une cohérence et en capitalisant sur un paradigme à base de composants logiques.

La seconde phase consiste à projeter ce modèle vers un modèle spécifique PSM (Platform Specific Model) grâce à l'implémentation d'une couche logicielle tranchant les choix complexes de projection d'un modèle abstrait vers un modèle technologique concret. Cela permet d'obtenir des résultats très rapidement sans avoir à reconstruire une architecture complète de composants (à l'instar de [52]).

Nous avons appliqué, dans ce chapitre, la projection de modèle CUP vers le modèle EJB++. D'autres projections sont possibles. Ainsi, le modèle CCM, similaire par bien des aspects à EJB++, peut bénéficier de ces choix de projection. Il nous semble qu'une projection CUP vers CCM, en s'inspirant de la présente étude EJB++ soit facilement spécifiable.

La partie suivante valide l'approche sur des cas concrets de l'ingénierie logicielle empruntés à l'industrie. Elle met en avant les points critiques et verrous technologiques synthétisés dans la problématique exposée dans la première partie, et comment l'application de notre approche permettrait de les minimiser.

Chapitre 6

Evaluation et mise en oeuvre dans des environnements industriels

La thèse s'est en partie déroulée en collaboration avec la société de services NORSYS. Cette société s'est spécialisée, depuis une dizaine d'années, dans les domaines de l'assurance, la banque, la santé.

Après un recueil des attentes quant aux promesses de meilleure productivité d'une approche composant, nous exposerons les résultats des études menées sur des projets de la société. Nous présentons ici trois études de cas qui exploitent l'approche CUP sous des angles différents. L'étude de cas 1 présente la normalisation d'un composant préfabriqué de LA POSTE offrant des fonctionnalités de vérification d'adresse. Elle montre l'avantage de notre approche pour mieux spécifier un composant de manière neutre vis-à-vis d'une technologie, pour une meilleure réutilisation et une réflexion sur l'indépendance vis-à-vis d'autres composants, comme ici, des sources de données. L'étude de cas 2 montre, dans un système d'information simple de gestion du personnel d'une entreprise, l'application de notre approche pour concevoir deux composants indépendants possédant des relations fonctionnelles. La troisième étude de cas montre les intérêts potentiels de notre approche dans un système d'information à large échelle, dans le cadre de refonte de système d'information des VOIES NAVIGABLES DE FRANCE.

6.1 L'existant et les objectifs de la société Norsys

Que peut apporter une approche à base de composants dans une société de services telle que NORSYS ? La réponse à cette question validera notre travail.

Une meilleure **réutilisabilité** permet de gagner du temps et de réduire les coûts de développement. Les projets menés par la société sont souvent réalisés au forfait, c'est à dire dans les locaux de la société, et non dans les locaux du client. Cela implique des engagements dans les termes suivants :
- **délai** : les délais signés en accord avec le client et les équipes de développement doivent absolument être les plus courts possibles et surtout respectés. La réutilisation de composant répond à cette problématique.
- **résultat** : c'est à dire répondre aux besoins exprimés par le client.
- **qualité** : un document nommé **plan d'assurance qualité** contient les termes du contrat entre le client et la société d'ingénierie en termes de qualité : livrables, parties prenantes, planning, etc. Un composant analysé, conçu, développé et testé dans un autre système est, tout au moins unitairement, de meilleure qualité.

La société NORSYS travaille depuis une dizaine d'années avec des client dits "fidèles" dans les domaines bancaire, d'assurance et de la santé. Les développements de nombreuses applications ont été réalisés. Chez un même client, les développements ont été réalisés sur des domaines fonctionnels différents, cependant il est constaté que le même concept métier pouvait être présent dans plusieurs de ces domaines. Par exemple, le *contrat* de VNF, la troisième étude de cas, est présent dans les domaines *Gestion du domaine fluvial* et *Gestion de la tarification*. La société NORSYS a construit des démarches de longues durées chez ses clients. La notion de composant logiciel prend alors tout son sens, c'est pourquoi il convient de bien les spécifier.

Des études ont été menées sur d'anciennes technologies, dites "client-serveur", afin d'abstraire les concepts métiers transversaux, dans des objets. Une couche objet métier a été pensée et développée dans un framework. La volonté est bien de ne pas réinventer ce qui existe d'un projet à l'autre. Un framework de composants techniques a été développé dans des technologies client/serveur. Ce framework contenait des composants techniques simples tels que des listes chaînées, service de trace, d'instanciation de segment (structure), etc. L'inconvénient dans cette approche est que ces objets métiers étaient surtout des structures de données, et les relations entre ces objets n'étaient pas étudiées. La réutilisation n'était garantie que par le nommage de ces structures et il n'y avait pas de représentation des spécifications de celles-ci.

Tout était à refaire lors du passage aux nouvelles technologies. La première problématique était la maîtrise de la technologie. De plus, les spécifications étaient en avance sur les éditeurs qui proposaient chacun leurs outils, qui ne respectaient pas les spécifications. Comment faire dans ces conditions pour garantir les critères de réutilisabilité des développements ? et d'indépendance vis-à-vis des plates-formes ?

Le langage Unified Modeling Language a permis dans un formalisme uniformisé, de répondre au besoins des clients. Les clients souhaitent en effet, tout en ayant différents prestataires de services, que tous les développements soient spécifiés de la même manière. Comment homogénéiser, à travers

la méthode, chez un client, la matérialisation des composants (éléments réutilisables) de son système d'information ?

La volonté de la société NORSYS est de développer des stratégies de valorisation des acquis des salariés. Dans les projets les salariés doivent être compétents non seulement dans la maîtrise de la technologie mais aussi, dans la maîtrise du métier de leurs clients. La volonté est de capitaliser sur les travaux réalisés qui ne sont pas que du développement et d'accroître l'expertise métier pour répondre au mieux aux besoins du client.

De plus en plus, les clients doivent rénover la totalité de leur système d'information. De l'étude du projet à la production, en passant par les activités des équipes internes au client, des problématiques nouvelles sont apparues. De nombreux prestataires ont agi sur le système d'information, en utilisant souvent des outils différents. Le personnel informatique des clients est parti en retraite, les développements n'étant pas correctement documentés, il y a une perte de connaissance du système d'informations, qui se dégrade. Les projets de refontes débutent par une cartographie des processus métiers, en amont des développements des nouvelles applications. Cette source permet de réfléchir à l'identification de composants métiers dès les premières phases du cycle de vie du logiciel.

De nombreuses interrogations sont soulevées par une approche composant pendant toutes les étapes du cycle de vie du logiciel : l'analyse, la conception et le développement d'une partie d'un système d'information. Les rôles dans un projet, ont des demandes. Les analystes doivent adopter un formalisme permettant de cadrer leur travail. Il doit leur permettre d'identifier un composant métier, de le matérialiser dans le dossier d'analyse. Les concepteurs doivent savoir comment développer un composant spécifié, externaliser les paramètres. Comment le développer dans un environnement EJB ? par un seul EJB, un couple EJB session et EJB entité ? L'architecte, bénéficie d'une vue globale. Il est responsable des normes, du packaging. Les équipes d'industrialisation ou de production doivent produire des fiches, un versionning, régler les conflits.

Enfin, l'évolutivité du composant est un aspect important. Comment y ajouter une fonctionnalité, changer l'implémentation ?

6.2 Étude de cas 1 : Le composant d'Adresse de la poste (SNA2002)

6.2.1 Un composant sur l'étagère

Vision du composant Adresse

La poste, par le biais du Service National de l'Adresse[1] fournit gratuitement le **Composant Adresse**, notamment en version java Bean. Il permet le contrôle des adresses saisies via Internet. Ce composant effectue les corrections liées à la norme AFNOR de l'adresse, et contrôle le code postal et la localité.

L'avantage de la fourniture d'un composant contenant un fichier d'adresses postales offre aux sites de commerce électronique la garantie que les adresses saisies par les clients ou fournisseurs soient correctes. Ainsi, cela améliore le lien entre société de commerce et leurs clients, évite les NPAI (N'habite Pas à l'Adresse Indiquée), facilite le travail de distribution du courrier par la Poste. De plus, la génération de courrier avec une adresse normée offre des avantages financiers quant aux frais postaux. Le développement d'une application destinée à recueillir des adresses peut donc inclure ce composant gratuitement, pour garantir une qualité de service.

Le composant est fourni avec une documentation décrivant les règles de La Poste pour améliorer la qualité de l'adresse.

1. Des informations ordonnées du nominatif (nom et raison sociale) à la localité du destinataire.
2. 6 lignes maximum : il ne doit pas y avoir de lignes blanches dans une adresse.
3. 38 signes (caractères et espaces compris) maximum par ligne. Un espace doit figurer entre chaque mot.
4. Aucun signe de ponctuation ni de souligné à partir de la ligne "numéro et libellé de la voie". Les virgules suivant les numéros de rue sont donc à proscrire.
5. La dernière ligne doit toujours être en majuscules. Il est conseillé de saisir les trois dernières lignes (numéro et libellé de voie, lieu-dit ou boite postale et ville) en majuscules.
6. Lignes d'adresse alignées à gauche.

Fonctionnalités offertes

Les contrôles effectués par le composant sont les suivants :

1. **Complément de remise (ETAGE, ...)** Vérification de l'absence de mots-clés appartenant à d'autres lignes (ex : ETAGE est bien en ligne 2, tandis que BATIMENT doit se trouver en ligne 3)
2. **Complément de distribution (BATIMENT, ...)** Vérification de l'absence de mots-clés appartenant à d'autres lignes (ex : RESIDENCE est bien en ligne 3, tandis que APPARTEMENT doit se trouver en ligne 3)
3. **Numéro et libellé de la voie** Vérification de l'absence de mots-clés appartenant à d'autres lignes (ex : AVENUE est bien en ligne 4, tandis que LOTISSEMENT doit se trouver en ligne 3)

[1] L'ensemble des informations fournies dans ce document et le composant lui-même, sont la propriété du Service National de l'Adresse de La Poste et disponible, ainsi que la notice d'utilisation sur *http ://www.laposte.fr/sna/*

4. **Lieu-dit ou service de distribution** Vérification de l'absence de mots-clés appartenant à d'autres lignes (ex : BP est bien en ligne 5, tandis que RUE doit se trouver en ligne 4)
5. **Code Postal Localité** Concordance entre le code postal et la localité à l'aide du référentiel des codes postaux et CEDEX. Vérification de l'existence du code postal ou du CEDEX. Vérification de l'existence de la localité. Proposition éventuelle d'une liste de localités ayant le même code postal. Lorsque l'on saisit le code postal, et le début de la localité cherchée, le libellé de la localité est automatiquement complété. Si un lieu-dit est détecté, il est remonté en ligne 5
6. **Pays** Si le pays est la France ou si le champ n'est pas renseigné, les lignes 1 à 6 sont vérifiées. Lorsqu'il s'agit d'un autre pays, le logiciel vérifie seulement que la ligne 1 est renseignée. Il compare en outre le pays saisi à une liste de pays au sens d'"acheminement international". Par exemple pour l'Écosse, il faut saisir Royaume Uni.

Exemples de corrections effectuées par le composant Adresse

Adresse saisie	Adresse corrigée
Plusieurs corrections sur une adresse L2 :4 résidence les Ursules L3 :28, rue du port L4 : L5 : L6 : 33740 arres	 L2 : L3 :4 RESIDENCE LES URSULES L4 :28 RUE DU PORT L5 : L6 : 33740 ARES
Libellé de la localité complété L6 : 34250 PALAVA	 L6 : 34250 PALAVAS LES FLOTS
Lieu-dit remonté en Ligne 5 L5 : L6 : 33380 CAUDOS	 L5 : CAUDOS L6 : 33380 MIOS
Correction du Code postal L6 : 33700 POMPIGNAC	 L6 : 33370 POMPIGNAC
Restructuration des lignes L2 : BATIMENT A APP 12 L3 : RES LES LILAS	 L2 :APPARTEMENT 12 L3 :BATIMENT A RESIDENCE LES LILAS

Documentation technique du composant Adresse

Le composant Adresse est un ensemble de Java Beans et de classes Java, prévu pour fonctionner dans un environnement Java 2 Enterprise Edition.
- **BeanControlePays** qui effectue des contrôles sur la saisie d'un pays
- **BeanControleCP** qui effectue des contrôles sur la saisie d'un code postal, de la localité et d'un lieu-dit
- **BeanControleLigne** qui restructure les lignes d'adresse 2 à 5

Ce composant est fourni avec la description de ces beans, contenant chacun un ensemble de méthodes.

6.2.2 Normalisation CUP du composant

Ce composant va nous permettre de valider notre formalisme de **composant logique**. Afin d'intégrer ce composant Adresse aux modèles CUP et à la démarche sous-jacente pour construire un système intégrant ce composant, nous présentons sa normalisation au formalisme CUP dans la suite.

Critique du composant Adresse

Le composant contient des Java Beans. Il n'y a pas d'interface, le composant, mal spécifié n'est pas facilement réutilisable. Un deuxième constat est que le composant Adresse est livré avec des fichiers contenant les renseignements nécessaires aux vérifications. Il serait intéressant de rendre autonome ce composant en offrant la possibilité d'utiliser n'importe quelle source de données comme entrée.

Réalisation

Comme nous l'avions précisé, la démarche CUP est présentée dans un ordre séquentiel de ses enchaînements d'activités. Nous allons voir, dans cette étude de cas, que les enchaînements sont abordés dans un ordre dépendant des informations disponibles.

Une première étape est de transformer les Java Beans en **interfaces fournies** (cf. figure 6.1). Pour cela, la documentation du composant fournit l'ensemble des méthodes gérées par chaque bean.

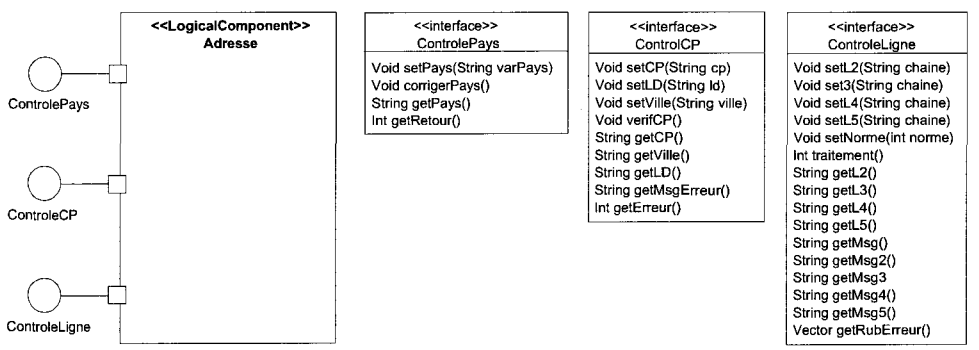


FIG. 6.1 – Interfaces fournies du composant logique Adresse

La vue de cas d'utilisation est très simple (cf. figure 6.2), étant donné que nous avons déjà les services rendus par le composant exprimé par les méthodes de l'interface. Pour trouver l'acteur **Agent vérificateur**, il faut imaginer la plus petite application utilisant uniquement ce composant. Une personne ou un système peut vérifier les adresses grâce à celui-ci.

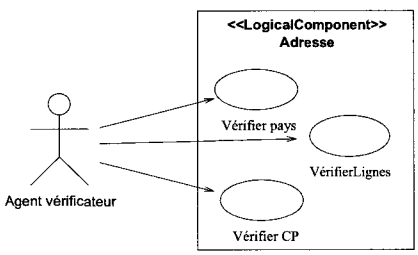


FIG. 6.2 – Vue des cas d'utilisation du composant logique Adresse

Un exemple de vue d'interaction est la suivante (cf. figure 6.3).

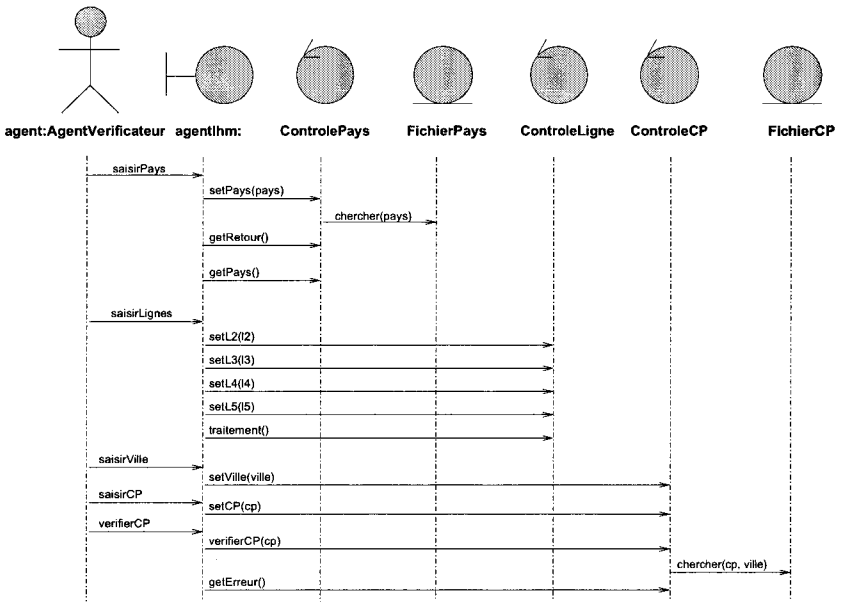


FIG. 6.3 – Exemple de vue d'interaction du composant logique Adresse

Il convient de rendre indépendant ce composant des sources d'information nécessaires pour faire les vérifications. Pour cela, dans les vues d'interactions et de conception de composants (cf. figure 6.4), on ajoute respectivement des rôles externes de contrôle et des interfaces requises. La vue d'interaction représente un scénario dans lequel, on interroge des entités externes, plutôt que les données internes fournies avec le composant Adresse.

Nous avons obtenu, dans cette étude, une représentation normalisée du composant. Les résultats obtenus peuvent ainsi servir à intégrer le composant Adresse dans l'analyse d'un système d'information qui réutilise ce composant préfabriqué. La génération EJB++ peut fournir une première implémentation du composant pour l'intégrer à la partie métier du développement d'une application contenant ce service. On bénéficie alors directement des avantages d'une architecture et approche MDA. Enfin, la souplesse d'évolution du système est garantie par les interfaces requises et fournies.

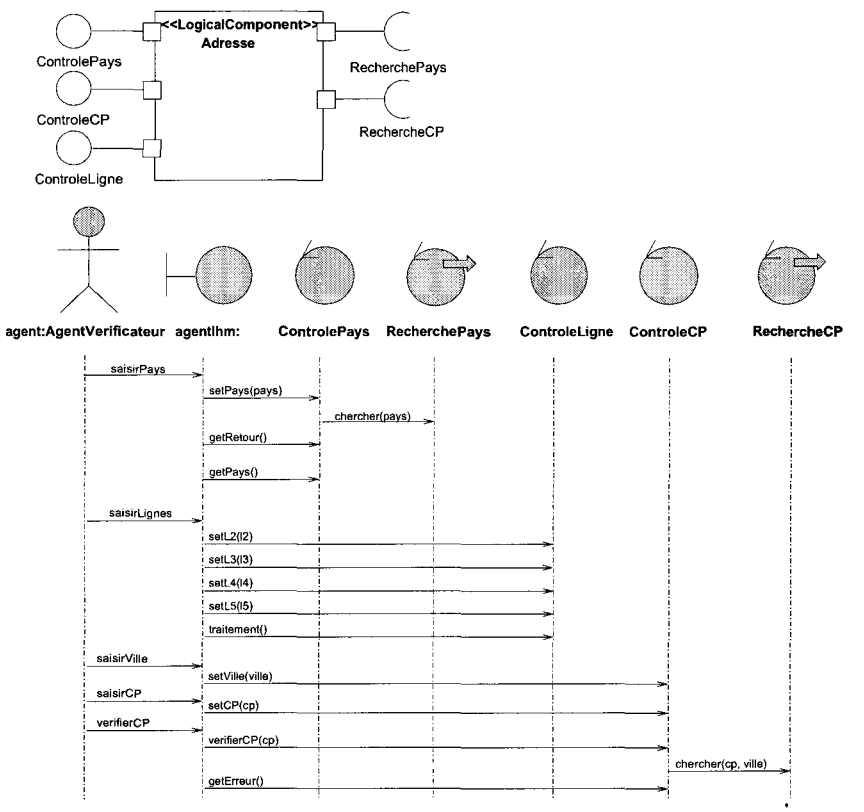


FIG. 6.4 – Vue partielle de conception de composant et une vue d'interactions

6.3 Etude de cas 2 : Gestion administrative du personnel

Le système de gestion du personnel est un outil informatique interne au système d'informations de la société NORSYS. Il doit gérer les demandes de congés des employés, les comptes rendus d'activités, la délivrance des tickets restaurants.

6.3.1 Descriptif de l'application

Le suivi administratif des congés payés et congés temps libre

Tout salarié a droit à 25 jours de **congés ouvrés**, soit 2,08 jours de congés par mois, qui se cumulent pour l'année suivante. La période de congés se situe du 1er Juin au 31 Mai de l'année suivante. Selon l'accord ARTT, la réduction du temps de travail a été annualisée, soit 20 jours de **congés temps libre** ouvrés ou 1,66 jours de congés temps libre par mois, à prendre dans l'année en cours, avant le 31 Mai sous peine d'être perdus. Selon l'accord ARTT, 5 jours ont été réservés à la **formation**. Le compteur sera mis à jour régulièrement par le comptable, dès la formation suivie. La société accorde des jours **exceptionnels** de congés. Dans ce compteur viennent s'ajouter les jours accordés pour des événements exceptionnels (mariage, naissance, ...). Ces jours sont calculés au prorata du temps de présence dans l'entreprise.

Afin de ne pas être pris au dépourvu, il est prévu 3 planifications sur l'année :
- le 15 Avril au plus tard, pour la période du 1er Juin au 31 Octobre
- le 15 Octobre au plus tard, pour la période du 1er Novembre au 28/29 Février
- le 15 Février au plus tard, pour la période du 1er Mars au 31 Mai

Remplissage d'une demande de congés

Le calcul des congés se fait en jours ouvrés (jours travaillés). Le programme répartit lui-même les congés en fonction des catégories et des règles. Pour la période Juillet/Août, 20 jours de congés doivent être pris obligatoirement sauf cas particulier faisant l'objet d'une validation ou d'une personne embauchée en cours d'année, pour qui les jours obligatoires seront calculés au prorata du temps de présence dans l'entreprise. Par exemple, une personne entrée le 1er Juillet a 18,5 jours de temps libre, elle devra prendre 8 jours de congés. Les demandes ne sont prises en compte que lorsqu'elles sont validées par le responsable de l'employé. Une fois validée, une demande ne peut plus être modifiée ou annulée par l'employé. Seul le responsable peut encore la modifier ou la supprimer. Quand une demande est prise en compte, la période de congés est inscrite dans les rapports d'activité de l'employé. Les rapports d'activité d'un employé contiennent, pour chaque jour travaillé, les différentes activités libellées de celui-ci.

Quelques exemples de demandes :
- 1 semaine de congés, soit 5 jours du Lundi au Vendredi
- 2,5 jours avec une après-midi, soit 2,5 jours du Mercredi à partir de l'après-midi au Vendredi
- 2,5 jours avec une matinée, soit 2,5 jours du Jeudi au Lundi matin inclus

Absence imprévue

Dans tous les cas, l'employé doit impérativement prévenir le secrétariat par téléphone ou par tout autre moyen, dès le début de son absence. Dès son retour dans l'entreprise, il doit remplir une demande

de congés.

En cas d'arrêt maladie, dans un premier temps, le salarié est tenu de prévenir l'employeur de son arrêt maladie par l'intermédiaire du secrétariat. Les volets 1 et 2 de l'arrêt de travail délivré par le médecin, doivent être envoyés dans les 48h par les soins du salarié à son centre de sécurité sociale. Le troisième est à transmettre au secrétariat. Lors de la reprise du travail du salarié, il doit avertir le comptable afin qu'il puisse établir le document servant au paiement de ses indemnités journalières.

6.3.2 Analyse des besoins

Deux acteurs sont identifiés : l'employé et le responsable. Les fonctionnalités attendues par chacun de ces acteurs sont :
- pour l'employé : demander, modifier et annuler une demande de congés
- pour son responsable : valider, refuser, supprimer une demande de congés de l'employé
- pour l'employé : saisir, modifier ses rapports d'activité
- pour son responsable : valider, modifier les rapports d'activité de l'employé

Deux pans fonctionnels sont identifiés et deviennent composants logiques candidats : les composants conges et rapportActivite. Et une relation de dépendance est identifiée (cf. figure 6.5).

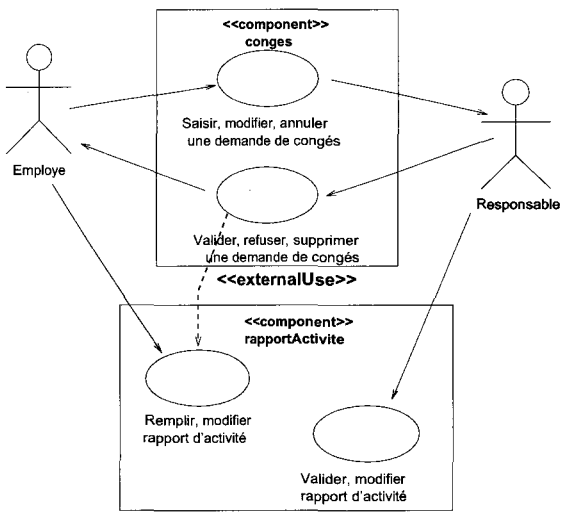


FIG. 6.5 – Vue des cas d'utilisation du système de gestion administrative du personnel

La relation **externalUse** modélise la dépendance des composants logiques. Plus précisément, ils expriment que lors de la validation, la modification ou la suppression d'une demande de congés par un responsable, les rapports d'activité de celui-ci sont modifiés.

Pour illustrer en premier lieu, la description des cas d'utilisation puis ensuite, dans la section suivante, la réalisation des cas d'utilisation, la validation d'une demande de congés est choisie pour montrer comment se consolide la dépendance externe de cas d'utilisation.

Voici une description du cas d'utilisation **Valider/Modifier/Refuser une demande de congés**

Cas d'utilisation Valider, modifier et refuser une demande de congés

Résumé : description de la validation, la modification et le refus d'une demande de congés d'un employé par son responsable
Acteurs : Le responsable (principal), l'employé (secondaire)
Pré-conditions :
- le responsable est averti qu'une demande de congés d'un de ses employés a été saisie
- le responsable choisit de valider des congés

Scénario nominal

1. le responsable sélectionne un employé
2. le système lui propose une liste de congés à valider pour cet employé
3. le responsable choisit une demande et la valide
4. le système enregistre la validation
5. le système modifie les rapports d'activité de l'employé
6. le système avertit l'employé que sa demande est validée

Scénarii alternatifs
A1 : refus de congés
le scénario A1 commence à l'étape 3 du scénario nominal

1. le responsable choisit une demande et la refuse
2. le système demande la raison du refus
3. le responsable indique la raison du refus
4. le système avertit l'employé que sa demande est refusée

Post-condition
- la demande ou le refus sont enregistrés

6.3.3 Réalisation des cas d'utilisation

La figure 6.6 représente le scénario nominal du cas d'utilisation **Valider, modifier et refuser une demande de congés**. Dans cette étude de cas, des cas d'utilisation ont été regroupés ensemble. En effet, la validation, la modification et le refus sont fonctionnellement très proches. Les interactions qui en découlent sont quasi-identiques. C'est le cas typique où il n'est pas nécessaire de réaliser la totalité des scénarii d'un cas d'utilisation pour obtenir toutes les fonctionnalités d'un système. Une fois la validation réalisée, il suffira d'ajouter au modèle des éléments tels que des attributs, méthodes, interfaces, ...

6.3.4 Conception du composant conges

La conception de composant (cf. figure 6.7) représente les structures internes et externes du composant **congés**.

6.3.5 Vue d'assemblage du système de gestion administrative du personnel

Cette vue peut illustrer un composant et ses interfaces, ou représenter globalement un système d'information conçu par assemblage de composants (cf. figure 6.8).

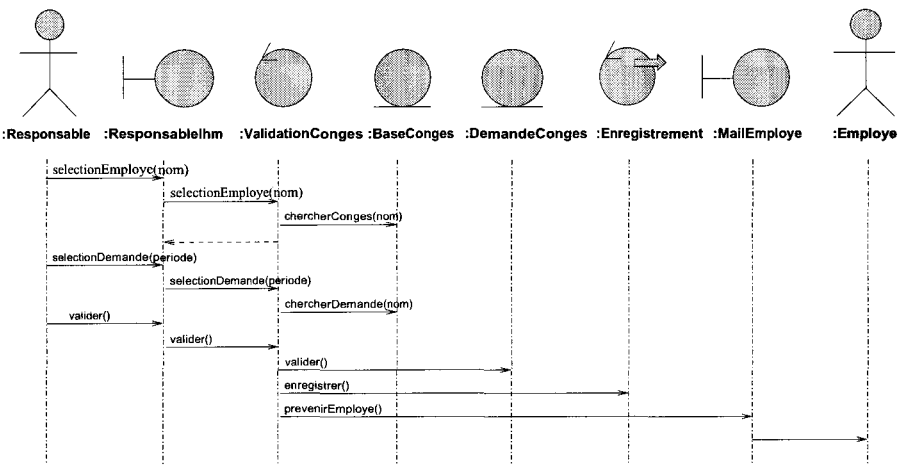


FIG. 6.6 – Vue d'interaction du cas d'utilisation Valider demande de congés

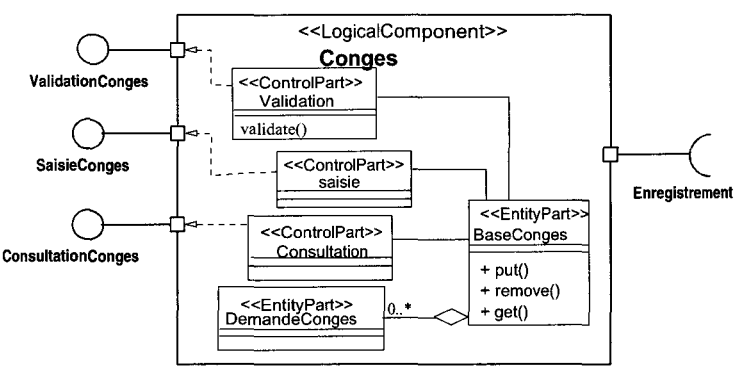


FIG. 6.7 – Vue de conception de composant

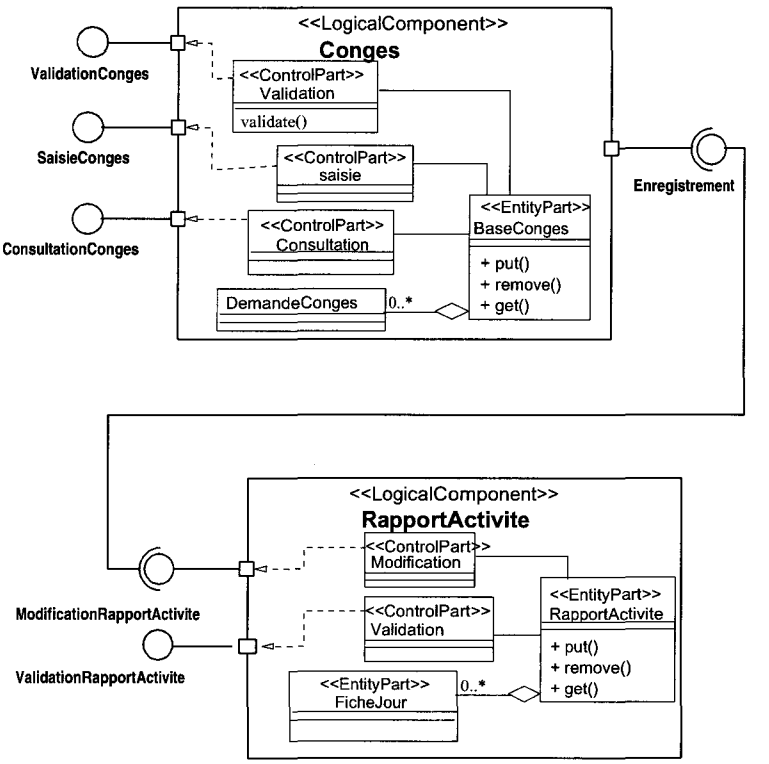


FIG. 6.8 – Vue d'assemblage du système de gestion administrative du personnel

Cette étude de cas montre l'avantage du modèle et du processus sous-jacent qui guide les acteurs d'un projet pour obtenir un système à base de composants de qualité. La cohérence des différentes vues du modèle lie les éléments de vues différentes. Enfin, la notion de composant logique omniprésente, cadre le processus.

6.4 Etude de cas 3 : Le système d'information des Voies Navigables Françaises VNF

6.4.1 Présentation du contexte du projet

Vision

VNF est une organisation privée sous tutelle de l'état. Elle gère environ les trois quarts des voies d'eau françaises (rivières, fleuves et canaux). Après dix ans d'existence, le système d'information s'est construit peu à peu sans démarche cohérente. Ne bénéficiant pas de grosse structure informatique, le système d'information est un ensemble de petites applications remplissant chacune une fonctionnalité. Le système est contraint par environ 500 utilisateurs potentiels, qui sont les utilisateurs internes de VNF, et les fonctionnaires d'état délégués à VNF qui ont souvent plusieurs missions à réaliser en même temps. VNF gère essentiellement la circulation sur les voies d'eau et le domaine public fluvial, c'est-à-dire, l'ensemble des berges, terrains et équipements qui bordent les voies d'eau.

L'application étudiée gère le domaine public fluvial. Lorsqu'une personne veut occuper un terrain, il doit signer un contrat avec VNF. Le terrain peut contenir des équipements, comme, par exemple, un ponton. L'occupation est soumis à une tarification. Les utilisateurs de terrains peuvent être des particuliers qui veulent amarrer leur bateau à un ponton ou entretenir un potager par exemple.
Dans le cas où l'utilisateur du terrain est un industriel et que l'occupation concerne une "prise et rejet d'eau", une tarification spéciale est appliquée : la taxe hydraulique. Cette taxe donne lieu à des recettes qui sont versées à l'état. Celui-ci verse des subventions prélevées sur les bénéfices à VNF. VNF a alors comme mission la valorisation du domaine fluvial.

La problématique est l'uniformisation des diverses applications développées dans ce système d'information. Ces applications contiennent des concepts métiers communs. L'application *Gestion des actes* qui permet d'établir un contrat entre un client et VNF pour l'occupation des terrains. Toute la tarification liée aux actes est faite "à la main", il n'y a pas de suivi possible, pas de contrôle sur les tarifs appliqués et les versements effectués. L'application *Impression des contrats* est faite avec Microsoft Word, avec un modèle de documents. Les modifications s'effectuent directement dans le document Word. Il n'y a pas d'historique et donc de suivi. L'application *tarification* est saisie dans un outil de comptabilité où l'on saisit les clients, la tarification et le type d'usage. Cette application de comptabilité doit être conservée dans le futur système, à la demande du client. Une application Microsoft Access permet de référencer les terrains et équipements pour inventorier le domaine. Une base de données Oracle et une application Visual Basic gère la taxe hydraulique.
Toutes les applications sont indépendantes, il n'y a pas de suivi des clients, des contrats. La traçabilité des dossiers est inexistante. Par exemple, il peut être intéressant, en outre, d'isoler les bons clients, de demander à renouveler un contrat si celui-ci est en fin de validité, de détecter les terrains inoccupés. Sous tutelle de l'état, une partie des bénéfices doit être utilisée pour réaménager le domaine fluvial. Ils doivent donc en faire un maximum afin de valoriser au mieux le domaine. Cependant, VNF ne dispose pas de recoupements entre les différents domaines fonctionnels, par exemple, il ne dispose pas directement des informations sur l'occupation des terrains dans certaines régions.
L'application de gestion du domaine, stratégique, est transversale aux domaines métiers suivants : gestion des clients, gestion des contrats, gestion de la taxe hydraulique, de la tarification et de l'inventaire. Les dépendances entre ces domaines sont identifiées comme les contrats clients, des requêtes sur

le système, comme par exemple le plus gros client par rapport à la tarification, les régions les moins valorisées, etc.

L'approche composant dans la refonte de ce système doit apporter les fonctionnalités suivantes : le lien entre les modèles, la production de tableaux de bord, la gestion des concessions (contrats de plus longue durée qui demandent une gestion différente), le dressement d'amende pour l'occupation des terrains sans titre, le contrôle sur l'application des tarifications suivant l'occupation. Cette étude montre l'intérêt stratégique de l'application de notre approche à un système large. Le point clé de cette stratégie étant l'étude des dépendances entre des composants d'un système d'information en amont qui implique des investissements lourds, mais promet des bénéfices plus importants.

Démarche mise en place dans un projet NORSYS

Le projet réalisé par la société Norsys est un projet d'un an et 2400 jours/hommes par une équipe de 10 personnes en moyenne et jusqu'à 20 personnes par période. L'objectif est double : mettre en place le socle applicatif (l'architecture) basé sur des technologies J2EE et réaliser l'application. Norsys est intervenue dans les phases de conception, la réalisation et la mise en place de l'architecture technique et le déploiement. Le projet a suivi une démarche de type Processus Unifié. Une première étape, réalisée par une autre SSII a permis la réalisation de documents d'analyse. Ces documents contiennent les diagrammes UML ainsi que les documents textuels de description (cas d'utilisation, règles de gestion, préconisations techniques). Le projet a été découpé en modules (cf. figure 6.9). Ces modules correspondent aux différentes applications indépendantes de l'ancien système d'information. Ce découpage a été conservé car il est semblable au découpage qui aurait été produit à partir de l'étude des besoins. Chaque module est décrit par des cas d'utilisation qui correspondent chacun à une fonctionnalité.

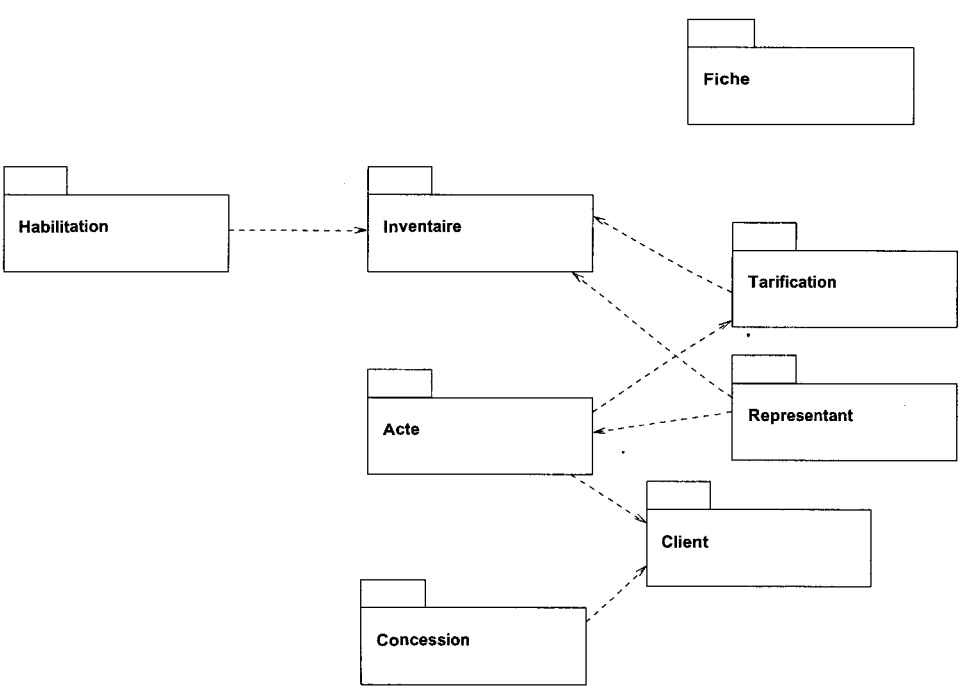


FIG. 6.9 – Les différents modules du système d'information VNF

La première phase du projet a consisté en une phase d'appropriation métier, basée sur l'étude des documents d'analyse produits (cf. figure 6.10). Suite à cette étape, des manques ont été identifiés dans l'analyse des besoins et une phase de revue des documents produits a été menée. Les diagrammes d'activités ont été complétés, les cas d'utilisation étoffés, les règles de gestion peu nombreuses initialement ont été établies. Tous les diagrammes ont été repris sur l'outil de modélisation Rational Rose. Parallélement, l'architecture applicative a été élaborée. Le travail de conception a été partagé entre quatre personnes (environ 300 jours).

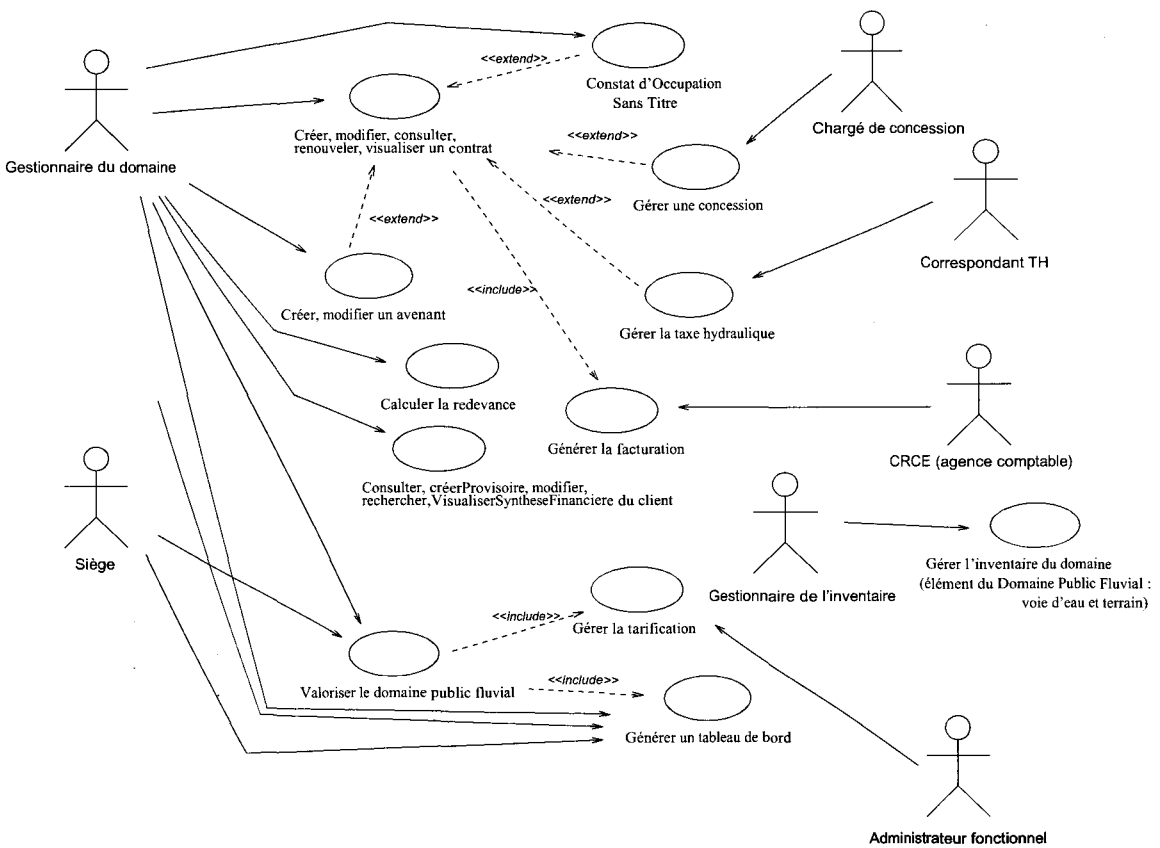


FIG. 6.10 – Extrait du diagramme de cas d'utilisation du projet VNF

A cause des délais relativement courts pour réaliser la reprise de l'analyse, les points de synchronisation entre les différents travaux étaient peu fréquents. Les différents modules ont donc été conçus en parallèle et la cohérence pas forcément maintenue. Les dépendances entre fonctionnalités de modules différents ne sont pas explicites. Aucune vue d'ensemble ne permet de les retrouver. Elles sont exprimées dans les scénarii de cas d'utilisation décrits dans les documents textuels. Pour chaque cas d'utilisation ce document contient les scénarii correspondants. La phase de conception a permis de modéliser les scénarii par des diagrammes de séquence (cf. figure 6.11).

Description du cas d'utilisation *Créer, modifier, consulter contrat*

Acteurs
L'utilisateur interne

Scénario nominal *Créer contrat*
sn1. L'utilisateur interne recherche un client
sn2. Le système indique que le client n'existe pas
sn3. L'utilisateur demande à créer le client
sn4. Le système crée le client et renvoie un identifiant correspondant au nouveau client
sn5. L'utilisateur interne recherche un élément du DPF
sn6. Le système trouve l'élément
sn7. L'utilisateur interne saisie les données du contrat relative à l'occupation (condition particulière, durée du contrat)
sn8. L'utilisateur interne choisit un objet tarifable (ex : ponton) de l'élément du DPF (ex : terrain)
sn9. L'utilisateur interne consulte la fiche associée
sn10. L'utilisateur interne saisit les conditions particulières de tarifcation de cet objet en modifiant la fiche
sn11. Le système calcule le montant de la redevance
sn12. Le système génère les lignes comptables de l'élément du DPF

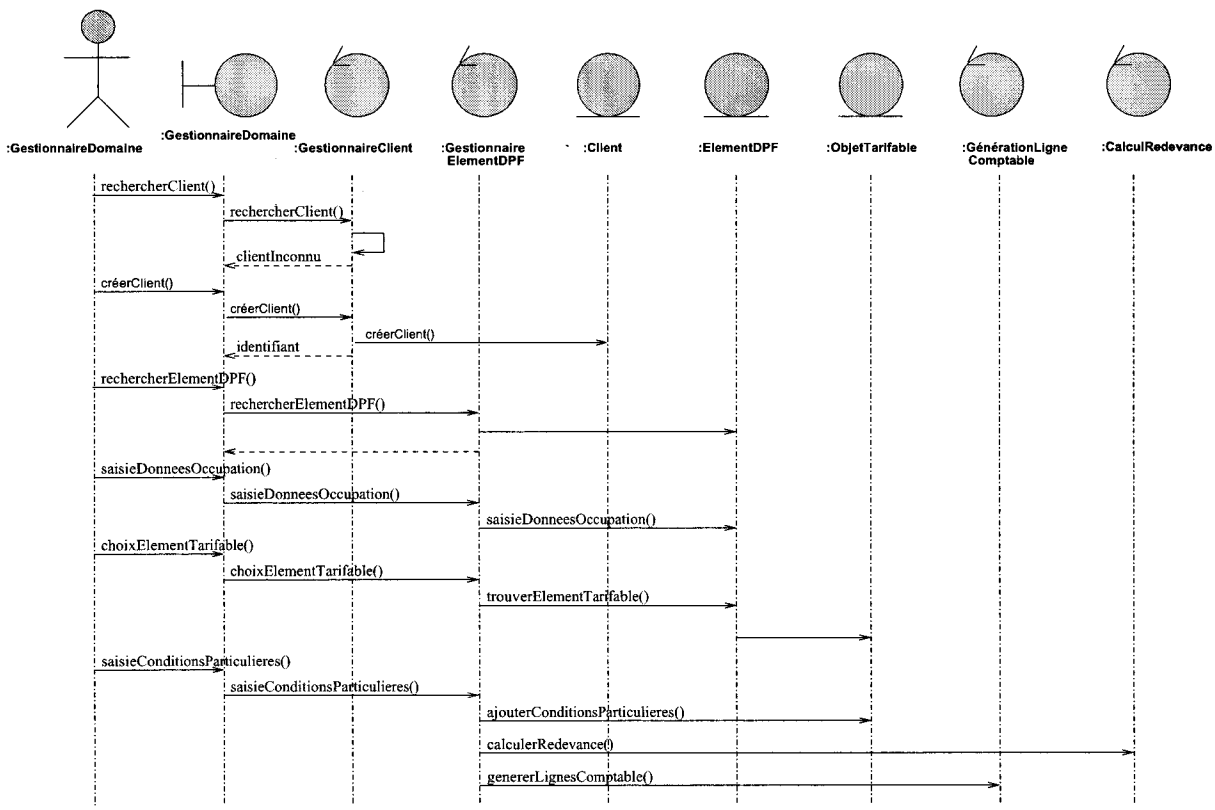


FIG. 6.11 – Diagramme de séquence du scénario Créer un acte

Dans la vue logique, un diagramme de classes a été construit pour représenter uniquement les objets métiers (cf. figure 6.12[2]). L'implémentation ne nécessite pas une analyse exhaustive de toutes les classes du système. Seules les classes contenant des informations et du comportement métier sont modélisées. Le code est ensuite généré à partir des diagrammes de séquence et de classes.

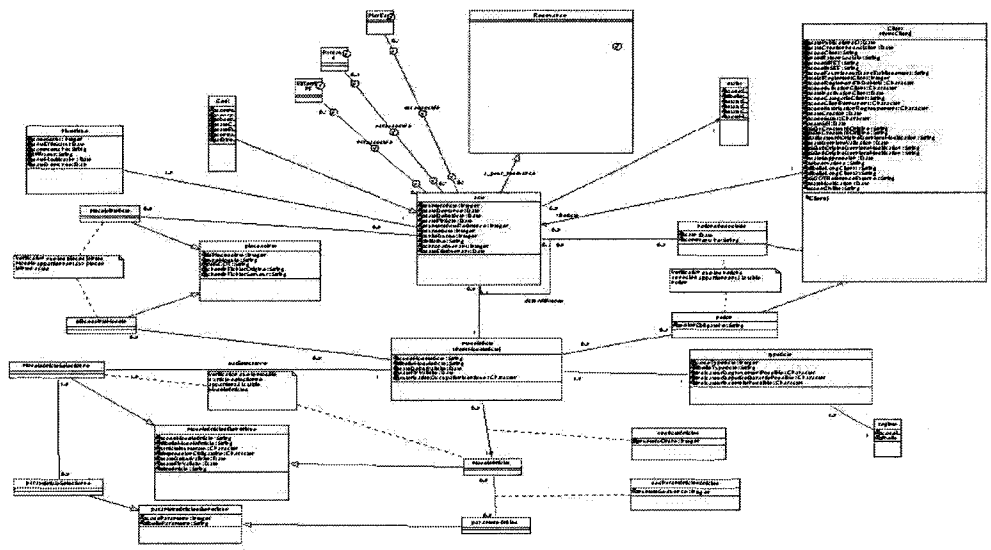


FIG. 6.12 – Extrait du diagramme de classe, contenu du module Acte

La cohérence entre les différentes vues ne sont pas vérifiées par l'outil Rational Rose et ce défaut est amplifié par la mise en parallèle de l'étude des différents modules.

La phase de réalisation a consisté d'abord à définir l'architecture technique sur laquelle générer le code. Pour celà, un guide méthodologique a été réalisé. Il contient l'architecture logicielle du projet, et ses différents composants et décrit l'intégration de frameworks open-sources existants. L'architecture est de type Model-View-Controller. La figure 6.13 schématise les principales caractéristiques de l'architecture logicielle du projet VNF, représentative des architectures dites "web" actuelles.

Les différents composants techniques à développer dans le cadre du projet se basent sur une architecture fonctionnelle à trois niveaux :

- **Un tier dédié à la présentation** (Interface Homme-Machine) autour de la technologie Java. Ce niveau a en charge toute la partie mise en forme des informations en provenance ou à destination de l'application, ainsi que la gestion des interactions avec l'utilisateur (affichage des données par exemple).
- **Un tier dédié à l'application** qui regroupe l'ensemble des composants associés à des besoins applicatifs non généralisables au niveau de l'entreprise. Il est constitué notamment de composants de contrôle qui gèrent les interactions entre le niveau présentation et le niveau métier. Ces composants sont souvent appelés tâches applicatives ou processus applicatifs.

[2] *Ce diagramme est volontairement non lisible dans un soucis de confidentialité*

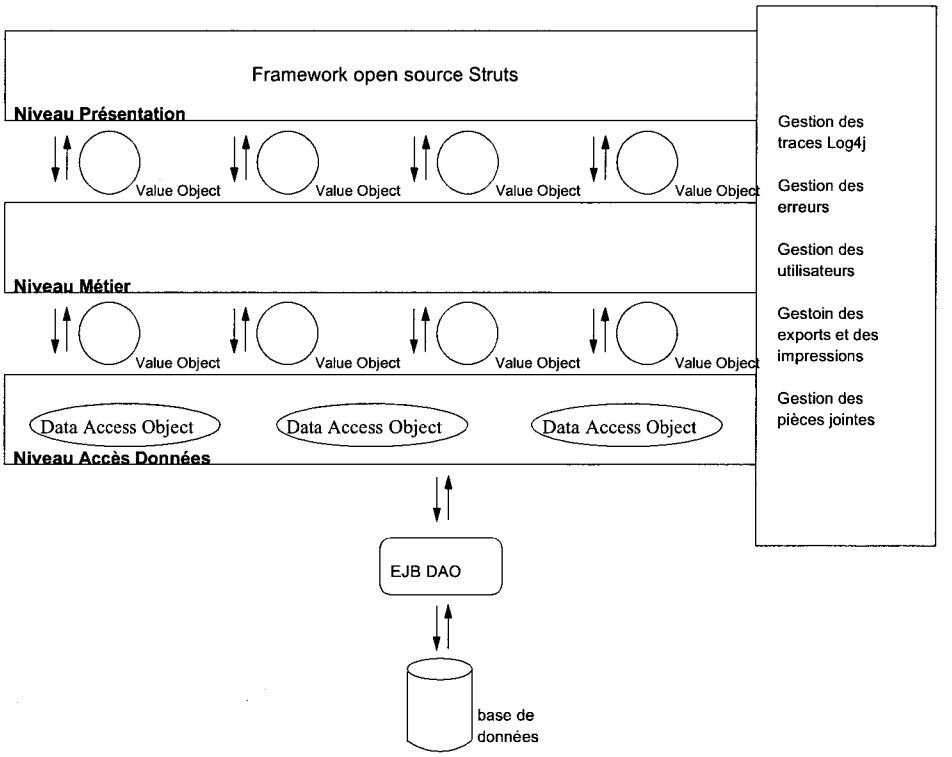


FIG. 6.13 – Architecture du projet VNF

- **Un tier dédié à l'accès aux données** géré via des EJB *Data Access Object* : ce niveau regroupe les composants génériques des différents domaines (concepts métiers) de l'entreprise. On y trouve les composants généraux partageables par plusieurs applications du système d'information global : des composants de type entité et des composants de type processus ou traitement mettant en jeu plusieurs entités.

Enfin, dernière phase du projet, la transition consiste en des tâches de Tierce-Maintenance-Applicative, la maintenance de l'application mise en production et éprouvée par les utilisateurs finaux. Cette phase permet de corriger les dernières anomalies non détectées par des tests de mises en production scénarisés par les cas d'utilisation. Ainsi, lorsqu'un utilisateur détecte une anomalie, il la signale, et une équipe de maintenance détecte les éléments mis en jeux ainsi que les traitements qu'il est nécessaire de corriger. Pour cela, il doit être possible de remonter aux besoins à partir de l'erreur, pour reparcourir les scénarii et vérifier que le système fonctionne complétement. L'équipe de TMA est souvent distincte des équipes de production.

6.4.2 Application de CUP au projet

Nous avons appliqué la démarche CUP sur cette application avec pour objectif de montrer les intérêts de CUP par rapport à l'approche de la société Norsys. Dans ce chapitre ; nous nous sommes focalisés sur la conception du composant **Acte**. Ce composant est au centre du système d'information, il est le plus intéressant du point de vue des dépendances avec les autres composants du système.

Vue des cas d'utilisation

La vue des cas d'utilisation permet de représenter les principales fonctionnalités du système sous forme graphique dans une vue globale (cf figure 6.14). L'introduction de composants logiques permet de partitionner le système en différents modules et de travailler sur les dépendances entre ces modules. L'apport de CUP ne se situe pas dans le découpage de l'architecture du système d'information VNF. Celui-ci a été réalisé en respectant la séparation des applications indépendantes existantes. Par contre notre approche a l'avantage de mieux structurer les cas d'utilisation et de montrer explicitement les relations de dépendances fonctionnelles entre composants logiques, ceci grâce à l'utilisation de la relation **externalUse** entre cas d'utilisation.

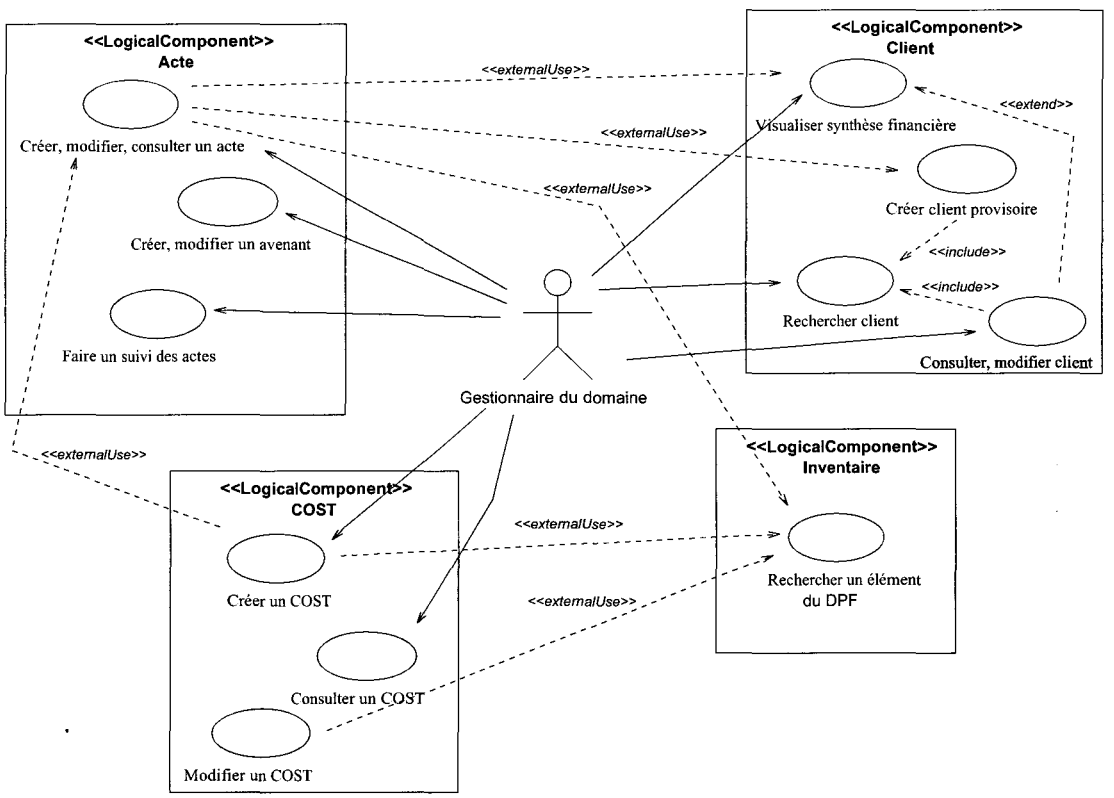


FIG. 6.14 – Extrait de la vue CUP des cas d'utilisation du projet VNF

Préalablement, les dépendances fonctionnelles sont exprimées dans les scénarii de manière textuelle et informellement à l'aide du signe (->) (cf scénario ci-dessous).

Description du cas d'utilisation *Créer, modifier, consulter acte*

Acteurs
L'utilisateur interne

Scénario nominal *Créer acte*
sn1. L'utilisateur interne recherche un client
sn2. Le système indique que le client n'existe pas
sn3. L'utilisateur demande à créer le client
sn4. Le système crée le client et renvoie un identifiant correspondant au nouveau client (->)
sn5. L'utilisateur interne recherche un élément du DPF (->)
sn6. Le système trouve l'élément
sn7. L'utilisateur interne saisie les données du acte relative à l'occupation (condition particulière, durée du acte)
sn8. L'utilisateur interne choisit un objet tarifable (ex : ponton) de l'élément du DPF (ex : terrain) (->)
sn9. L'utilisateur interne consulte la fiche associée
sn10. L'utilisateur interne saisit les conditions particulières de tarifcation de cet objet en modifiant la fiche
sn11. Le système calcule le montant de la redevance
sn12. Le système génère les lignes comptables de l'élément du DPF (->)

Dans CUP, la relation **externalUse** reliant deux cas d'utilisation de composants logiques distincts montre que le cas d'utilisation d'origine contient un scénario qui fait appel au service d'un autre composant. Certains scénarii peuvent s'avérer plus intéressants. Par exemple, le scénario *Créer Acte* permet de trouver et de consolider les dépendances du composant *Gestion des actes* avec d'autres composants du système d'information. L'avantage de cette vue par rapport au diagramme de cas d'utilisation classique est de fournir une vue globale du système d'information et d'expliciter les dépendances entre composants logiques à l'aide des cas d'utilisation.

Vue d'interactions

La vue d'interaction permet de modéliser chaque scénario d'utilisation sous la forme d'un diagramme de séquence UML (cf figure 6.15). La contribution de CUP est d'ajouter le rôle externalControlRole qui indique précisément, par un appel de méthode, représentant l'appel de service où se situe la dépendance. Ainsi, lors de la mise en production du logiciel, les erreurs dues à une mauvaise gestion des dépendances entre modules peuvent être identifiées directement en traçant les appels de méthodes. De plus, la vue d'interaction est cohérente avec la vue de cas d'utilisation. Ainsi, la présence de dépendances exprimées par la relation externalUse entre deux cas d'utilisation implique que dans un scénario du cas d'utilisation demandeur de service, il y a un rôle externe de contrôle. Si ce n'est pas le cas, la relation externe d'utilisation doit être supprimée de la vue de cas d'utilisation. L'outil CUPTool permet de vérifier cela en maintenant la cohérence entre les vues.

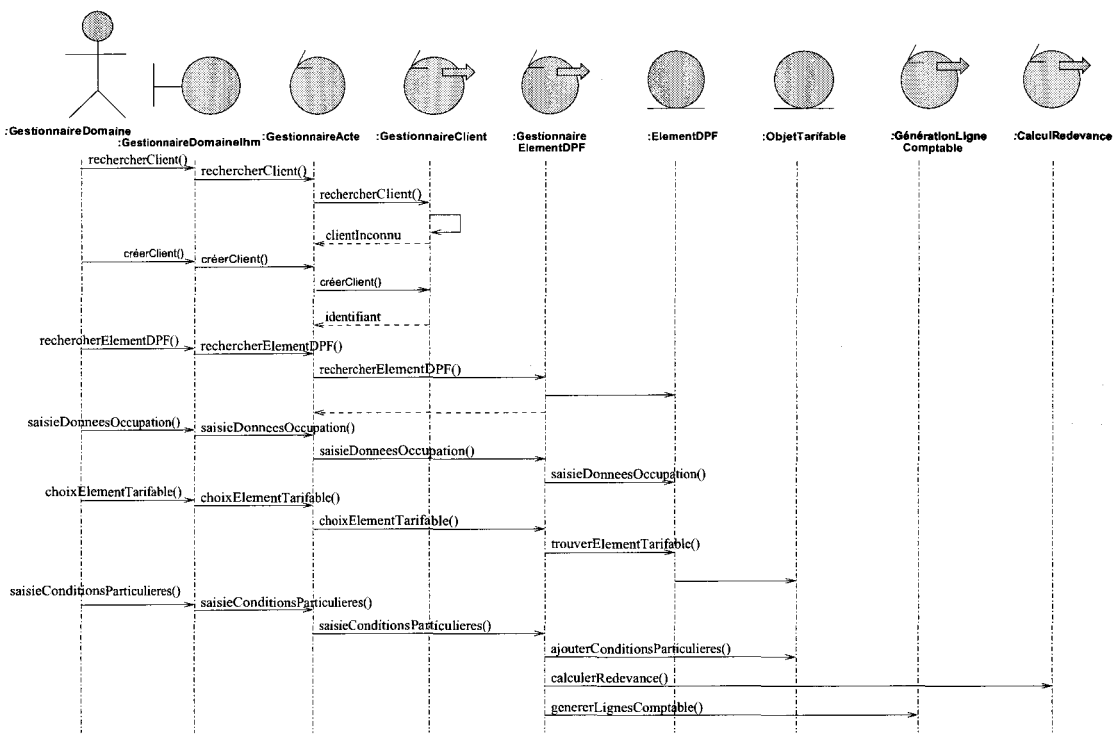


FIG. 6.15 – Vue d'interactions du scénario nominal du cas d'utilisation *Créer, modifier, consulter acte* de la phase de lancement

Vue de conception de composants

Dans un modèle d'analyse, les paquetages et classes sont représentés de manière indépendante d'une technologie. L'utilisation de stéréotypes n'est pas imposée. Aussi, dans le projet initial, ce modèle est souvent mis de côté au profit du modèle de conception représentant les éléments de l'architecture tels qu'ils seront développés et organisés dans la technologie choisie. L'inconvénient est le manque de traçabilité entre les modèles d'analyse et modèles de conception. Et l'évolution d'un besoin nécessite l'identification des éléments de l'architecture nécessaires à la réalisation de ce besoin. Aussi CUP apporte la vue de conception de composants, complément cohérent avec les autres vues (cf figure 6.16).

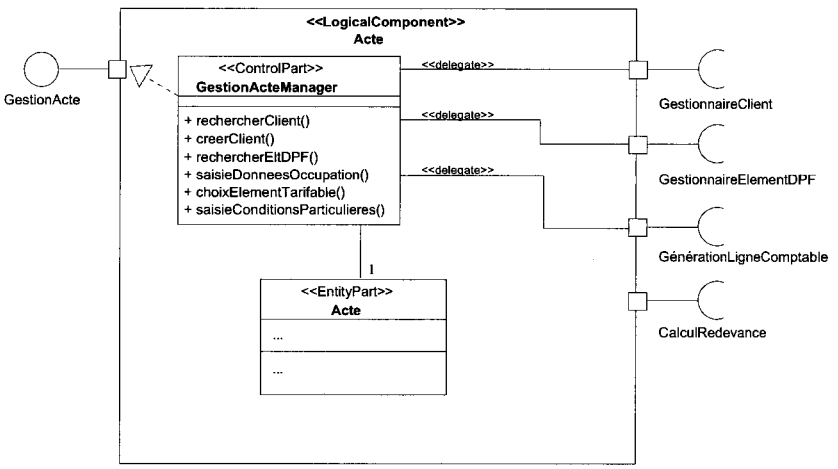


FIG. 6.16 – Extrait de la vue de conception du composant *Acte*

La figure 6.16 représente les interfaces du composant Acte, ainsi que sa structure interne issue de la seule vue de conception. CUP impose la différenciation entre les parties entités et les parties de contrôle. Ainsi, cela limite les choix de conception dans la plupart des architectures distribuées existantes qui font aussi cette différenciation. Cette vue est intéressante pour représenter les objets métiers identifiés, sous forme de EntityPart. En plus, contrairement au diagramme de classes, la vue de conception de composants met en évidence les interfaces fournies et requises des composants et participe ainsi à une meilleure définition du composant.

6.5 Synthèse de la validation expérimentale

Dans la présentation de notre proposition de démarche à base de composants, dénommée CUP, nous avons vu que, si la cohérence entre les vues était maintenue, la démarche n'imposait pas une application séquentielle de ses activités. L'aspect itératif de la démarche est illustré dans l'étude de cas du composant Adresse. En effet, les premiers livrables de cette étude sont travaillés dans la vue de conception de composant, car nous disposions des informations nécessaires pour cette vue. La normalisation CUP du composant a consisté à partir de la vue de conception de composant, à compléter les autres vues. Ainsi, le composant Adresse a été normalisé pour être réutilisable dans des développements à base de composants d'autres systèmes.

L'étude de cas Gestion Administrative du Personnel illustre l'application de notre démarche en partant de rien. Une analyse, qui peut être qualifiée de classique, montre comment concevoir un dossier d'analyse riche modélisé par un PIM. Ce PIM est suffisamment complet pour être projeté sur une plate-forme technologique par le biais d'un PSM. La conception de deux composants issus de l'analyse d'un unique système, montre l'avantage de l'approche en ce qui concerne l'identification et la réutilisation de composants dans des contextes différents.

La troisième étude de cas, incomplète car le système adressé est trop large pour figurer intégralement dans ce mémoire, illustre l'application de notre démarche aux systèmes à large échelle. On constate, grâce à cette étude, que plus le système est large, plus l'application d'une démarche à base de composants telle que CUP, s'avère utile. La partition du système de VNF, a permis de répartir les charges du projet sur plusieurs équipes, tout en bénéficiant de la cohérence maintenue par le modèle. Les frontières des différents composants sont bien marquées par la notion de composant logique. Mais, ce qu'apporte surtout cette notion, c'est l'étude des dépendances fonctionnelles entre les différentes parties du système. C'est une qualité importante de notre approche.

Ces exemples d'application de l'approche CUP à des cas concrets ont été réalisés à Norsys, dans des études démonstratrices, dédiées aux équipes d'analystes et architectes. L'apport de l'approche a été validé dans l'application parallèle à de véritables projets de l'industrie du logiciel. L'avantage de ce processus ouvert est qu'il peut être exploité selon diverses préoccupations et profiter à différentes parties prenantes de projet, selon les vues considérées. L'application du processus CUP en production logicielle se fera certainement par morceaux. Ce qu'autorise la structure de la démarche. Par l'application de diverses parties du processus, la qualité des projets devrait être améliorée.

Il est important de souligner que les descriptions des différentes études de cas de ce chapitre ne permettent pas de mettre en valeur toutes les qualités de l'atelier CUPTool et notamment une vérification automatique de la cohérence entre les vues (implémentation des règles OCL détaillées dans le chapitre 5).

Au sein de la société NORSYS, ces études ont mis en valeur une nécessaire évolution de la répartition des rôles pour chaque acteur du développement logiciel. Ainsi, l'approche conduit l'ingénieur à modéliser la totalité du système d'information. Réaliser la totalité des modèles peut s'avérer long et fastidieux, mais les retours sur la qualité de définition de composants, la maintenance des logiciels produits et leur réutilisabilité montrent l'intérêt d'appliquer l'ingénierie des modèles. De nombreux problèmes comme l'analyse d'impact de l'évolution des besoins, ou la remontée de la traçabilité lors

de la découverte d'une anomalie sont résolus beaucoup plus rapidement grâce aux livrables produits en appliquant la démarche CUP.

Chapitre 7

Conclusion et perspectives

7.1 Bilan du travail de recherche

Dans ce mémoire, nous avons introduit l'intérêt d'adopter une démarche à base de composants dirigée par les modèles. Pour cela, nous avons examiné ce qu'implique l'adoption d'une démarche d'ingénierie dans un projet de développement d'applications dans le chapitre 2. Puis nous avons fait l'état de l'art des démarches à base de composants et des représentations, en comparant l'adaptation d'un formalisme standard tel qu'UML 1.4 aux formalismes dédiés, en indiquant que leur utilisation dans les deux cas ne fournit pas de cadre structurant pour l'identification et la réutilisation de composants (cf figure 7.1). Enfin, nous avons validé l'importance d'adopter un formalisme multi-vues permettant aux différents acteurs d'un projet d'avoir leurs propres points de vue. Cependant, nous avons vu que cela nécessite une cohérence entre les différentes vues.

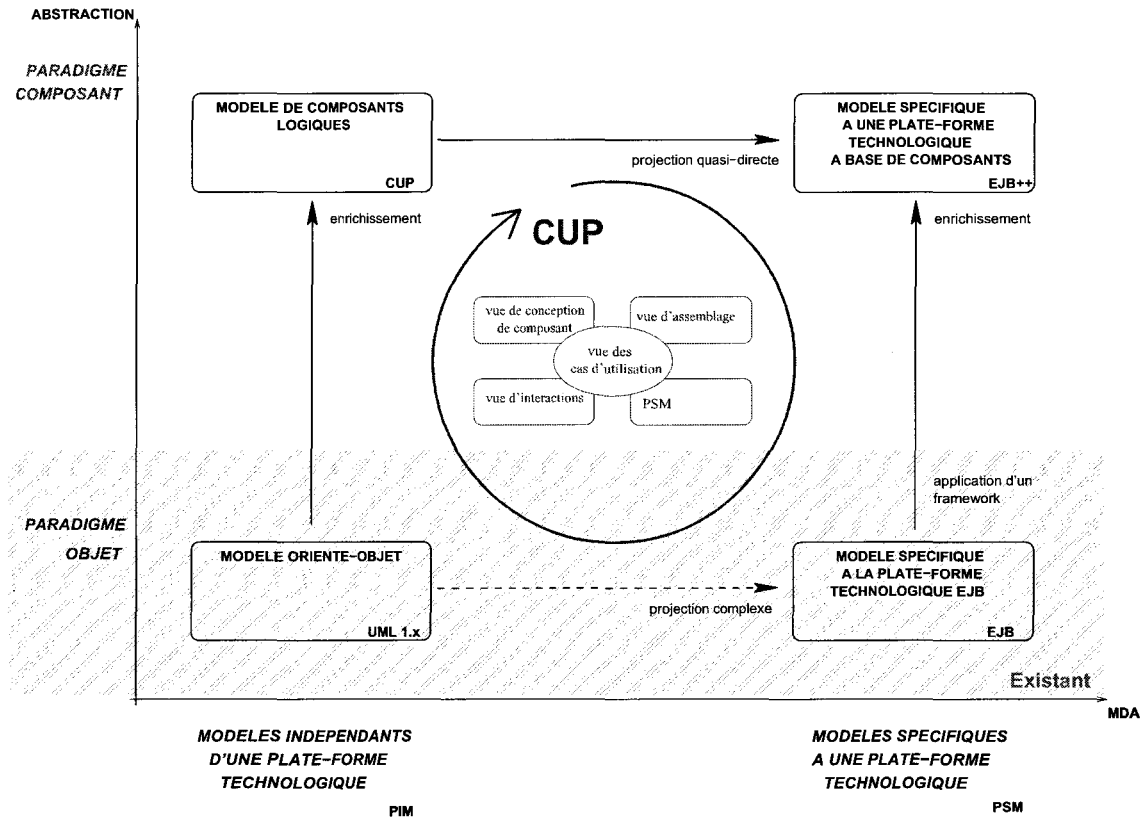


FIG. 7.1 – Schéma de l'approche à base de composants dirigée par les modèles

Le chapitre 3 a présenté la génèse de la notion de composant logique, ou composant de modélisation, que nous proposons dans la thèse comme une meilleure unité de structuration d'une démarche itérative et de réutilisation (cf figure 7.1). Un processus orienté objet, ainsi qu'un formalisme tel qu'UML 1.4 décomposent le système trop finement. La projection d'un modèle orienté objet tel que UML 1.4 vers une plate-forme spécifique à base de composants telle que EJB est complexe. Il est ainsi difficile de remonter en abstraction pour identifier un composant dans les livrables déjà fournis d'un projet, ou dans le cas où l'on souhaite réutiliser un composant sur l'étagère. Les cas d'utilisation permettent d'exprimer ce qu'attendent les utilisateurs du système en terme de fonctionnalités offertes.

Une démarche basée sur les cas d'utilisation garantit que le système développé répond aux besoins de ses utilisateurs finaux. Le cas d'utilisation est réalisé logiquement par des collaborations d'éléments de l'architecture. La notion de collaboration UML, n'est pas utilisée explicitement dans les démarches d'ingénierie classiques. Nous proposons d'exploiter cette notion par le biais des diagrammes d'interactions, et de s'en servir pour identifier les comportements réutilisables significatifs du système. Un composant logique est composé de cas d'utilisation. Les cas d'utilisation sont réalisés par un ensemble de collaborations. En regroupant les collaborations connexes, on définit les frontières du composant logique. L'ensemble des éléments architecturaux, impliqués dans ces collaborations sont ainsi "enfermés" dans le composant logique. Le recoupement entre les collaborations est réalisé par le mécanisme de composition de framework de Catalysis.

Le chapitre 4 développe notre proposition de démarche, **Component Unified Process**, présentée dans [46]. Le **Component Unified Process** est une démarche de conception d'applications à base de composants logiques. La démarche et le formalisme utilisés enrichissent respectivement le Processus Unifié et le langage de modélisation UML. Le modèle de composants logiques est formalisé par un méta-modèle qui est structuré selon quatre vues.

- La **vue de cas d'utilisation** permet de représenter les cas d'utilisation d'un composant, c'est-à-dire, les besoins des utilisateurs, et les services rendus par ce composant.
- La **vue d'interactions** permet de réaliser les cas d'utilisation en montrant les comportements des collaborations des membres du composant.
- La **vue de conception de composants** donne une représentation statique de la spécification du composant en terme d'interfaces et de la représentation interne en terme de parties.
- La **vue d'assemblage** permet de représenter la connexion des composants entre eux pour construire un système ou réutiliser des composants existants.

Le modèle proposé garantit la cohérence globale du système entre les différentes vues, ainsi que des critères de qualité logicielle tels que la lisibilité, la réutilisabilité, la traçabilité, l'évolution et le courtage de composants. Le modèle ainsi que la démarche ont été outillés afin de fournir un environnement de développement d'applications à base de composants autour de la plate-forme de développement industrielle Eclipse.

Le chapitre 5 souligne l'indépendance du modèle CUP vis-à-vis des plates-formes technologiques et l'illustre avec une projection. Il présente l'outil de projection et un prototype de projection du modèle de composant logique, *Platform Independent Model* dans la terminologie *Model Driven Architecture*, vers un modèle *Platform Specific Model* EJB. Présenté dans [47], l'outil et l'expérimentation valident la qualité du modèle de composants logiques ainsi que la démarche. Le framework défini dans [11] montre la nécessité de définir un modèle intermédiaire qui donne une abstraction à base de composants d'une plate-forme orientée objet *Enterprise Java Beans* en l'enrichissant. Ce modèle a été implémenté sous la forme d'un framework EJB++ au-dessus de la plate-forme EJB [12] et ajoute aux spécifications EJB des concepts orientés composants empruntés au modèle abstrait de composants Corba Component Model. La thèse offre ainsi un cadre complet de conception de systèmes à base de composants. Le travail de thèse s'intègre dans l'approche standardisée par l'OMG, **Model Driven Architecture**, qui préconise de modéliser le système de manière indépendante de la plate-forme technologique choisie, et de décliner par transformations de modèles, le modèle PIM vers différents modèles PSM spécifiques à une technologie.

Le chapitre 6 présente la validation de notre travail dans le développement "sur le terrain" de systèmes d'information d'entreprise. La première étude de cas montre comment tirer parti du formalisme lorsque l'on réutilise un composant d'étagère. En le normalisant dans le modèle de composant logique, c'est-à-dire, en développant les différentes vues de ce composant, le procédé garantit une meilleure exploitation de ce composant dans une approche à base de composants. La deuxième étude de cas montre comment appliquer le processus unifié à base de composants en partant du cahier des charges. Enfin, la troisième étude de cas valide l'approche proposée dans cette thèse dans la refonte d'un système d'information de dimension large. Ces études de cas ont été abordées au sein de la société Norsys.

L'apport de la thèse est dans l'offre d'une démarche, de formalismes et d'outils pour la conception d'applications avec une approche à base de composants, dirigée par les modèles, et dans la définition et la réalisation d'une chaîne de production de systèmes à base de composants.

7.2 Perspectives

7.2.1 Perspectives à court terme

Les phases d'expression des scénarii des composants

Le processus CUP repose sur l'expression des besoins de manière textuelle dans les scénarii. Les cas d'utilisation résultent d'une étape d'analyse de ces scénarii qui consiste à les regrouper logiquement selon le but à atteindre. Le langage humain est soumis à interprétation et à des incompréhensions, une mauvaise lecture peut conduire un système à sa perte. Cockburn [15] travaille sur les phases amonts de notre proposition en structurant les cas d'utilisation par la notion de but. Les travaux de Colette Rolland [49] permettent de faire découvrir les besoins en se servant de scénarii écrits en langage naturel et de les formaliser grâce à une syntaxe ET/OU. L'intégration de ces travaux sous la forme d'une vue supplémentaire apporterait d'avantage de fiabilité à la démarche CUP, et offrirait davantage de critères d'identification de composants COTS[1].

La prise en compte des processus métiers

La prise en compte des possibilités de réutilisation encore plus tôt, permettrait d'améliorer la rapidité de développement. Grâce à l'utilisation d'outils conformes à l'approche MDA, la génération d'applications pourrait se faire à partir de modèles UML d'objets métiers. Eric Lefebvre [19] travaille sur le concept de composants d'affaires, faciles à réutiliser, c'est-à-dire, à assembler sur n'importe quelle plate-forme technologique. C'est un champ d'application de la démarche et de l'outillage CUP qui pourraient être intéressants pour cette approche qui vise à détecter et réutiliser encore plus tôt les composants métiers, notamment dans l'expression des fonctionnalités et dépendances entre composants grâce aux cas d'utilisation et dans le processus qui en découle.

Les projections PIM vers PSM

CUP 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 CUP peut être projeté vers

[1] Commercial Off The Shelves

diverses plates-formes technologiques. Un travail en cours de réalisation permettra de valider la projection vers la plate-forme OpenCCM [71]. Nous l'avons vu dans le document, le modèle EJB++ permettant de rendre plus directe la correspondance entre les concepts du modèle CUP et les concepts de la plate-forme technologique EJB, est proche du modèle abstrait de composants CCM. Ceci implique que la projection vers CCM doit être assez proche de celle présentée dans ce document. Un prototype de projection vers la plate-forme académique Fractal [6] permettrait une autre validation de projection. On constate l'intérêt de l'approche CUP permettant d'adopter des mécanismes MDA, et à partir d'un modèle neutre vis-à-vis d'une technologie de générer l'implémentation de systèmes sur diverses plates-formes technologiques.

L'amélioration du ciblage PIM vers PSM

Un autre objectif à court terme de notre travail est d'améliorer le ciblage de notre modèle PIM vers les PSM en ajoutant des informations non fonctionnelles. Notre modèle se focalise sur les caractéristiques fonctionnelles du système. La projection sur les plates-formes technologiques fait des choix quant aux aspects non fonctionnels de l'implémentation. Les développeurs peuvent alors ensuite améliorer les paramètres non fonctionnels des applications. La vue de conception de composants serait alors la plus impactée par cette évolution, par l'ajout de caractéristiques venant paramétrer le modèle afin d'obtenir un modèle PSM plus proche de ce qu'on attend du système, notamment des exigences non fonctionnelles. On peut citer par exemple l'exigence transactionnelle [50], l'exigence de déploiement [4]. Le modèle de composants du projet RNTL ACCORD [30] contient la notion de contrat, une annotation du modèle pour mieux valider un assemblage permettant de faire de l'Aspect Oriented Modeling.

7.2.2 Perspectives à plus long terme

Adaptation à d'autres méthodologies

Dans le mémoire nous proposons un formalisme, une démarche et l'outillage permettant d'exploiter le formalisme et de mettre en oeuvre la démarche pour la conception d'applications. Notre notation, suffisamment générale, permet notamment de mieux caractériser un composant. Aussi, nous souhaitons pouvoir l'étendre pour l'**adapter à d'autres méthodologies**, comme CROME, Catalysis et UML Component. Le métamodèle que nous proposons ne bénéficie pas, en l'état, de mécanisme d'extensibilité. Pour cela, nous avons étudié les mécanismes d'extensibilité de UML 1.4, de UML 2.0 afin de munir notre méta-modèle de composant logique d'un tel attribut, et de permettre à l'approche CUP de s'adapter à d'autres approches. Nous privilégions la notion de profil pour étendre notre notation et ainsi proposer une architecture plus adaptée à d'autres démarches tout en garantissant la compatibilité de la notation et de l'outillage.

L'adaptation de la notation à ces démarches permettra de bénéficier des avantages de meilleure définition de la notion de composant logique et ainsi, améliorerait l'identification et la réutilisation de composants. Dans le chapitre 5, nous avons souligné les qualités de structuration fonctionnelle du framework EJB++. Une étude au niveau modèle sur l'adaptation de CROME [31] et sur la poursuite de CROME menée actuellement dans l'équipe permettra de valider les apports de CUP dans ces démarches respectives.

La préoccupation gestion de projet

Une prise en compte de la préoccupation **Gestion de Projet** dans le méta-modèle, permettrait de modéliser différentes démarches ou processus d'ingénierie selon deux axes : le temps et les tâches. Dans la figure 7.2 est présentée une adaptation du métamodèle de processus logiciel *Software Process Engineering Metamodel* [37]. Le déroulement d'un projet dans le temps est décomposé en phases réalisées séquentiellement. Chaque phase est décomposée plus finement en itération. Dans l'axe des tâches, chacune des activités est jalonnée par la livraison de produits qui correspondent à un état particulier d'un ou plusieurs éléments du modèle. Des contraintes OCL sont appliquées sur les produits et vérifiées plus ou moins selon la précision et l'avancée du projet dans le temps. Cette extension permet de modéliser n'importe quelle méthodologie et de capitaliser sur un processus d'ingénierie. L'outillage guide alors les intervenants du projet. Les différentes activités de raffinement du processus MDA seront ainsi introduites dans le métamodèle et offriront un cadre d'analyse et développement d'un système. La finalité est la production d'un cadre de conception générique avec un paradigme composant, adaptable à toute méthodologie.

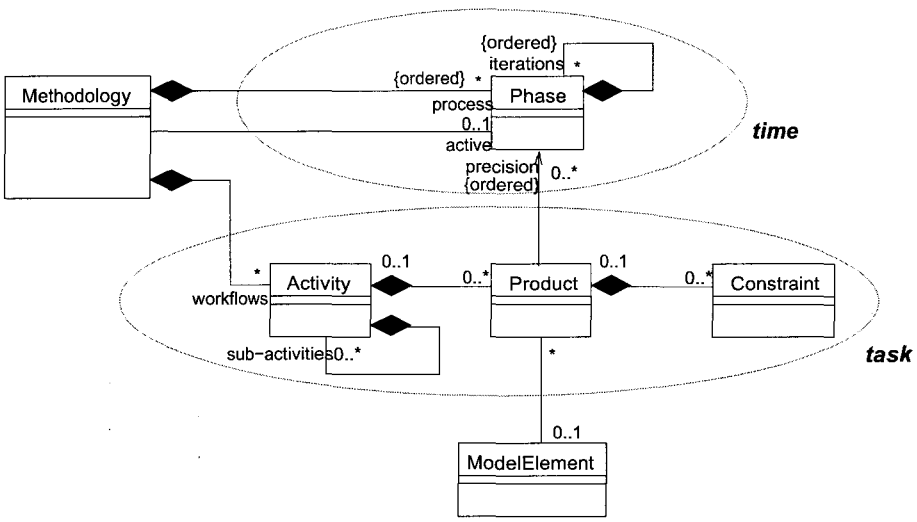


FIG. 7.2 – Extensibilité de méta-modèle CUP pour intégrer la méthodologie

Ces différentes perspectives adoptent la même philosophie, qui est celle appliquée dans cette thèse : grâce à une définition rigoureuse par méta-modélisation, il est possible d'obtenir une véritable **ingénierie des modèles** ; et offrent de nombreux avantages : la génération de cadre de conception, la génération d'outils de contrôle et de la cohérence, des modèles (guides), outils de transformations de modèles, ...

La réutilisation de COTS

La contribution de la thèse est de fournir un moyen de mieux concevoir les composants logiques grâce à un formalisme, une démarche et un outillage offrant de nombreux avantages. Nous considérons que ce travail constitue une base solide vers notre Graal :

Des composants de modèles sur l'étagère

Cet objectif pourra se réaliser par l'avènement de plusieurs domaines. Au niveau du domaine MDA, les travaux sur les transformations de modèles PIM vers PSM feront que des composants de modèles sur l'étagère seront beaucoup plus exploitables que des composants binaires liés à une seule technologie. Les composants de modèles seront ainsi projetables sur plusieurs technologies. Au niveau de la modélisation, il convient de fournir aux composants de modèles une infrastructure qui permet de les localiser, de les décrires, de vérifier leur adéquation pour un éventuel assemblage.
En effet, la meilleure définition des composants logiques devrait permettre de mieux les partager, de mieux les publier. Aussi, la réutilisation de COTS est une problématique scientifique qui dépend beaucoup de la notation et de la démarche utilisées pour modéliser les composants, et ceci est particulièrement vrai pour les composants métiers. Améliorer la réutilisation de composants préfabriqués demande de pouvoir les catégoriser. Habituellement, les composants métiers sont catégorisés selon le domaine métier. Ensuite, l'interface d'un composant permet de l'identifier à partir des opérations qu'il propose. Les cas d'utilisation, les rôles et donc les collaborations, évidemment les interfaces et constituants du composant, puis ses possibilités d'assemblage différentes lorsqu'il est composé de plusieurs composants, sont autant de critères permettant d'améliorer le courtage. Ensuite, le composant choisi doit pouvoir être intégré en étant associé à d'autres composants et ceci à moindre coût et facilement. Là aussi, de nombreuses problématiques sont sous-jacentes à l'intégration de composant, comme la compatibilité des interfaces, la résolution des conflits de nommage, et le test d'intégration. Une application conçue par assemblage de composants ne remplit pas forcément l'ensemble des fonctionnalités voulues. Il faut pouvoir garantir par contrat la couverture fonctionnelle. Enfin, lors de la conception d'une application, il est possible de produire de nouveaux composants aux qualités suffisamment remarquables pour être publiés, voire même des composants composites qui assemblent des composants COTS.
L'ingénierie classique des systèmes distribués n'est pas distincte de l'ingénierie à base de composants. Elle l'intègre au fur et à mesure que la recherche avance dans ce domaine et que les résultats d'évaluations dans un contexte industriel s'avèrent probants.

Bibliographie

[1] E.P. ANDERSEN and T. REENSKAUG. *System Design by Composing Structures of Interacting Objects*. In *ECOOP'92*, pages 133–152, Netherlands, June 1992. Springer-Verlag.

[2] F. BARBIER, C. CAUVET, M. OUSSALAH, D. RIEU, S. BENNASRI, and C. SOUVEYET. *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.

[3] G. BOOCH, I. JACOBSON, and J. RUMBAUGH. *The Unified Modeling Language User Guide*. Addison-Wesley, 1998.

[4] F. BRICLET, C. CONTRERAS, and P. MERLE. *Une infrastructure à composants pour le déploiement d'applications à base de composants CORBA*. In *1ère conférence sur le Déploiement et la (Re)Configuration de Logiciels - DECOR 2004*, Grenoble - France, October 2004. Hermès Sciences.

[5] L. BROWNSWORD, T. OBERNDORF, and C.A. SLEDGE. *Developing New Processes for COTS-Based Systems*. IEEE Software 17, 4, 2000.

[6] E. BRUNETON, T. COUPAYE, and J.B. STEFANI. *The Fractal Component Model, version 2.0-3*, February 2004. Online documentation http ://fractal.objectweb.org/specification/.

[7] E. CARIOU and A. BEUGNARD. *The Specification of UML Collaborations as Interactions Components*. In *Fifth International Conference on the Unified Modeling Language, UML'2002*, 2002.

[8] E. CARIOU, A. BEUGNARD, and J.M. JÉZÉQUEL. *An Architecture and a Process for Implementing Distributed Collaborations*. In *EDOC'2002*, 2002.

[9] O. CARON, B. CARRÉ, and L. DEBRAUWER. *Contextualization of OODB Schemas in CROME*. In *Proceedings of the DEXA Conference*, 2000.

[10] O. CARON, B. CARRÉ, and L. DEBRAUWER. *CromeJava : une implémentation du modèle CROME de conception par contextes pour les bases de données à objets en Java*. In *Proceedings of LMO'00*, January 2000.

[11] O. CARON, E. RENAUX, and J.M. GEIB. *Vers de Véritables Composants EJB Réutilisables*. In *OCM-SI'02*, Nantes, June 2002.

[12] O. CARON, E. RENAUX, and J.M. GEIB. *Vers de Véritables Composant EJB Réutilisables. Ingénierie des composants dans les systèmes d'information, Numéro spécial de la revue l'Objet*, 10(1), 2004.

[13] R.G.G. CATTELL and al. *The Object Database Standard : ODMG 2.0.* International Thomson Publishing, 1997.

[14] S. CLARKE. *Extending standard UML with model composition semantics. Science of Computer Programming*, 4(1) :71–100, July 2002.

[15] A. COCKBURN. *Structuring Use Cases with Goals.* In *technical report, Human Technology, HaT.TR.95.1, 84121*, Salt Lake City, UTAH, 1995. *http ://members.aol.com/acocburn/papers/usecases.htm.*

[16] CUPTOOL. *CUPTool Web Page, http ://www.lifl.fr/~renaux/recherche/cup.html*, 2003.

[17] L. DEBRAUWER. *Des vues aux contextes pour la structuration fonctionnelle de bases de données à objets en CROME.* PhD thesis, Laboratoire d'Informatique Fondamentale de Lille I, Lille, décembre 1998.

[18] L. DEBRAUWER, B. CARRÉ, and G. VANWORMHOUDT. *Un cadre de conception par contextes fonctionnels de systèmes d'information à objets.* In *XVème Congrès INFORSID*, Toulouse, juin 1997.

[19] J. DE LUCA E. LEFEBVRE, P. COAD. *Java Modeling in Color with UML.* Java Modeling in Color with UML, 1999.

[20] M. FOWLER. *UML Distilled, a Brief Guide to the Standard Object Modeling Language* . Addison-Wesley, Third Edition edition, 2004.

[21] J.M. GEIB, C. GRANSART, and P. MERLE. *CORBA : des Concepts à la Pratique.* Inter-Editions, 1997.

[22] IBM. *Eclipse home page, http ://www.eclipse.org*, 1999.

[23] I. JACOBSON. *Use Cases and Aspects - Working Seamlessly Together. Journal of Object Technology*, 2(4) :7–28, July-August 2003.

[24] E. LEFEBVRE. *Laboratoire en Architecture de Systèmes Informatiques (LASI) http ://www.lasi.etsmtl.ca/*, 2004.

[25] R. LEMESLE. *Techniques de modélisation et de méta-modélisation.* PhD thesis, Université de Nantes, Nantes, 2000.

[26] X. LE PALLEC and G. BOURGUIN. *RAM3 : un outil dynamique pour le Meta-Object Facility.* In *Proceedings of LMO'01*, janvier 2001.

[27] H. MILI, H. MCHEICK, J. DARGHAM, and S. DELLOUL. *Distribution d'objets avec vues.* In *Proceedings of LMO'01*, January 2001.

[28] MOF. *Essential MOF (EMOF) O.M.G. MOF2.0 Core Submission, Chap. 7, http ://www.omg.org/*, 2002.

[29] MOF. *O.M.G. MOF2.0 Core Submission, http ://www.omg.org/*, 2002.

[30] A. MULLER. *La démarche MDA*, 2002.

[31] A. MULLER, O. CARON, B. CARRÉ, and G. VANWORMHOUDT. *Réutilisation d'aspects fonctionnels : des vues aux composants.* In *Langages, Modèles et Objets - LMO 2003*, pages 241–255, Vannes - France, January 2003. Hermès Sciences.

[32] L. DEBRAUWER O. CARON, B. CARRÉ. *Contextualization of OODB Schemas in CROME. DEXA 2000, 11th International Conference, London*, Septembre 2000.

[33] OMG. *OMG Model-Driven Architecture Home Page, http ://www.omg.org/mda.*

[34] OMG. *CORBA 3.0 New Component Chapter*, 11 2001. OMG ptc/2001-11-03.

[35] OMG. *CORBA 3.0 New Components Chapters*. Object Management Group, Novembre 2001. OMG ptc/2001-11-03.

[36] OMG. *UML1.4 - chapter 6 -Object Constraint Language*. Object Management Group, Septembre 2001. OMG formal/01-09-77.

[37] OMG. *Software Process Metamodel Specification http ://www.omg.org/technology/documents/formal/spem.htm*, 2002.

[38] OMG. *Spécification du profil UML pour CORBA, version 1.0 http ://www.omg.org/technology/documents/formal/ profile_corba.htm*, 2002.

[39] OMG. *O.M.G. Home Page, http ://www.omg.org*, 2003.

[40] OMG. *Catalog of OMG Modeling and Metadata Specifications, http ://www.omg.org/technology/documents/modeling_spec_catalog.htm*, 2004.

[41] OMG. *Spécification du profil UML pour EDOC http ://www.omg.org/technology/documents/formal/edoc.htm*, 2004.

[42] R. PAWLAK and H. YOUNESSI. *On Getting Use Cases and Aspects to Work Together. Journal of Object Technology*, 3(1) :15–26, January-February 2004.

[43] J.D. POOLE. *Model-Driven Architecture : Vision, Standards And Emerging Technologies. ECOOP 2001, Workshop on Metamodeling and Adaptive Object Models*, April 2001.

[44] R. PAWLAK. *Java Aspect Component home page, http ://jac.objectweb.org*, 1999.

[45] T. REENSKAUG, P. WOLD, and O.A. LEHNE. *Working woth Objects. The OOram Software Engineering Method*. Manning Publications Co., 1995.

[46] E. RENAUX, O. CARON, and J.M. GEIB. *The Component Unified Process Project*. In *SEA'03*, pages 669–674, Los Angeles USA, November 2003. IASTED, ACTA Press.

[47] E. RENAUX, O. CARON, and J.M. GEIB. *Chaîne de production de systèmes à base de composants logiques*. In *Proceedings of LMO'04*, pages 147–160, Lille, France, March 2004.

[48] D. RIEU, J.P. GIRAUDIN, and A. CONTE. *Pattern-Base Environments for Information Systems Development*. In *International Conference, The Sciences of Design*, Lyon - France, Mars 2002.

[49] C. ROLLAND, C. SOUVEYET, and C. BEN ACHOUR. *Guiding Goal Modeling Using Scenarios*. In *IEEE Transactions on Software Engineering*, volume 24, pages 1055–1071, 1998.

[50] R. ROUVOY and P. MERLE. *GoTM : Vers un canevas transactionnel à base de composants*. In *Langages Modèles et Objets - Journées Composants - LMO 2004-JC 2004*, volume 10 of *L'objet*, pages 131–146, Lille - France, March 2004. Hermès Sciences.

[51] COLLETTI TARDIEU, ROCHFELD. *La méthode MERISE, principes et outils (Tome 1 et 2)*. Les éditions d'organisation, 1986.

[52] D. BARDOU. *Roles, Subjects and Aspects : How do they relate ?*, July 1998. Position paper at the Aspect Oriented Programming Workshop, ECOOP'98, Brussels, Belgium. Extended abstract published in ECOOP'98 Workshop Reader, Serge Demeyer and Jan Bosch, editors, Lecture Notes in Computer Science (LNCS), vol. 1543, Springer, 418–419, December 1998.

[53] M. BARTORELLO, H. MAGUIN, A. OCELLO, M. BLAY-FORNARINO, and A-M. DERY. *Intégration de services au sein d'un serveur EJB*. In *Proceedings of LMO'02*, 2002.

[54] K. BECK. *Extreme Programming Explained : Embrace Change.* Addison-Wesley, October 1999.

[55] J. CHEESMAN and J. DANIELS. *UML Components - A Simple Process for Specifying Component-Based Software.* Addison-Wesley, 2001.

[56] D. FRANCIS D'SOUZA and A.C. WILLS. *Objects, Components, and Frameworks with UML - The Catalysis Approach.* Addison-Wesley, 1998.

[57] E. GAMMA, R. HEBN, R. JOHNSON, and J. VLI SSIDES. *Design Patterns, Elements of Reusable Object-Orien ted Software.* Addison-Wesley, 1994.

[58] A. GERBER and K. RAYMOND. *MOF to EMF : There and Back Again.* In *Proceedings of Eclipse Technology Exchange Workshop, OOPSLA 2003, Anaheim. USA*, pages 60–70, October 2003.

[59] P. HERZUM and O. SIMS. *Business Component Factory - A Comprehensive Overview of Component-Based Development for the Enterprise.* Wiley Computer Pub., 2000.

[60] I. JACOBSON. *Object Oriented Development in an Industrial Environment.* In *Proceedings of OOPSLA'87.* ACM, 1987.

[61] I. JACOBSON. *Basic Use Case Modeling.* In *ROAD 1*, 1994.

[62] I. JACOBSON, G. BOOCH, and J. RUMBAUGH. *The Unified Software Development Process.* Addison-Wesley, 1997.

[63] I. JACOBSON, G. BOOCH, and J. RUMBAUGH. *The Unified Software Development Process.* Addison-Wesley, 1999.

[64] G. KICZALES AND AL. *Aspect-Oriented Programming.* In *Proceedings of the 11th European Conference on Object-Oriented Programming (ECOOP'97)*, pages 220–242, 1997.

[65] P. KRUCHTEN. *Architectural BluePrints – The "4+1" view Model of Software Architecture.* In *IEEE Software 12*, pages 42–50, November 1995.

[66] K. LIEBERHERR, D. LORENZ, and M. MEZINI. *Programming with aspectual components.* Technical Report NU-CCS-99-01, Northeastern University, Boston, MA 02115, 1999.

[67] C.V. LOPES and G. KICZALES. *Recent Developments in AspectJ.* In Springer-Verlag LNCS 1543, editor, *Proceedings of ECOOP'98 Workshop Reader*, 1998.

[68] C. LÜER and D. S. ROSENBLUM. *UML Component Diagrams and SoftwareArchitecture : Experiences from the Wren Project.* In *1st ICSE Workshop on Describing Software Architecture with UML*, pages 79–82, Toronto, 2001.

[69] R. MARVIE and M.-C. PELLEGRINI. *Modèles de composants, un état de l'art. Numéro spécial de L'Objet*, 2001.

[70] R. MARVIE, P. MERLE, J.-M. GEIB, and C. GRANSART. *Des objets aux composants CORBA, première étude et expérimentation avec CorbaScript.* In *Proceedings of the 12th International Conference on Software and Systems Engineering and their Applications (ICSSEA'99)*, Paris, France, December 2000.

[71] P. MERLE, C. CONTRERAS, M. BORN, J. REZNIK, A. HOFFMAN, T. RITTER, A. RENNOCH, B. NEUBAUER, S. EFREMIDIS, M. VADET, U. LANG, and R. SCHREINER. *State of the Art : Component Models for Distributed Systems.* Technical Report D1.2, Requirement Analysis of Telecom CORBA Components IST Programme. Project IST-2001-34445. 1 April 2002 to 31 March 2004, March 2003.

[72] B. MEYER. *Object-Oriented Software Construction, 2nd Edition.* Prentice Hall, 1997.

[73] M. MEZINI and K. LIEBERHERR. *Adaptive plug-and-play components for evolutionary software development.* In *Proceedings of OOPSLA'98*, pages 18–22, October 1998.

[74] R. MONSON-HAEFEL. *Enterprise Java Beans.* O'Reilly, 2001.

[75] H. OSSHER and W. HARRISON. *Subject-Oriented Programming (a Critique of Pure Objects).* In *Proceedings of the 8th Conference on Object-Oriented Programming Systems (OOPSLA'93)*, pages 411–428, 1993.

[76] H. OSSHER, M. KAPLAN, W. HARRISON, A. KATZ, and V. KRUSKAL. *Subject-Oriented Composition Rules.* In *Proceedings of the 10th Conference on Object-Oriented Programming Systems (OOPSLA'95)*, pages 235–250, 1995.

[77] D. PARNAS. *On The Criteria to be used in decomposing systems in modules.* In *Communication on the ACM*, volume 15, pages 1053–1058, 1972.

[78] R. PAWLAK, L. DUCHIEN, G. FLORIN, L. MARTELLI, and L. SEINTURIER. *Distributed separation of concerns with aspect components.* In *Proceedings of 33rd International Conference TOOLS 33*, Technology of Object-Oriented Languages, pages 276–287, June 2000.

[79] M. PELTIER. *Transformation entre un profil UML et un métamodèle MOF. Application du langage MTrans.* In *Proceedings of LMO'02*, January 2002.

[80] M. PELTIER, J. BÉZIVIN, and G. GUILLAUME. *MTRANS, A general framework based on XSLT for model transformations.* In *Proceedings of the Worksop on Transformations in UML, WTUML'01*, Genova, Italy, April 2001.

[81] J. PUTMAN. *Architecting with RM-ODP.* Prentice Hall, 2001.

[82] P. ROQUES and F. VALLÉE. *UML en action - 2e édition.* Eyrolles, 2002.

[83] SUN. *Spécification du profil UML pour EJB*
http ://jcp.org/aboutJava/communityprocess/review/
jsr026, 2001.

[84] SUN. *E.J.B. Home Page, http ://www.javasoft.com/ejb*, 2001.

[85] SUN. *Java Beans Home Page, http ://java.sun.com/products/javabeans/beanbuilder*, 2001.

[86] SUN. *Sun's Core J2EE(TM) Patterns : Session Facade pattern*, 2001.

[87] G. SUNYÉ, A. LE GUENNEC, and J.M. JÉZÉQUEL. *Design Pattern Application in UML.* In Springer Verlag Lecture Notes in Computer Science, editor, *ECOOP'2000 Proceedings*, pages 44–62, June 2000.

[88] P. TARR, H. OSSHER, W. HARRISON, and S. SUTTON. *N Degrees of Separation : Multi-Dimensional Separation of Concerns.* In *ICSE 1999 Conference Proceedings*, pages 107–119, 1999.

[89] A.C. WILLS. *Frameworks and component-based development.* In Springer, editor, *OOIS'96*, 1996.

[90] UML. *O.M.G. UML Ressource Page, http ://www.omg.org/*, 1997.

[91] G. VANWORMHOUDT. *CROME : un cadre de programmation par objets structurés en contextes.* PhD thesis, Laboratoire d'Informatique Fondamentale de Lille I, Lille, Septembre 1999.

[92] J. WARMER and A. KLEPPE. *The OCL Constraint Language.* Addison-Wesley, 1999.

Annexe A

Code du framework EJB++

fr.lifl.goal.EJBPlusPlus.Facet

```
package fr.lifl.goal.EJBPlusPlus;

public interface  Facet extends java.rmi.Remote, Receptacle {}
  // each component facet must extend this interface
```

fr.lifl.goal.EJBPlusPlus.Receptacle

```
package fr.lifl.goal.EJBPlusPlus;

public interface  Receptacle extends java.rmi.Remote {}
  // each component receptacle must extend this interface
```

fr.lifl.goal.EJBPlusPlus.ExtendedEJBObject

```
package fr.lifl.goal.EJBPlusPlus;

import java.rmi.RemoteException;
import java.util.Vector;

public interface  ExtendedEJBObject extends javax.ejb.EJBObject {
  // EJB remote interface must extend this interface,
  // instead of javax.ejb.EJBObject
  public void connect(String receptacleName, Facet facet)
    throws RemoteException, AlreadyConnectedException ;
    // to connect this component to facet of another component
  public void disconnect(String receptacleName)
    throws RemoteException, NotConnectedException ;
    // to connect this component to facet of another component
  public Receptacle getReceptacle(String receptacleName)
    throws RemoteException,  UnknownPortException;
    // retrieve a receptacle by its name
  public Facet getFacet(String facetName)
    throws RemoteException, UnknownPortException  ;
    // retrieve a facet by its name
  public void configuration_complete()
    throws RemoteException, InvalidConfigurationException ;
    // check the configuration of component implemented by the
    // developer
  public Vector getFacetNames() throws RemoteException ;
    // obtain all facette names to retrieve them
  public Vector getReceptacleNames() throws RemoteException ;
    // obtain all receptacle names to retriexe them
}
```

fr.lifl.goal.EJBPlusPlus.ExtendedEJBObjectImpl

```
package fr.lifl.goal.EJBPlusPlus;

import java.lang.reflect.InvocationTargetException;
import java.lang.reflect.Method;
import java.rmi.RemoteException;
```

```
import java.util.Hashtable;
import java.util.Vector;

import javax.ejb.EJBObject;

public abstract class ExtendedEJBObjectImpl {
  // must be extend by the bean of the component

  protected Hashtable receptacles;
  protected Vector receptacleNames;

  public ExtendedEJBObjectImpl() {
    receptacles = new Hashtable();
    receptacleNames = new Vector();
  }

  protected abstract void setReceptacleNames();

  public void connect(String receptacleName, Facet facet)
    throws AlreadyConnectedException {
    if (!receptacles.containsKey(receptacleName))
      receptacles.put(receptacleName, facet);
    else
      throw new AlreadyConnectedException();
  }

  public void disconnect(String receptacleName)
    throws NotConnectedException {
    if (!receptacles.containsKey(receptacleName))
      throw new NotConnectedException();
    receptacles.remove(receptacleName);
  }

  public Facet getFacet(String facetName)
    throws RemoteException, UnknownPortException {
    Class interfaces[] = getEJBMetaData().getRemoteInterfaceClass()
                         .getInterfaces();
    for (int i = 0; i < interfaces.length; i++) {
      Class interf = interfaces[i];
      if (interf.getName().compareTo(facetName) ==0) {
        return (Facet) this.getEJBObject();
      }
    }
    throw new UnknownPortException();
  }

  public Receptacle getReceptacle(String receptacleName)
    throws UnknownPortException {
    if (!receptacles.containsKey(receptacleName))
      throw new UnknownPortException();
    return (Facet) receptacles.get(receptacleName);
  }
```

```
public abstract javax.ejb.EJBContext getEJBContext();
  // must be implemented by the developer which indicate
  // session or entity context

private EJBObject getEJBObject()throws RemoteException {
  try {
    javax.ejb.SessionContext ctx =
      (javax.ejb.SessionContext) getEJBContext();
    return ctx.getEJBObject();
  } catch (Exception e) {
    javax.ejb.EntityContext ctx =
      (javax.ejb.EntityContext) getEJBContext();
    return ctx.getEJBObject();
  }
}

private javax.ejb.EJBMetaData getEJBMetaData ()
  throws RemoteException {
  return getEJBObject().getEJBHome().getEJBMetaData();
}

public Vector getFacetNames() throws RemoteException {
  Class interfaces[] = getEJBMetaData().getRemoteInterfaceClass()
                       .getInterfaces();
  Vector facetNames = new Vector();
  for (int i = 0; i < interfaces.length; i++) {
    Class f = interfaces[i];
    if (f.getName().indexOf("Facet") != -1)
      facetNames.addElement(f.getName());
    }
    return facetNames;
}

protected Object invokeMethod(String receptacleName,
                              String methodName,
                              Class[] argTypes,
                              Object[] args)
  throws UnknownPortException, UnknownMethodException, RemoteException {
  Class interfaces[] = getEJBMetaData().getRemoteInterfaceClass()
                       .getInterfaces();
  for (int i = 0; i < interfaces.length; i++) {
    Class f = interfaces[i];
    if (f.getName()==receptacleName) {
      try {
        f.getMethod( methodName,argTypes);
      } catch (NoSuchMethodException e){
        throw new UnknownMethodException();
      }
    }
  }
  Facet f = (Facet)this.getReceptacle(receptacleName);
  Object result=null;
  try {
```

```
      Method m = f.getClass().getMethod(methodName,argTypes);
      result=m.invoke(f,args);
    } catch (Exception e) {
      throw new UnknownMethodException();
    }
    return result;
  }

  public Vector getReceptacleNames() throws RemoteException {
    return receptacleNames;
  }

  public abstract void configuration_complete()
    throws InvalidConfigurationException;
}
```

fr.lifl.goal.EJBPlusPlus.AlreadyConnectedException

```
package fr.lifl.goal.EJBPlusPlus;

public class AlreadyConnectedException extends java.lang.Exception{
  public AlreadyConnectedException(){super();}
  public AlreadyConnectedException(String m){super(m);}
}
```

fr.lifl.goal.EJBPlusPlus.InvalidConfigurationException

```
package fr.lifl.goal.EJBPlusPlus;

public class InvalidConfigurationException extends java.lang.Exception{
  // throws by configuration_complete method
  public InvalidConfigurationException(){super();}
  public InvalidConfigurationException(String m){super(m);}
}
```

fr.lifl.goal.EJBPlusPlus.UnknownMethodException

```
package fr.lifl.goal.EJBPlusPlus;

public class UnknownMethodException extends java.lang.Exception{
  // throws when there is a call to a receptacle
  // and the method doesn't exists
  public UnknownMethodException(){super();}
  public UnknownMethodException(String m){super(m);}
}
```

fr.lifl.goal.EJBPlusPlus.UnknownPortException

```
package fr.lifl.goal.EJBPlusPlus;

public class UnknownPortException extends java.lang.Exception{
  // throws when by getFacet and getReceptacle method when the
```

```
  // argument name doesn't correspond to a port
  // of the component
  public UnknownPortException(){super();}
  public UnknownPortException(String m){super(m);}
}
```

Annexe B

Code d'un composant X

xComponent.XFacet1

```
package xComponent;

import fr.lifl.goal.EJBPlusPlus.Facet;
import fr.lifl.goal.EJBPlusPlus.UnknownPortException;

public interface XFacet1 extends Facet{
  public String getName() throws java.rmi.RemoteException;
  public Integer plus (Integer arg1, Integer arg2)
    throws UnknownPortException, java.rmi.RemoteException;
}
```

xComponent.XFacet2

```
package xComponent;

public interface XFacet2 extends fr.lifl.goal.EJBPlusPlus.Facet {
  public Integer sub(Integer arg1, Integer arg2)
    throws java.rmi.RemoteException;

}
```

xComponent.XReceptacle

```
package xComponent;

public interface XReceptacle extends fr.lifl.goal.EJBPlusPlus.Receptacle {
  public Integer add(Integer arg1, Integer arg2)
    throws java.rmi.RemoteException;
}
```

xComponent.XRemote

```
package xComponent;

public interface XRemote extends fr.lifl.goal.EJBPlusPlus.ExtendedEJBObject,
                                 XFacet1, XFacet2 {}
```

xComponent.XRemoteHome

```
package xComponent;

import javax.ejb.CreateException;
import javax.ejb.EJBHome;

public interface XRemoteHome extends EJBHome {
  public XRemote create()
    throws CreateException,java.rmi.RemoteException;
}
```

xComponent.XBean

```
package xComponent;

import java.rmi.RemoteException;
import javax.ejb.SessionBean ;
import javax.ejb.SessionContext;
import java.rmi.RemoteException ;
import fr.lifl.goal.EJBPlusPlus.*;
import fr.lifl.goal.EJBPlusPlus.ExtendedEJBObjectImpl;

public class XBean extends ExtendedEJBObjectImpl implements SessionBean {
  SessionContext ctx;
  public Integer plus (Integer arg1, Integer arg2)
    throws UnknownPortException, UnknownMethodException, RemoteException {
    Object[] args = {arg1, arg2};
    Class[] argTypes = {Integer.class, Integer.class};
    return (Integer)invokeMethod("XReceptacle", "add", argTypes, args);
  }
  public Integer sub (Integer arg1, Integer arg2) throws RemoteException {
    return new Integer(arg1.intValue()-arg2.intValue()) ;
  }
  public String getName() {
    return "tartempion";
  }
  public void ejbCreate () throws java.rmi.RemoteException {
    this.setReceptacleNames();
  }
  public void ejbPostCreate() {
    this.setReceptacleNames();
  }
  public void setSessionContext (javax.ejb.SessionContext ctx) {
    this.ctx=ctx;
  }
  public javax.ejb.EJBContext getEJBContext(){
    return ctx;
  }
  public void configuration_complete()
    throws InvalidConfigurationException {
    try {
      getReceptacle("XReceptacle");
    } catch (UnknownPortException e) {
      throw new InvalidConfigurationException();
    }
  }
  public void setReceptacleNames(){
    this.receptacleNames.addElement("XReceptacle");
  }
  public void ejbRemove() {}
  public void ejbActivate() {}
  public void ejbPassivate() {}
}
```

code client

```
import xComponent.*;
public class main {

  public static void main(String[] args) {
    javax.naming.InitialContext ctx= null;
    try {
      // rechercher et instanciation d'un composant xComponent
      xComponent.XRemoteHome unComposantHome=null ;
      xComponent.XRemote unComposant ;*
      ctx= new javax.naming.InitialContext();
      Object ref1= ctx.lookup("xcomp") ;
      unComposantHome =
        (xComponent.XRemoteHome)javax.rmi.PortableRemoteObject
        .narrow(ref1,xComponent.XRemoteHome.class );
      unComposant = unComposantHome.create() ;

      // exploitation du composant xComponent via la facette XFacet1
      xComponent.XFacet1 uneFacetteDuComposantX =
        (xComponent.XFacet1)unComposant.getFacet("xComponent.XFacet1");
      System.out.println(uneFacetteDuComposantX.plus(new Integer(5),
                                                     new Integer(1)));
      // rechercher et instanciation d'un composant yComponent
      Object ref2= ctx.lookup("ycomp") ;
      yComponent.YRemoteHome unComposantYHome=null ;
      yComponent.YRemote unComposantY ;
      unComposantYHome =
        (yComponent.YRemoteHome)javax.rmi.PortableRemoteObject
        .narrow(ref2,yComponent.YRemoteHome.class );
      unComposantY = unComposantYHome.create() ;

      // exploitation du composant yComponent via la facette YFacet
      System.out.println(unComposantY.getFacet(''yComponent.YFacet'')
                                    .add(new Integer(9),
                                         new Integer(4)));

      // connexion des deux composants
      unComposant.connect("XReceptacle",
                          (yComponent.YFacet)unComposantY
                          .getFacet("yComponent.YFacet"));
      // vérification de la configuration
      unComposant.configuration_complete();

      // appel de la méthode plus, déléguée au composant Y
      System.out.println(((XFacet1)unComposant
                         .getFacet("xComponent.XFacet1"))
                         .plus(new Integer(5),new Integer(4)));
    } catch (Exception e){
      e.printStackTrace();
    }
  }
}
```

Résumé de la thèse

Résumé de la thèse :
Définition d'une démarche de conception de systèmes à base de composants

Ce travail s'inscrit dans le domaine du génie logiciel et traite de la conception d'applications distribuées à base de composants. La plupart des industriels utilise des technologies à base de composants telles que les environnements distribués Corba ou EJB. La notion de composants améliore la qualité du logiciel, c'est une unité de code robuste et éprouvée. Or, la conception d'applications grâce à des démarches de type Processus Unifié, ne garantit pas l'identification et donc la réutilisation de composants préfabriqués. En fait, l'identification, la réutilisation et la conception de composants ne sont pas correctement prises en compte par les concepts introduits par la notation UML. Enfin, le portage des modèles UML vers les plate-formes technologiques n'est pas direct et demande de nombreux choix de conception, ce qui rend la tâche complexe et coûteuse.

L'objectif de la thèse est de fournir aux concepteurs un cadre méthodologique, ainsi qu'un environnement complet pour la conception d'applications à base de composants et la génération de code vers des technologies spécifiques permettant le développement d'applications dans des environnements distribués. Nous proposons un formalisme de modélisation à base de composants articulé autour de quatre vues correspondant à quatre préoccupations de l'ingénierie logicielle. La vue de cas d'utilisation permet de décrire les besoins des utilisateurs finaux et les dépendances entre composants au niveau fonctionnel. La vue d'interactions des éléments du système donne une représentation plus fine des dépendances entre composants par échange de message et permet de définir les interfaces. La vue de conception de composants décrit la structure des composants. La vue d'assemblage des composants logiques décrit le modèle du système. Chaque vue est décrite par une partie du méta-modèle définissant les concepts mis en jeu et leurs relations. Ce méta-modèle garantit la cohérence entre les vues et les activités sous-jacentes.

Ce formalisme, ainsi que la démarche améliorent le portage des composants modélisés vers des composants de plate-formes technologiques spécifiques. Cependant, la projection vers ces plate-formes reste difficile. Elle peut être assistée par le biais de projections vers des modèles intermédiaires. La thèse présente l'implantation d'une chaîne globale de conception de composants qui s'appuie principalement sur l'atelier ouvert Eclipse utilisé aussi bien dans l'industrie que dans le monde académique. L'implantation d'un modèle intermédiaire par un framework enrichissant le modèle EJB, montre que cette démarche augmente la productivité de systèmes à base de composants. Ils sont mieux conçus et les possibilités de courtage et de réutilisation de composants préfabriqués sont accrues. Enfin, l'étude d'un mécanisme d'extensibilité de notre modèle offre des perspectives d'adaptation de notre travail à d'autres paradigmes et d'autres démarches qui devraient permettre une exploitation plus large de notre travail.

Abstract :
Component based approach approach to design information system

This work is about software engineering and more precisely about component based distributed application building. Most of industrials use component based technologies such as CORBA distributed environments or Enterprise Java Beans. The concept of component improves software quality, as a robust and tested unit of code. However, application design drived by methodology like Unified Process, does not guarantee identification and thus re-use of pre-build components. In fact, identification, re-use and design of component are not correctly taken into account by concepts introduced by the UML notation. Lastly, the mappig of UML models towards technological platforms isn't direct and requires many design choices, which is more complex and expensive.

The goal of the thesis is to give designer a methodology framework, as well as an environment to component based application building and code generation towards specific technologies that allow application development in distributed systems. We propose a component based modelisation notation articulated around four views corresponding to four software engineering concerns. Use case view describes final user needs and functional dependencies between components. Interaction view give a thiner representation of component dependencies illustrated by message exchanges. This view allows to find component interfaces. Design view describe component structure. Assembly view shows how to connect component to realize the system. Each view is described by a part of the metamodel which defines concepts at work and their relations. This metamodel guarantees coherence between views and activities needed to complete each views.

This notation, and the approach, improves the mapping from logical component towards specific technological platform components. However, the mapping towards these platform remains difficult. It can be assist by projection via intermediate models. The thesis presents implementation of a component product line on the Eclipse development environment. It is used in industrials as welle as in university domain. Implementation of a framework give an intermediate model implementation that enriches EJB model. It shows that this approach increase component based system productivity. It products a better design and component are more specified. Lastly, the study of a extensibility mecanism and other approach should allow a more large exploitation of our work.