UNIVERSITE SAAD DAHLAB DE BLIDA

Faculté des Sciences
Département d'Informatique

MEMOIRE DE MAGISTER

Spécialité : Systèmes d'Informations et de Connaissances

SPECIFICATION D'INTERFACE HOMME MACHINE

SELON UNE APPROCHE ARCHITECTURE LOGICIELLE

Par

CHERFA Imane

Devant le jury composé de :

H. ABED	Professeur, U. de Blida	Présidente
N. BENBLIDIA	Maître de Conférence A, U. de Blida	Examinatrice
N. BOUSTIA	Maître de Conférence B, U. de Blida	Examinatrice
A. HACHICHI	Maître de Conférence B, USTHB	Examinatrice
D.BENNOUAR	Maître de Conférence A, U. de Blida	Promoteur

Blida, Avril 2012

RESUME

Face à la croissance fulgurante de la taille et de la complexité des systèmes informatiques, les architectureslogicielles s'imposent comme un allié précieux, pour la conception des systèmes. Notre travail s'inscrit dans le cadre des architectures logicielles à base de composants. Il adresse la problématique d'assemblage des composants d'interface homme machine (IHM), dans les architectures logicielles. Notre travail trouve ses origines dans l'incapacité des modèles actuels tel que EJB, .NET ou fractal à supporter la notion de composition d'interface homme machine, malgré leur capacité à gérer les différents aspects non fonctionnels. En effet, une application construite par assemblage de composants, verra ainsi son IHM conçue selon les approches classiques. Ceci va conduire à deux difficultés principales : les IHMs deviennent moins portables, et seront peu réutilisables. Pour répondre à la problématique abordée, nous avons dans un premier temps extrait les causes, qui rendent les modèles actuels inadaptés pour la spécification des IHMs. Enfin, nous avons proposé dans le contexte de l'approche IASA (IntegratedApproach for Software Architecture), un modèle de composants pour la spécification d'interface homme machine. Ce modèle permet d'associer pour chaque composant d'IHM, les composants métier correspondant, les liens entre les deux types de composants sont bien définis. Une application est donc enfin considérée comme un composant, lui-même constitué d'un assemblage de composants, pour l'instant, on identifie deux vues, une vue métier et une vue IHM.

Mots-clés : Architecture logicielle, modèle de composants, spécification d'IHM, composition d'IHM, vue métier, vue IHM.

REMERCIEMENTS

Il est clair que l'accomplissement de ce travail, doit énormément au soutien des gens, qui m'ont entourés et compris, dans le cadre professionnel, aussi bien que dans le cadre personnel. Je tiens donc à les remercier au travers de cette page.

Je tiens à remercier notre créateur letout miséricordieux, pour m'avoir permis d'achever ce travail.

Je tiens à présenter toute ma gratitude, et mes sincères remerciements, à mon promoteur, monsieur Djamal Bennouar, pour avoir dirigé ce travail, et pour sa grande disponibilité, pendant ces années de travail, Il a su m'orienter dans ce grand monde qu'est la recherche, et a fait preuve de patience. Je le remercie pour ses précieux conseils, sa compréhension et son aide inestimable.

J'aimerais exprimer ma gratitude, àMme.ABED, Mme.BENBLIDIA,Mme.BOUSTIA, et Mme.HACHICHI, pour m'avoir fait l'honneur de participer à mon jury de soutenance.

Je ne pourrais assez remercier Yahia Chaabane, et Brahim Medjahed, qui m'ont gracieusement envoyés un grand nombre d'articles.

Je pense également aux enseignants que j'ai eus, depuis le primaire jusqu'à l'université. J'ai rencontré parmieux beaucoup de gens passionnés, et passionnant qui m'ont encouragé, et m'ont appris l'envie de savoir.

Sur un plan plus personnel, je tiens à remercier mes parents, pour leur soutien sans faille, etleur aide inestimable. Je les remercie pour les corrections qu'ils ont effectuées dans ce mémoire, «j'ai du accepter un grand nombre de critiques !!! ». Un remerciement particulier à Sihem, Soussou, et Wahiba, qui m'ont toujours soutenues, et qui m'ont permis detenir le cap jusqu'au bout. Je n'oublierai pas biensur, d'adresser mes plus sincères remerciements à Zaki, Lyes et Abderezak pour leur présence, et leurs encouragements. Et enfin, un grand merci à Amine, d'avoir accepter de passer plusieurs mois sans ordinateur, pour que j'accomplisse mon travail, et pour son soutien.

Mes plus profonds remerciements, pour mes amies qui m'ont tout le temps soutenu, Khadidja, Hassiba, Imane, et Maouia.

Tous mes remerciements, et mes profondes excuses à tous ceux que j'aurai pu oublier.

Ce mémoire marque la fin d'un chemin sinueux, mais n'est que le commencement d'une route encore plus longue . . .

TABLE DES MATIERES

INTRODUCTION

Face à la complexité croissante des systèmes informatiques, l'approche « Architecture Logicielle » représente, selon un consensus largement établi dans le monde du génie logiciel, une voie très prometteuse pour la maitrise de cette complexité et des autres facteurs, liés à la qualité du logiciel, tels que la facilité de réutilisation, la maitrise de l'évolution du logiciel, la gestion efficace des changements, associés au logiciel etc.

Actuellement, un grand intérêt est porté au domaine des architectures logicielles. Cet intérêt est motivé principalement par la réduction des coûts, et des délais de développement des systèmes informatiques. L'architecture logicielle permet d'exposer de manière compréhensible et synthétique la complexité d'un système logiciel, et de faciliter l'assemblage de composants logiciels.

L'approche « Architecture Logicielle », qui est actuellement une discipline à part du génie logiciel, offre une panoplie d'outils pour laspécification, la validation et la prédiction des propriétés non fonctionnelles d'un système informatique, à un haut niveau d'abstraction. La spécification est basée sur deux concepts fondamentaux : les composants et les connecteurs.

Les travaux sur l'approche « Architecture Logicielle » ont fait l'état de nombreuses recherches et de plusieurs propositions. Ces dernières ont donné naissanceà divers modèles de composants et de connecteurs, et à divers outils de spécification et de validation d'architecture logicielle. Les outils de spécification, très liés aux modèles de composants et de connecteurs, connus sous le nom de Langages de Description d'Architecture (ADL), permettent la description de la structure d'un système comme un assemblage de composants logiciels, liés par des connecteurs.

En « Architecture Logicielle », et en général dans les approches dites à composants, les composants sont principalement considérés comme des entités *métier*. Ilssont l'incarnation de la description, par le concepteur, de la logique applicative du système en concepts clés, clairement identifiés, et encapsulent le code fonctionnel. Dans certaines approches, basées sur le concept de composants, notamment les approches dites industrielles, telles que Microsoft .Net, les EJB de

SUN Microsystems, et les CCM de CORBA, les plateformes offrent actuellement une panoplie de services, permettant de gérer différents aspects non fonctionnels d'une application, tels que la sécurité et les transactions. Cependant, ces approches, qu'elles soient académiques (Wright, Darwin, Fractal, IASA) ou industrielles (.Net, EJB, CCM), n'offrent pas de possibilités de gérer,de manière aisée et efficace, la composition d'interface homme machine. Cette situation est souvent due aux objectifs premiers de chaque approche à composants. En effet, pour la plupart des approches, l'aspect IHM ne figure pas parmi les objectifs principaux. A titre d'exemple, le composant dans l'ADL Darwin ne peut pas exprimer une IHM, parce qu'une une seule sémantique lui est associée : *le processus*. Ainsi, un composant ne permet pas d'exprimer un autre élément.L'ADL *Rapid* ne décrit pas un connecteur de manière explicite, et ne permet donc pas d'exprimer les interactions, entre vue métier et celle d'IHM. Les *Java Beans*, souvent décrits comme étant des composants, sont des modèles jugés efficaces pour la descriptiond'IHMs. Cependant, dans la réalité, les *Java Beans* sont plutôt des objets, répondant à un modèle qui facilite la construction d'Interface. Cependant, le modèle des *Java Beans* n'est pas un modèle adapté à la spécification d'architecture d'une IHM.

Problématique

L'architecture logicielle a été introduite au départ comme une nouvelle discipline du génie logiciel, orientée vers la spécification de gros logiciels, en se basant sur des composants de taille importante. C'est ce qui est souvent référencé par le terme *programmation dans le large*(programming in the large). Récemment, certaines approches, telles que IASA, ArchJava et Fractal, ont permis l'utilisation de l'architecture logicielle pour la conception d'applications, basées sur des composants très fins. A titre d'exemple, dans IASA, le concept de composants a été étendu, pour représenter les opérateurs fondamentaux (addition, soustraction etc..) et les structures de contrôle. En ArchJava, un composant est directement spécifié, en utilisant le langage Java.

Ce qui est remarquable à travers les diverses approches d'architecture logicielle, c'est leur faible intérêt pour l'aspect spécification d'IHMs. La plupart des approches se basent sur des modèles de composants généraux, orientés vers la spécification du métier. Ainsi, une application respectant le modèle MVC (Model,View, Controller), verra sa partie contrôle spécifiée selon une approche *architecture logicielle*,alors que les parties modèle et IHM, seront souvent spécifiées

selon une approche objet. Cette situation est du au fait que les modèles de composants actuels, ne supportent pas, de manière efficace, la spécification des IHMs.

Nous noterons qu'il est possible de spécifier des IHM avec les modèles généraux actuels, mais, aux prix d'une grande difficulté, et d'efforts très importants de la part de l'architecte d'application. Cette dernière situation est semblable à la programmation orienté objet, avec un langage modulaire tel que C. Dans ce type de situation, l'approche « Architecture Logicielle » ne pourra pas répondre correctement à un de ses objectifs importants, qui est la réalisation rapide d'applications.

Objectifs

Actuellement, dans le monde de la programmation orienté objet, la construction d'IHMsest devenue très rapide et très efficace. Le concepteur n'a plus à écrire les parties du programme qui se chargent de dessiner et de placer les divers composants visuels, tels que les boutons, les zones de texte et les menus. De plus, le concepteur est déchargé de la déclaration des procédures qui prennent en charge le traitement des divers événements, que produisent les divers éléments de l'IHM (click de souris, double click, enfoncer une touche du clavier etc..). Le concepteur commence plutôt par dessiner son interface, et c'est l'IDE (IntegratedDevelopmentenvironment) qui se chargera de générer le code, pour produire l'interface et le code qui prendra en charge les événements. La liaison, entre la partie d'un programme qui se charge de dessiner l'IHM, et la partie qui se charge du métier ou du modèle, se fait par l'ajout d'un nombre très réduit d'instructions, au niveau des procédures qui captent les événements, produits par les composants visuels (click de souris, saisie d'un caractère, etc..). En orienté objet, la construction d'IHMs, dans le contexte d'IDE, a effectivement atteint un haut degré de maturité.

Notre objectif est de permettre à l'architecture logicielle d'atteindre ce haut degré de maturité, atteint par l'orienté objet, afin qu'elle soit une approche qui permettrait de réaliser rapidement les diverses facettes d'une application. Actuellement, en l'absence de cette capacité de construire aisément les IHM selon une approche à composants, les applications devront construire leurs IHMs, selon un modèle objet par exemple, et le reste, selon un modèle de composants. Ainsi, deux modèles seraient utilisés pour bâtir une application. La conception

totale d'une application, basée sur un même modèle, donnera au processus de conception une homogénéité, qui permettrait de le rendre plus efficace.

Un modèle de composants, efficace pour la construction des IHMs par assemblage de composants, permettrait, par exemple, d'obtenir un assemblage de composants, représentant l'IHM, directement à partir du dessin de l'IHM, à base de composants visuels. Si l'on fait une comparaison avec l'orienté objet, la vue assemblage de composants de l'IHM, correspond à la vue code source, généré automatiquement, lors de la construction des IHMs dans les IDEsorientés objet. Un tel modèle de composants nous permettrait ainsi, de construire des IDEs, dans lesquels nous pouvons avoir les deux vues d'une IHM : La vue présentation réelle, et la vue logique, qui expose l'assemblage des composants visuels, et leurs connexions aux autres composants métier. Si l'on fait une autre comparaison avec l'orienté objet, la réalisation d'une connexion, dans la vue logique d'une spécification d'architecture logicielle de l'IHM, correspond à la vue code source, dans laquelle le programmeur ajouterait les instructions nécessaires, pour lier le métier (ou le modèle) à l'IHM, au niveau de la procédure qui capte un événement, issu d'un composant visuel.

Lors de la spécification d'une application selon une approche *architecture logicielle*, l'IHM sera construite, en plaçant les composants visuels aux endroits les plus adéquats : c'est la conception de la vue. La vue doit être liée au métier, et parfois au modèle. Ces derniers sont spécifiés, selon une approche*architecture logicielle*. Le passage de la vue présentation à la vue architecture (assemblage de composants), devra nous montrer les composants visuels et leurs connexions aux composants métier. Ainsi, pour une IHM, nous aurons deux vues : La vue de *présentation* et la vue *architecture*.

L'objectif de ce travail de recherche, est de faire une étude approfondie des modèles de composants existants, et de déterminer les causes qui les rendent inadaptés à la spécification des IHMs. En effet, faire la spécification architecturale d'une IHM, revient à exprimer quatre éléments essentiels dans une architecture :

- L'expression de la structure d'une IHM, et donc permettre l'assemblage des composants visuels.
- L'application des modifications dans les attributs relatifs aux composants visuels.
- L'expression du comportement de l'IHM (acquisition de données, …).

- Enfin, l'expression des liens existants entre les composants métier et les composants d'IHM.

Après cette étude, nous devons proposer un modèle de composants, adapté à la spécification des IHMs, selon une approche *architecture logicielle*. La validation de ce modèle se fera par la réalisation d'un IDE, basé sur ce modèle de composants. Cet IDE permettrait la spécification de toute une application, y compris son IHM.

Organisation du mémoire

Ce mémoire est organisé en cinq chapitres :

Le premier chapitre aborde le domaine des architectures logicielles. Il expose les concepts, les définitions et la terminologie des architectures logicielles.

Le deuxième chapitre présente le domaine des interfaces homme machine, et présente les travaux existants sur ce thème. Il introduit les notions, liées aux interfaces homme machine et aux systèmes interactifs. Il expose brièvement les techniques utilisées actuellement pour la spécification d'interfaces homme machine, et aborde leurs limites. Ce chapitre présentera par la suite les langages de description d'interface homme machine, qui représentent la seule contribution de composition dans ce domaine, et montre leurs limites.

Le troisième chapitre sera consacré à un état de l'art concernant la prise en charge des IHM par l'*architecture logicielle*, et les approches basées sur les composants. Nous commencerons ce chapitre par la présentation des divers langages de description d'architecture, des modèles de composants académiques et industriels. Une discussion sur l'importance donnée aux IHMs dans les diverses approches sera présentée, et mettra en évidence les causes qui rendent les modèles de composants actuels inadaptés à la spécification de l'architecture logicielle d'une interface homme machine.

Le quatrième chapitre aborde le traitement du problème de composition des IHMs, en s'appuyant sur l'état de l'art, présenté dans les premiers chapitres, pour motiver notre approche.Il présente ensuite notre contribution, à savoir un modèle de composants pour la spécification d'interface homme machine, dans le contexte de l'approche IASA (IntegratedApproach Software Architecture).

Le dernier chapitre sera consacré à la validation expérimentale de notre modèle de composants. Les résultats obtenus seront utilisés pour améliorer le logiciel IASASTUDIO. Ce dernier devra être enrichi par une nouvelle vue, qui s'intéresse à la construction interactive d'IHM, tout en maintenant la synchronisation avec la vue spécification architecturale de l'IHM, et sa relation avec la vue métier d'une application

Nous terminerons ce document par une conclusion générale, dans laquelle seront présentés un bilan et les perspectives des travaux réalisés.

CHAPITRE 1
INTRODUCTION A L'ARCHITECTURE LOGICIELLE ET AUX APPROCHES ORIENTEES COMPOSANTS

1.1 Introduction

L'Architecture logicielleest considérée comme une sous-discipline du génie logiciel. Une architecture est considérée comme l'organisation nécessaire d'un système, caractérisé par ses composants, leurs relations avec l'environnement, et les principes qui guident leur conception et évolution. Les architectures logicielles forment la colonne vertébrale de la construction des logiciels complexes et de grande taille. La description des architectures permet d'avoir l'abstraction nécessaire pour modéliser les systèmes logiciels complexes durant leur développement, déploiement et évolution.

Le but de ce chapitre est d'apporter les définitions et les notions de base relatives de l'ArchitectureLogicielle, ainsi que les modèles utilisés pour la spécification d'architecture.Pour cela, nous présentons quelques définitions de la notion d'architecture proposées dans la littérature, et nous décrivons les différents éléments architecturaux. Enfin, nous terminons ce chapitre enprésentant les apports de l'architecture logicielle dans la conception de logiciels.

1.2 Concepts et terminologie de l'Architecture Logicielle

Les travaux sur les architectures logicielles ont connu une effervescence au début des années 90. Ces travaux soulignaient l'importance d'avoir une structure globale à un niveau d'abstraction élevé d'un système avant sa construction. Cette structure globale devait aboutir à des applications plus structurées, plus faciles à comprendre et plus facile à faire évoluer. Bien que tous ces travaux s'accordent sur l'importance de l'architecture logicielle pour le développement des systèmes à base de composants, la définition consensuelle de la notion d'*architecture logicielle* n'a pas pu être établie. Même si de nombreuses définitions ont été

proposées, aucune ne s'est vraiment imposée. La page web dédiée à ce sujet du Software Engineering Institute **[1]** témoigne de la diversification de ces définitions. Dans cette section, nous présentons quelques définitions de la notion d'*architecture logicielle*.

Une définition couramment admise de l'architecture logicielle est celle de BASS, CLEMENTS et KAZMAN **[2]** (*cf.* Définition 1.1). Cette définition décrit l'architecture comme une abstraction constituée d'éléments logiciels et leurs relations. Cependant, elle ne décrit pas la nature des entités logicielles représentées dans l'architecture. Elle permet ainsi à chacun de choisir les entités et peut donc être utilisée dans toutes les communautés utilisant le concept d'architecture.

Définition 1.1. L'architecture logicielle d'un programme ou d'un système est la ou les structures du système, c'est-à-dire en particulier les éléments logiciels, les propriétés visibles extérieurement de ces éléments et leurs relations.

Après avoir examiné les architectures dans d'autres disciplines (matériel, réseaux et bâtiments), Perry et Wolf **[3]** décrivent plus clairement les entités qui composent l'architecture (*cf.* Définition 1.2)

Définition 1.2. ... l'architecture logicielle est un ensemble d'éléments architecturaux qui ont une forme particulière. Les éléments architecturaux sont divisés en trois catégories : les éléments de calcul ; les éléments de données ; et les éléments de connexion.

Dans cette définition, l'architecture est donnée en énumérant les propriétés des différents éléments et les relations entre ces éléments. Un autre élément essentiel de l'architecture selon Perry et Wolf est sa logique (*rational* en anglais) qui englobe, entre autres, les aspects de qualité.

Par la suite, la définition des éléments architecturaux a progressé et un consensus s'est formé sur leurs noms et leurs rôles dans l'architecture. Chaque élément architectural est ainsi nommé et leur rôle dans l'architecture est clairement défini. On adopte la définition 1.3 qui est

suffisamment souple pour offrir une grande marge de manœuvre dans la description des éléments architecturaux **[4]**.

Définition 1.3. L'architecture est une vue abstraite d'un système en terme d'éléments architecturaux.Ces éléments sont :

- Les composants qui décrivent les fonctionnalités métier de l'application ;
- Les connecteurs qui décrivent les communications et connexions entre les composants;
- La configuration qui décrit la topologie des connexions entre composants et connecteurs.

Dans ce qui suit, nous présentons les différents éléments architecturaux. Pour chacun, nous proposons une synthèse des points communs entre les définitions couramment utilisées.

1.2.1Les composants

Les composants sont les éléments architecturaux qui encapsulent la partie métier de l'architecture. Il peut être aussi petit qu'une procédure simple ou aussi grand qu'une application **[5]**. Un serveur, une base de données, une fonction mathématique sont des exemples de composants. Les définitions de cet élément sont nombreuses. Néanmoins celle de SZYPERSKI **[6]** est la plus répandue dans la littérature (*cf.* Définition 1.4). Cette définition souligne les propriétés fondamentales du composant. Elle présente également une partie des éléments externes représentant un composant : les interfaces fournies et requises.

Définition 1.4. Un composant logiciel est une unité de composition possédant des interfaces spécifiées par contrat et des dépendances contextuelles explicites. Un composant logiciel peut être déployé indépendamment et est sujet à la composition par un tiers.

Le consensus relatif autour de cette définition provient sans doute du fait qu'elle englobe les définitions des composants issues de différents domaines. Ainsi, elle permet de décrire aussi bien un composant abstrait[1]tel que ceux présents dans les premières phases de conception qu'un

[1]référencé aussi par le terme composant académique

composant exécutable[2]selon un modèle tel que CCM (Corba Component Model) **[7]** ou COM (Component Object Model) **[8]**, par exemple. Cette généricité est due à la fois au manque de précision sur les structures externes du composant et sur l'absence de description des structures internes du composant.

Au delà de cette définition, les composants ont bénéficié des rapprochements entre les différentes communautés utilisant l'architecture logicielle. La définition a ainsi progressivement évolué pour préciser les éléments communs aux différentes communautés. Grâce à cette évolution, nous pouvons compléter la définition des composants en décrivant plus précisément leurs structures externes et internes ainsi que leurs relations de composition.

1.2.1.1Structure externe d'un composant

La structure externe d'un composant est généralement caractérisée par deux types d'éléments **[9]**. Les premiers sont les interfaces du composant. Elles sont la spécification des services fournis et requis par le composant. Les seconds sont les propriétés du composant. Elles servent à documenter l'architecture en décrivant les aspects relevant de la conception ou de l'analyse du composant.

1.2.1.1.1Les interfaces

Les interfaces sont la partie visible d'un composant (*cf.* Figure 1.1). Elles servent à déclarer les services fournis et requis par le composant et constituent, d'après la définition 1.5 proposée par CHEFROUR **[10]**, la caractéristique majeure de celui-ci.

Définition 1.5. Un composant est une entité logicielle qui fournit un service particulier via une interface séparée de l'implantation mettant en œuvre ce service.

Une interface est composée de deux dimensions :

[2]référencé aussi par le terme composant industriel

- Une dimension sémantique : les interfaces décrivent la sémantique des fonctionnalités fournies et requises proposées par le composant. Ces fonctionnalités, appelées aussi services, déterminent le type de l'interface. Ainsi, si le service associé à une interface décrit le comportement fonctionnel du composant, c'est une interface fournie. Au contraire, si le service décrit les fonctionnalités dont le composant en a besoin pour fonctionner, l'interface est une interface requise.

- Une dimension structurelle : en plus de décrire les fonctionnalités du composant, l'interface est le point d'interaction entre le composant et son environnement. Cet aspect structurel est parfois décrit comme une entité séparée, appelée port. C'est cette dimension qui est représentée lorsque l'on schématise un composant (*cf.* Figure 1.1).

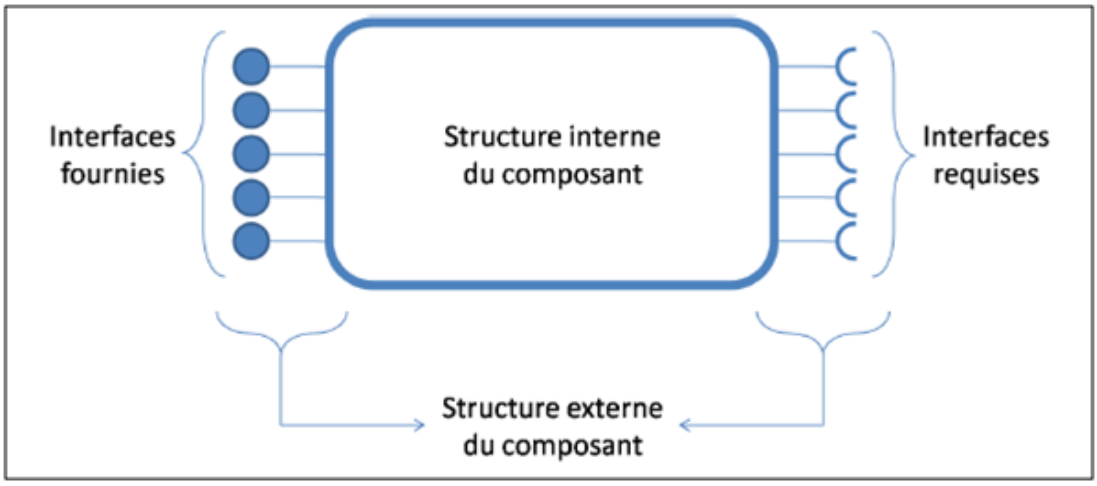


Figure.1.1: Représentation d'un composant

1.2.1.1.2Propriétés des composants

Les propriétés des composants sont de trois types :

- Propriété non fonctionnelle d'un composant: la propriété non fonctionnelle d'un composant représente toute contrainte particulière liée à l'environnement d'utilisation du composant. Les propriétés non fonctionnelles sont nécessaires pour permettre la simulation du comportement des composants, leur analyse, leur traçabilité depuis leur conception jusqu'à leur implémentation et l'aide dans la gestion de projet (par exemple, en définissant des conditions

de performance rigoureuses, le développement d'un composant peut devoir être assigné au meilleur ingénieur).

- Contrainte d'un composant : la contrainte d'un composant précise des restrictions qui doivent être vérifiées afin de respecter les utilisations prévues d'un composant et d'établir les dépendances parmi ses éléments internes. Les contraintes sont des propriétés qui doivent être vérifiées pour que le système soit considéré comme cohérent.

- Contrats : ces propriétés portent sur les interfaces du composant. Les contrats portant sur les interfaces fournies définissent des contraintes et garantissent que les services de l'interface respectent ces contraintes. Les contrats portant sur les interfaces requises imposent aux composants fournisseurs un ensemble de contraintes qui doivent être vérifiées par les services fournis au travers de leurs contrats.

1.2.1.2Structure interne d'un composant

Il existe deux types de structures internes des composants. Pour un premier type, la structure interne est représentée par une description de l'implémentationdes fonctionnalités du composant (code source dans un langage de programmation, code binaire etc.). Les composants possédant une telle structure interne sont dits des composants atomiques. Ils représentent les blocs de base de l'architecture.

Le second type de structures internes est composé d'autres composants. Ces composants sont des composites constitués de composants internes. Ces composants internes peuvent eux aussi être atomiques ou composites. Comme le composant atomique, le composant composite dispose de ses propres interfaces. Les fonctionnalités proposées ou utilisées par ces interfaces peuvent alors être déléguées aux composants internes ou directement pris en charge par l'implémentation du composant composite. Pour réaliser cette délégation et pour interagir avec ses composants internes, le composant composite possède également des interfaces internes qui comme pour les externes peuvent être fournies ou requises.

Néanmoins, si la structure interne de tous les composants composites est constituée de composants, les composants composites peuvent être classés selon deux axes :

- Les interactions entre les composants internes : soit les composants internes peuvent interagir directement entre eux, soit les composants internes n'interagissent qu'avec le composant composite. Dans ce dernier cas, le composite peut avoir à jouer le rôle de médiateur entre les différents composants internes ;

- La délégation des interfaces : soit le composant composite peut prendre en charge les interfaces, soit il délègue ces interfaces aux composants internes à travers des ports spécifiques. Dans ce dernier cas, le composant composite constitue une enveloppe pour les services fournis par ses composants internes. Au contraire, dans le premier cas le composite propose des services supplémentaires par rapport à ses composants internes. La figure 1.2 illustre les quatre cas possibles en fonction de ces deux axes.

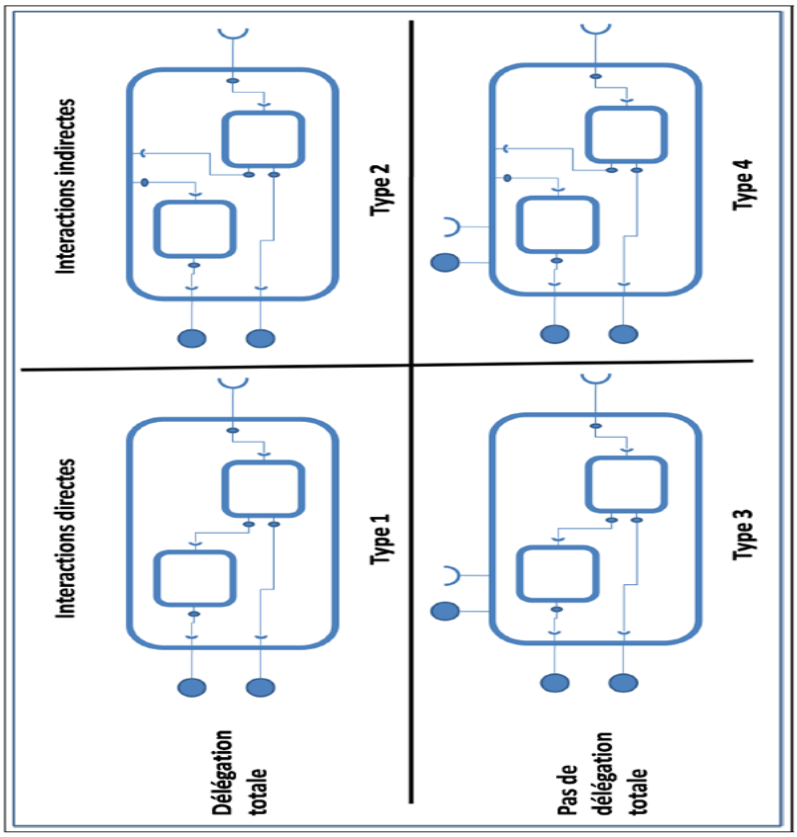


Figure.1.2 : Exemple de structure interne pour les composants composites

1.2.1.3Relation de composition des composants

Les relations de compositions entre composants sont de deux types. Il existe d'abord des relations de compositions horizontales. Ce sont les relations déterminées par les interfaces requises et les connecteurs. Ces relations permettent d'assembler les composants et les connecteurs. Elles n'imposent pas de contrainte particulière sur les éléments impliqués. A ce titre, ces relations sont les plus simples à utiliser pour assembler des composants provenant de sources diverses.

Les autres relations sont dites verticales. Ces relations représentent les relations entre un composant composite et ses sous-composants. Elles possèdent une sémantique différente de celle de la composition horizontale et constituent la différence majeure entre les composants composites et les composants atomiques. En effet, contrairement aux relations de compositions horizontales, la composition verticale impose des contraintes strictes sur les différentes parties impliquées. Ces contraintes varient en fonction du type de la relation qui peut être plus ou moins forte **[11]**. Par exemple, une relation de composition verticale forte implique la simultanéité dans la création et la destruction du composites et des sous composants. Ces contraintes sont associées à la sémantique de la relation de composition verticale, mais elles ne sont pas représentées directement dans l'architecture. Ainsi, ces relations apparaissent, comme les relations horizontales, à travers des interfaces requises ou fournies qui relient le composite et les sous-composants.

1.2.2Les connecteurs

Les connecteurs constituent un élément architectural au même titre que les composants. Ainsi, ce sont les principales entités dans une architecture. Cette considération identique pour les composants et les connecteurs est l'atout principal des architectures. En effet, la distinction entre les aspects métiers et ceux de communication se fait plus facilement en considérant de la même manière les deux entités **[4]**.

A la différence des composants, les définitions des connecteurs sont peu nombreuses et relativement convergentes. Par exemple, SHAW et GARLAN **[12]** présentent les connecteurs selon la définition 1.6.

Définition 1.6. Les connecteurs gèrent les interactions entre les composants ; c'est-à-dire, ils établissent les règles qui gouvernent les interactions entre composants et spécifient tous les mécanismes auxiliaires nécessaires.

D'après cette définition, Les connecteurs sont des entités architecturales qui relient des ensembles de composants et agissent en tant que médiateurs entre eux. Les exemples de connecteurs incluent des formes simples d'interaction, comme des pipes, des appels de procédure et l'émission d'évènements. Les connecteurs peuvent également représenter des interactions complexes, comme un protocole client/serveur ou un lien SQL entre une base de données et une application **[13]**.

Cette définition propose une description des rôles des connecteurs. De la même manière, dans la plupart des travaux, l'aspect structurel est laissé de coté au profit de l'aspect fonctionnel. Ceci permet de conserver suffisamment de généricité et de pouvoir utiliser toutes entités remplissant le rôle adéquat **[4]**. Cependant, l'augmentation de l'intérêt pour cet élément architectural à conduit à une formalisation de sa structure. Nous pouvons ainsi décrire plus précisément les structures internes et externes des connecteurs.

1.2.2.1Aspect fonctionnel du connecteur

L'aspect fonctionnel des connecteurs est décrit en détail dans la taxonomie de MEDVIDOVIC **[14]**.Elle classe les connecteurs selon deux niveaux décrivant les services et les techniques utilisés par les connecteurs.

Le premier niveau de la taxonomie de Medvidovic contient quatre types de connecteurs qui se distinguent suivant les services qu'ils proposent :

- La communication: les connecteurs de communication transfèrent les données entre les composants. Ce service est l'élément de base de l'interaction entre composants. Les connecteurs de communication permettent aux composants de transmettre les messages, d'échanger les données ou de communiquer les résultats des calculs ;

- La coordination: les connecteurs de coordination gèrent le transfert de contrôle entre les composants, permettant aux composants d'interagir par transfert du flot d'exécution. Les

appels de fonctions ou les invocations de méthodes sont des exemples de connecteurs de coordination simple ;

- La conversion: les connecteurs de conversion servent d'interprètes entre composants hétérogènes, c'est-à-dire provenant de sources différentes et surtout non prévus pour interagir à l'origine. Les connecteurs convertissent le service fourni par un composant pour permettre à un autre de recevoir son service requis sous le bon format. Les conversions peuvent porter sur le nombre ou le type des données échangées, sur la fréquence ou l'ordre des interactions ou encore sur le nom des services ;

- La facilitation: même lorsque les composants sont prévus pour être assemblés entre eux, il peut être nécessaire d'introduire des mécanismes pour faciliter et optimiser leurs interactions. Les connecteurs de facilitation offrent donc des services tels que l'équilibrage des charges, la planification ou encore le contrôle de la concurrence.

Les services permettent de faire un premier classement des connecteurs mais ils laissent de coté de nombreux détails essentiels sur leur mise en pratique par les connecteurs. Pour répondre à cela, le second niveau classe les composants en fonction de la manière dont ils réalisent le service. Les huit types proposés peuvent être utilisés pour proposer plusieurs services. Leurs mises en pratique dépendent de plusieurs paramètres qui sont représentés, dans la taxonomie, par leurs dimensions et dont les valeurs possibles sont représentées par les valeurs de la taxonomie. Les huit types sont :

- Les appels de procédures : ils permettent aux connecteurs de proposer les services de coordination et de communication. Dans le premier cas, le flot de contrôle est modélisé par différentes techniques d'invocation. Dans le deuxième cas, le passage des paramètres permet le transfert de données d'un composant à un autre ;

- Les évènements **:** un évènement est un effet instantané de la fin (normale ou anormale) de l'invocation d'une opération sur un objet. Elle se produit dans le contenant de l'objet **[15]**. Ces évènements permettent une modélisation du flot de contrôle similaire aux appels de

procédures. Ce type de connecteurs propose donc des services de coordination. De plus, les messages associés à chaque évènement peuvent être adaptés pour contenir plus d'informations et ainsi permettre de fournir un service de communication ;

- Les accès aux données : les connecteurs d'accès aux données permettent aux composants d'interagir avec des composants de stockage de données. Ils offrent ainsi un service de communication. Cependant, la communication avec ce type de composant peut nécessiter des post ou prétraitements tel qu'une conversion de format des données ou encore un nettoyage après la suppression de données. Le connecteur propose alors un service de conversion ;

- Les liens : les connecteurs de type « liens » permettent la mise en place de canaux de communication entre les composants. Ces canaux sont ensuite utilisés par les autres connecteurs pour mettre en place leurs services. Les connecteurs liens proposent donc un service de facilitation, en particulier de la phase de mise en place du système. Ils peuvent d'ailleurs disparaitre après cette phase ;

- Les flots **:** les flots permettent le transfert de grande quantité de données. Ils sont donc utilisés par les connecteurs pour proposer un service de communication. Ils sont particulièrement mis à contribution lorsque la communication requiert un protocole particulier et complexe ;

- Les arbitres : les connecteurs arbitres organisent le fonctionnement du système. Ainsi, ils peuvent remplir deux services. Le premier est un service de facilitation. En effet, les connecteurs arbitres fournissent les outils pour négocier le niveau des services ou encore les interactions nécessitant un certain niveau d'isolation ou de fiabilité. L'autre service rendu est la coordination. En effet, les connecteurs arbitres peuvent rediriger le flot d'exécution entre les composants. Cette redirection peut dépendre, par exemple, d'une planification ou d'un objectif d'équilibrage des charges entre les composants ;

- Les adaptateurs : les connecteurs adaptateurs proposent un service de conversion. L'adaptateur peut ainsi permettre l'interaction entre composants qui ne sont pas prévus pour communiquer. Pour cela, il doit adapter la politique de communication et le protocole d'interaction des composants. Par exemple, les adaptateurs permettent de résoudre les

problèmes de polymorphisme en faisant le lien entre des interfaces fournies et requises qui se complètent mais n'ont pas le même nom ;

- Les distributeurs : les connecteurs distributeurs proposent un service de facilitation. Ils sont chargés d'identifier les chemins d'interaction et de fournir des informations de routage pour la communication ou la coordination entre les composants. Ces connecteurs ne sont jamais utilisés directement, mais fournissent plutôt une assistance aux autres connecteurs pour remplir leurs fonctions.

1.2.2.2Structure externe du connecteur

Un connecteur est caractérisé par deux éléments **[16]**: ses interfaces qui spécifient les types et le rôle des composants communicant à travers le connecteur ; et ses propriétés qui décrivent les aspects relevant de la conception ou de l'analyse des connecteurs et documentent l'architecture d'une manière similaire aux propriétés des composants.

1.2.2.2.1Les interfaces

Les interfaces, appelées rôles dans certains langages de description d'architecture tel que Wright **[17]**, sont la partie visible du connecteur (*cf.* Figure 1.3). Elles servent à déclarer les participants à l'interaction décrite par le connecteur. Comme celles des composants, elles constituent les points de connexion entre les connecteurs et les composants. Néanmoins, à la différence des composants, les interfaces ne décrivent pas de services fonctionnels mais des mécanismes de connexion. Elles décrivent également le rôle de chacun des composants impliqués.

Contrairement aux composants, on ne distingue généralement pas de directions pour les interfaces de connecteurs. Cependant, dans le reste de ce manuscrit, nous utilisons, comme pour les composants, les appellations d'interfaces entrantes et sortantes. Les premières relient le connecteur à des interfaces fournies de composants alors que les secondes relient le connecteur à des interfaces requises de composants.

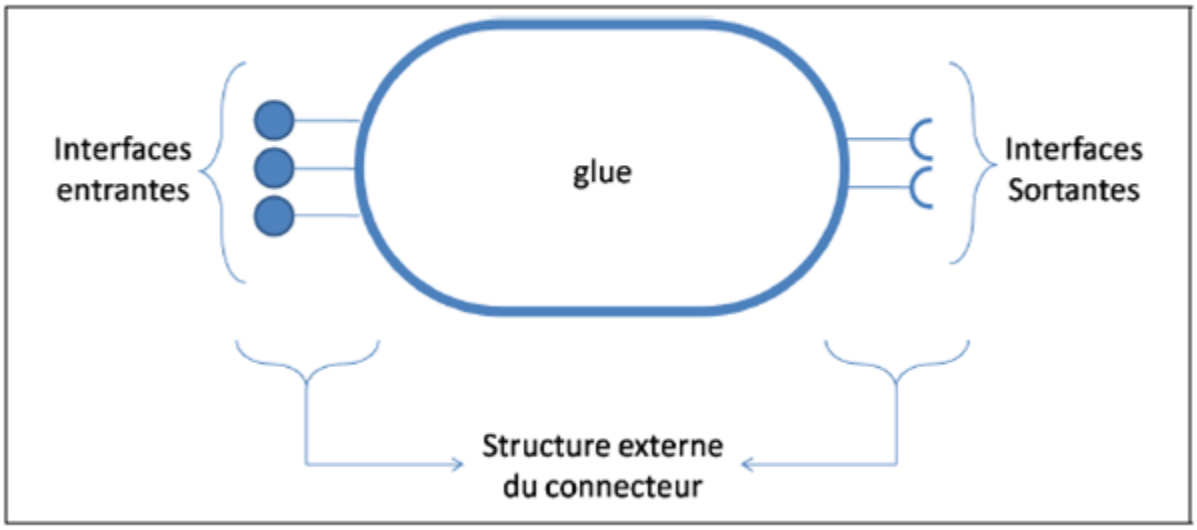


Figure.1.3 : Représentation d'un connecteur

1.2.2.2.2Propriétés des connecteurs

Les propriétés des connecteurs sont de deux types :

- Les propriétés non fonctionnelles d'un connecteur:ne sont pas forcément obtenues des spécifications de sa sémantique. Elles représentent en général les informations additionnelles nécessaires pour l'implémentation correcte d'un connecteur. Par exemple, elles peuvent concerner la performance ou la sécurité. Comme pour les composants, ces propriétés marquent une séparation claire entre les aspects fonctionnels et non fonctionnels. Elles permettent ainsi de simuler le comportement des connecteurs à des fins d'analyse, de définition des contraintes ou encore de sélection des connecteurs **[16]**;

- Contraintes: afin d'assurer les protocoles d'interaction prévus, d'établir les dépendances intra-connecteur et de fixer les conditions d'utilisation des connecteurs, des contraintes de connecteur doivent être spécifiées. Un exemple d'une contrainte simple est la restriction du nombre de composants qui interagissent à travers un connecteur donné. Comme pour les contraintes portant sur les composants, ces contraintes doivent être vérifiées pour que le système soit considéré comme cohérent.

1.2.2.3Structure interne du connecteur

La structure interne des connecteurs est une autre de leurs caractéristiques. De la même manière que pour les composants, on distingue deux types de structures internes pour les connecteurs : structure atomique ou composite.

1.2.2.3.1 Connecteur atomique

La structure interne des connecteurs atomiques est appelée glu (*cf.* Figure 1.3). Elle forme la passerelle entre les interfaces du connecteur. Pour cela, elle décrit le protocole de communication entre les interfaces, points d'accès des composants vers le connecteur.
Ce type de connecteur est le plus couramment utilisé. En effet, même si les connecteurs sont, en théorie, des entités du même niveau que les composants, dans la pratique, les outils utilisant ou décrivant les architectures considèrent les composants comme les éléments prépondérants de l'architecture. Ainsi, alors que les composants composites sont utilisés et sont répandus, les connecteurs sont souvent considérés comme des interactions simples, et sont donc décrit par des connecteurs atomiques. Par exemple, dans les langages de description d'architecture, les connecteurs sont proposés de trois façons : il existe un seul type de connecteur simple, parfois même sans représentation explicite (Rapide **[18]**) ; les connecteurs doivent être choisis dans un ensemble prédéfini (UNICON **[19]**) ; enfin, il est parfois possible de définir un connecteur atomique (Wright **[17]**, Aesop**[20]**).

1.2.2.3.2 Connecteur composite

Les connecteurs composites ont une structure interne plus complexe que celle des connecteurs atomiques. A l'image des composants composites, les connecteurs composites possèdent une structure interne composée de composants, de connecteurs et d'une configuration. Ainsi la structure interne d'un connecteur composite est une architecture interne à ce connecteur (*cf.* Figure 1.4) **[4]**. Ce type de connecteur permet, par exemple, de décrire des protocoles complexes de communication qui demandent un pré et un post-traitement des données.

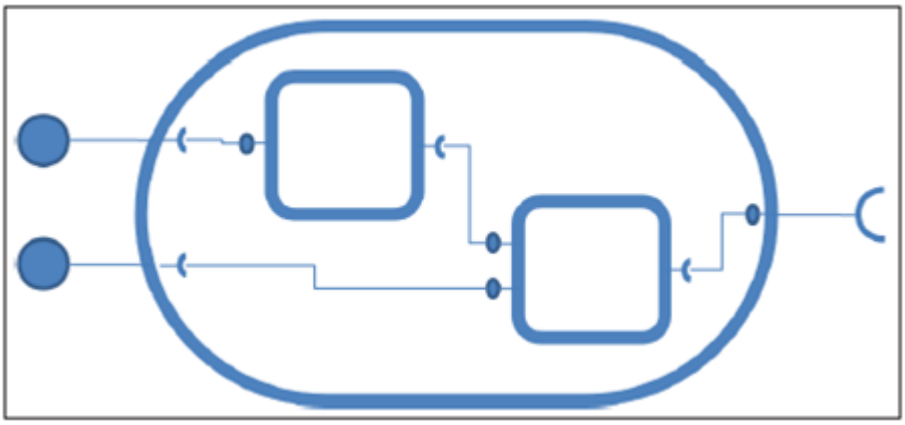


Figure.1.4 : Exemple de connecteur composite

1.3 Avantages des architectures logicielles

La conception architecturale des grands systèmes a toujours joué un rôle significatif dans la réussite des projets logiciels: le choix d'une architecture inappropriée peut entrainer des résultats désastreux. La reconnaissance actuelle de l'importance de l'architecture logicielle semble améliorer notre capacité à construire des systèmes logiciels efficaces.

L'architecture affecte la plupart des attributs d'un système **[12]**. Ces impacts ont été décrits par GARLAN et PERRY **[21]**, selon cinq angles :

- La compréhension du système: l'architecture fournie une représentation d'un système à un haut niveau d'abstraction. Cette vue synthétique du système met en valeur la plupart des décisions de conception ainsi que les conséquences de ces mauvaises décisions. En effet, l'architecture met en valeur les contraintes de haut niveau qui justement intéressent les concepteurs.

- La réutilisation: les descriptions architecturales favorisent la réutilisation à plusieurs niveaux. Les travaux en cours qui traitent la réutilisation se concentrent surtout sur les bibliothèques de composants. La conception architecturale supporte la réutilisation de grands composants *(large components)*, ainsi que les frameworks ou les composants peuvent être intégrés. Elle permet également, à travers les connecteurs, d'identifier les dépendances existantes entre ces parties réutilisables d'un système **[4].**

- L'évolution: l'architecture fournie un squelette du système. Ce squelette permet d'identifier les parties fortement utilisées ainsi que les parties potentiellement fragiles. L'architecture permet ainsi de mettre en valeur les parties nécessitant une attention particulière lors de l'évolution du système. Mais l'architecture permet également de révéler une image précise des dépendances entre les composants. Cette image est nécessaire pour connaitre les impacts de la modification d'un composant sur les autres composants du système et donc les conséquences des différentes évolutions. Elle permet aussi de modifier ces dépendances pour améliorer certains attributs du système tels que la performance ou l'interopérabilité. Enfin, l'architecture permet de corriger les erreurs à la source plutôt que là où elles apparaissent (*cf.* Figure 1.5). Ceci peut être réalisé en localisant le composant fautif ou encore certaines dépendances ou contraintes non documentées ;

- L'analyse :la vue abstraite fournie par l'architecture permet de mesurer différents attributs tels que la consistance du système, son respect du style architectural ou encore d'autres attributs de qualité. Elle permet également de vérifier que les changements prévus dans le système sont conformes au style et aux objectifs de qualité fixés à la conception ;

- La gestion de projets : l'architecture permet une gestion plus précise des coûts et des risques de modifications, en particulier en soulignant les dépendances entre les composants. Elle permet également une évaluation des qualités du système dans son ensemble, mais aussi des qualités de chaque composant. Ceci permet d'identifier les parties les plus faibles du système et ainsi d'examiner et de cibler précisément leurs faiblesses. Cette identification des composants les plus faibles permet de mettre en valeur les composants les plus problématiques et de décider de leur réingénierie ou de leur redéveloppement. Enfin, la connaissance de la valeur de chaque composant et de leurs dépendances permet de planifier la réingénierie d'un système complexe en ordonnant les modifications selon leurs impacts sur la qualité du logiciel et le risque qu'elles soulèvent.

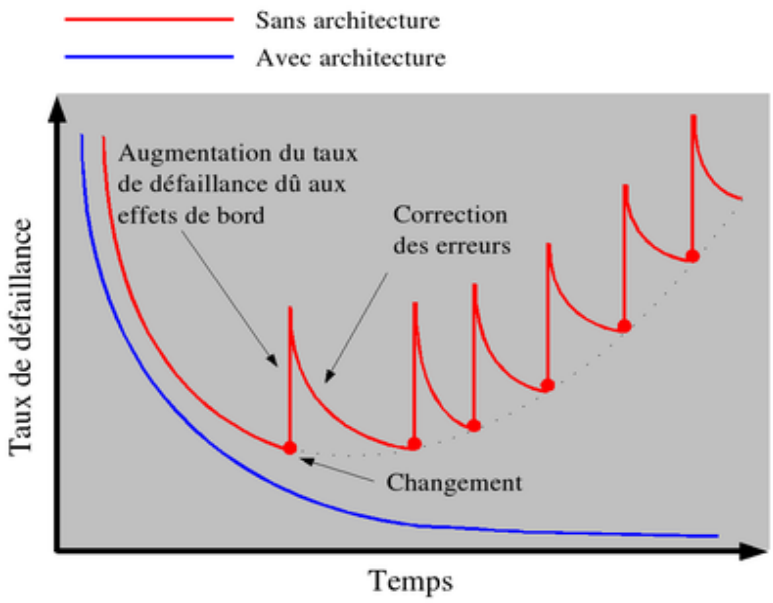


Figure 1.5 : Graphique de dégradation du logiciel en fonction de l'architecture

1.4 Définitions communautaires de l'architecture

La définition de l'architecture varie en parallèle avec les paradigmes de programmation (module, sous-système, objets, patrons, composants...) **[4]**. Cependant, ce facteur n'est pas la seule source des variations dans la définition de la notion d'architecture. Une autre cause majeure de cette hétérogénéité est l'utilisation de l'architecture dans différentes communautés scientifiques puisque leurs définitions de l'architecture sont directement issues de leurs préoccupations et de leurs historiques **[22]**.

Au final, les communautés possèdent chacune une définition propre de l'architecture. Les intérêts portés aux éléments architecturaux varient également d'une communauté à l'autre, de la même manière que l'utilisation qui est faite de l'architecture.

Les principales communautés utilisant l'architecture sont : la communauté « objet », la communauté « Langage de Description d'Architecture » (*Architecture Description Langage*, ADL), la communauté « réingénierie » et enfin la communauté « d'ingénierie logicielle à base de composants » (*Component Based Software Engineering*, CBSE).

- Architectures et communauté ADL:la recherche dans cette communauté porte sur la nécessité de trouver des notations expressives pour représenter les concepts et les styles architecturaux **[21]**. Cette communauté propose un ensemble de langages plus ou moins formels pour décrire une architecture logicielle. L'objectif de ces langages de description est de décrire directement les entités architecturales sans utiliser de notions issues d'un autre paradigme. La notion de connecteur bénéficie d'une attention particulière dans cette communauté par rapport aux autres. En effet, c'est dans les travaux de cette communauté que la parité entre les composants et les connecteurs est la plus établie.

- Architectures et communauté CBSE : La communauté d'ingénierie logicielle à base composants vise à assembler des applications en utilisant des composants logiciels. Ces composants sont une notion très proche des composants architecturaux. Ainsi, les architectures sont ici exprimées en termes de composants logiciels. Par contre, les connecteurs sont encore souvent considérés comme des entités de second plan, se résumant alors à un simple appel de méthode. Cette situation tend, cependant, à s'atténuer suite aux rapprochements avec la communauté ADL.

- Architectures et communauté objet : La structuration de grands systèmes orientés objet est un problème ouvert. La communauté est parvenue à la conclusion que les notions de classes et d'objets sont insuffisantes pour résoudre ce problème. Par conséquence, l'architecture d'un système est souvent, dans ce domaine, exprimée en termes de patrons de conception. Comme pour la communauté CBSE, les connecteurs constituent des éléments secondaires dans l'architecture. Ils représentent des appels de méthodes ou des relations de dépendances entre les patrons de conception.

1.5Conclusion

Nous avons introduit, dans ce chapitre, la terminologie liée aux architectures logicielles et les avantagesobtenus grâce à l'utilisation de cette notion dans la conception des systèmesinformatiques. L'étude de cette terminologie nous a permis de montrer queles définitions de l'architecture et des différents éléments architecturaux sont suffisamment précises

etadmises pour pouvoir en tirer un consensus. L'étude de cette terminologie a permis aussi de mettre en lumière l'évolution de la notiond'architecture à travers les différentes communautés scientifiques qui l'utilisent, tels que les ADLs et les approches indisrielle (CBSE).

Pour asseoir le développement de telles architectures, il est nécessaire de disposer de notations formelles et d'outils d'analyse de spécifications architecturales. La communauté ADL et la communauté CBSE proposent de bonnes réponses. Nous les abordons par la suite dans le troisième chapitre 3 tout en évaluant leur capacité à supporter la spécification aisée et efficace d'IHM.

Vu les avantages des architectures logicielles, nous voulons que cette notion soit utilisé pour la conception de toutes les parties d'un système, même les IHMs. Pour cela, nous présenterons au chapitre 2 une étude assez approfondie sur les techniques de construction et spécification d'IHM. Par la suite, au chapitre 3, nous identifierons les problèmes rendent difficile l'utilisation des approches actuelles d'Architecture logicielle pour la conception d'IHM.

CHAPITRE 2
SPECIFICATION D'INTERFACE HOMME MACHINE

2.1 Introduction

Comme nous l'avons vu dans le chapitre précédent, l'architecture fournit des avantages importants, pour la conception d'un logiciel. Malheureusement, lors de la conception d'un système interactif, la partie métier est la seule à bénéficier de la présence d'une architecture. Quand à la partie IHM, elle est construite en utilisant les approches classiques.

Dans ce chapitre, nous allons présenter l'état de l'art dans le domaine desIHMs, ainsi que les méthodes etoutils de développement d'interfaces graphiques, et nous abordons l'utilisation desarchitectures logicielles, dans le développement de cette famille de produits. Afin de déterminer les composantes principales d'une interface, nous parcourons aussi dans ce chapitre, les modèles d'architecture des systèmes interactifs. Nous terminons ce chapitre par une discussion, afin de déterminer les limites des méthodes actuelles.

2.2 Définitions d'interface homme machine

L'interface homme-machine est un domaine en pleine évolution, à la confluence de deux disciplines : les sciences cognitives et les techniques informatiques. Plusieurs définitions d'une IHM ont été proposées. Nous pouvons en citer deux, issues de chacun des deux domaines**[23]**.

Définition 2.1 : vue par les informaticiens

Une interface homme-machine est un ensemble de dispositifs matériels et logiciels, qui permettent le dialogue entre un utilisateur et le système informatique, en vue de l'accomplissement de certaines tâches.

Cette définition souffre du manque d'intérêt accordé à l'utilisateur qui joue un rôle central dans les IHMs.

Définition 2.2 : vue par les cogniticiens

L'interface homme machine désigne l'ensemble des phénomènes physiques et cognitifs qui participent à la réalisation de tâches informatisées.

Les deux approches ont évolué et ont donné lieu à une nouvelledéfinition de l'interface homme-machine :

Définition 2.3 : Une interface est un ensemble de moyens informatiques (logiciels et matériels), permettant au couple (machine, utilisateur) decommuniquer, en vue de l'accomplissement d'une tâche requisepar l'utilisateur.

2.3 Spécification et construction des systèmes interactifs

Une application interactive ou système interactif est un logiciel qui communique avec un utilisateur. Les échanges s'effectuent grâce à une interface homme-machine.La qualité d'une IHM est primordiale, car plus la distance entre les variables du monde del'application et celles de l'utilisateur est grande, plus le risque d'erreurs et d'incompréhensionest grand **[24]**.

L'analyse de la littérature met en évidence plusieurs types de modèles mettant l'accent surdifférents aspects de la conception des interfaces, et faisant intervenir plusieurs acteurs dans leprocessus de construction des systèmes interactifs :

- Les modèles de tâches: destinés au concepteur d'applications, auquel ils fournissent un schéma général de l'activité de l'utilisateur du système ;

- Les modèles de dialogue : destinés au concepteur de dialogue, pour décrire la structure des échanges entre l'homme et l'application. Ils identifient les objets du dialogue ;

- Les modèles d'architecture : ils fournissent aux réalisateurs une structure générique de l'application, à partir de laquelle il est possible de construire un système interactif particulier.

Dans cette section, nous parcourons brièvement les modèles de tâches, les architectures, ainsi que les techniques etles outils employés pour la conception et le développement des systèmes interactifs. Notons que, nous ne nous sommes pas intéressés aux modèles liés au dialogue.

2.3.1 Les modèles de tâches

L'utilisation de modèles de tâche est motivée par la possibilité qu'elle offre, de décrire les intentions des utilisateurs, et les activités dans lesquelles ces utilisateurs doivent s'engager, pour accomplir leurs buts**[25].**

Les principaux bénéficiaires de l'utilisation des modèles de tâche sont :

- Les concepteurs, car l'utilisation de modèles de tâche fournit une approche structurée et de haut niveau, pour les aspects interactifs et fonctionnels d'un système ;

- Les utilisateurs finaux, car l'utilisation de modèles de tâche permet la construction de systèmes interactifs plus utilisables ;

La modélisation des tâches a bénéficié de la contribution de plusieurs domaines de recherche. Plusieurs définitions du concept de *tâche*ont été proposées, nous pouvons en citer deux, issues de chacun des deux domaines :

- Du domaine des sciences cognitives : une tâche est « ce qui est à faire, ce qui est prescrit » **[26]**. Dans cette définition, Falzon différencie le concept de tâche, du concept d'activité qui est « ce qui est fait, ce qui est mis en jeu par le sujet pour effectuer la tâche »**[26]**. Les chercheurs en sciences cognitivesproposent des modèles, pour l'identification et la caractérisation des tâches (par exemple GTA **[27]** ou MAD **[28]**) ;

- Du domaine informatique : une tâche est « une activité dont l'accomplissement par un utilisateur produit un changement d'état significatif d'un domaine d'activité donné dans un

contexte donné » **[29]**. Les informaticiensproposent des approches et des notations pour la représentation des tâches, et de leurs relations telles que UAN**[30]** et CTT **[31]**.

Etant donné le grand nombre de propositions de modèles de tâches, différentes classifications ont été élaborées. Scapin et Bastien **[32],** etTabary**[33]** classent les modèles de tâches dans trois catégories : les modèles linguistiques, les modèles hiérarchiques orientés tâches et les modèles de connaissance. Dans **[31]** (*cf.* Figure II.1), la classification s'effectue selon le type de formalisme utilisé pour décrire le modèle, l'information qu'il contient et le support qu'il fournit. Balbo **[34]** propose une comparaison pour aider le concepteur à choisir une notation adaptée à ses besoins. A partir de cette étude et de l'utilisation de différents modèles de tâches, une taxonomie est proposée suivant six axes : le but, l'utilisation pour la communication, l'utilisation pour la modélisation, l'adaptabilité, la couverture et l'extensibilité. Limbourg et Vanderdonckt**[35]** proposent une autre classification distinguant : but, discipline, relations conceptuelles et pouvoir d'expression. Finalement, il est à noter que les auteurs **[36]** distinguent deux types de modèles de tâches : les modèles de tâches prescrits et les modèles de tâches effectifs (ou réels). Les modèles de tâches prescrits sont composés des tâches, telles qu'elles sont prévues par le concepteur, alors que les modèles de tâches effectifs, expriment les tâches telles qu'elles sont réellement exécutées par l'utilisateur.

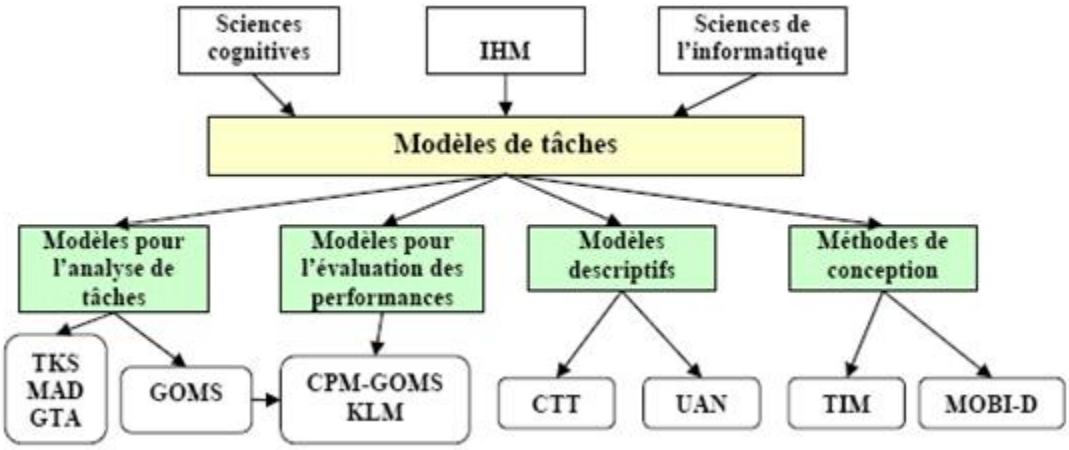


Figure.2.1 : Différentes approches de la modélisation des tâches

Sans prétendre à l'exhaustivité, nous présentons dans la section suivante quelques modèles représentatifs de l'analyse et de la modélisation des tâches.

2.3.1.1 GOMS (Goal, Operator, Method, Selector)

GOMS **[35]** (**G**oals, **O**perators, **M**ethods, **S**elector) (*Buts, Opérateurs, Méthodes, règles de Sélection*) est la première approche sémantique de la conception d'interfaces utilisateur. C'est une méthode basée sur le modèle cognitif « le processeur humain » (the Human Processor **[35]**), elle est décrite par un ensemble de processeurs et de mémoires dont le comportement est défini par quelques règles (c'est une décomposition en trois sous-systèmes : perceptif, moteur et cognitif, qui interagissent).

GOMS (Goal, Operator, Method, Selection) introduit quatre ensembles pour représenter l'activité cognitive, d'un individu engagé dans la réalisation d'une tâche : les Buts, les Opérateurs, les Méthodes et les règles de Sélection.

Un but est une structure symbolique qui définit un état recherché. Il lui est associé un ensemble de méthodes qui toutes conduisent à cet état. En cas d'échec, il constitue un point de reprise, à partir duquel, il est possible d'amorcer d'autres tentatives. Les buts sont organisés de manière hiérarchique : un but complexe est atteint lorsque plusieurs sous-buts sont satisfaits et ceci récursivement jusqu'à atteindre les buts élémentaires. Les buts élémentaires sont réalisés par l'exécution d'une suite d'opérateurs. Les buts forment donc une structure arborescente dont les feuillessont des opérateurs.

Un opérateur est une action élémentaire dont l'exécution provoque un changement d'état (état mental de l'utilisateur et/ou état de l'environnement). Un opérateur se caractérise par des opérandes d'entrée et de sortie, et par le temps nécessaire à son exécution. L'action définie par l'opérateur dépend du niveau de raffinement auquel la modélisation s'effectue. Lorsque l'analyse est fine, l'opérateur reflète des mécanismes psychologiques élémentaires (sensoriels, moteurs ou cognitifs). Lorsqu'elle s'effectue à un niveau d'abstraction élevé, les opérateurs sont des unités d'action spécifiques à l'environnement (par exemple les commandes du système).

Une méthode décrit le procédé qui permet d'atteindre un but (*cf.* figure II.2) **[37]**. Elle s'exprime sous la forme d'une suite conditionnelle de buts et d'opérateurs, où les conditions font référence au contenu de la mémoire à court terme, et à l'état de l'environnement. Les méthodes

représentent un savoir-faire : elles constituent la connaissance procédurale. Elles ne sont pas des plans d'action construits dynamiquement, pendant l'accomplissement de la tâche. Elles sont le résultat de l'expérience acquise.

Une règle de sélection exprime le choix d'une méthode lorsqu'il y a conflit, c'est-à-dire lorsque plusieurs méthodes conduisent au même but.

Parallèlement aux notions de but, opérateur, méthode et règle de sélection, la notion de niveau de modélisation, permet de considérer GOMS comme un générateur d'une famille de modèles. Un niveau de modélisation se définit essentiellement par les temps d'exécution des opérateurs. Dans la mesure du possible, ces temps sont du même ordre de grandeur. Un modèle de niveau "x secondes", peut être raffiné en un modèle de niveau "y secondes" (avec y<x), en convertissant les opérateurs du niveau x en buts, et en produisant des sous-buts jusqu'à ce que ces sous-buts soient atteints par des opérateurs de y secondes. A l'inverse, il est possible de constituer un modèle de niveau x par agrégation d'éléments du niveau y. De toute évidence, GOMS applique à la conception des interfaces, les méthodes de la programmation structurée par raffinement et par abstraction.

```
Selection rule set for goal: select text
   If text-is word, then
         accomplish goal: select word.
   If text-is arbitrary, then
         accomplish goal: select arbitrary text.
Return with goal accomplished.

Method for goal: select word
  Step 1. Locate middle of word.                    (M)
  Step 2. Move cursor to middle of word.            (P)
  Step 3. Verify the the correct word is located.   (M)
  Step 4. Double-click mouse button.                (BB)
  Step 5. Verify that correct text is selected.     (M)
  Step 6. Return with goal accomplished.

Method for goal: select arbitrary text
  Step 1. Locate beginning of text.                          (M)
  Step 2. Move cursor to beginning of text.                  (P)
  Step 3. Verify that the correct beginning is located.      (M)
  Step 4. Press mouse button down.                           (B)
  Step 5. Locate end of text.                                (M)
  Step 6. Move cursor to end of text.                        (P)
  Step 7. Verify that correct text is marked.                (M)
  Step 8. Release mouse button.                              (B)
  Step 9. Verify that the correct text is selected.          (M)
  Step 10. Return with goal accomplished.
```

Figure 2.2 : Exemple de GOMS

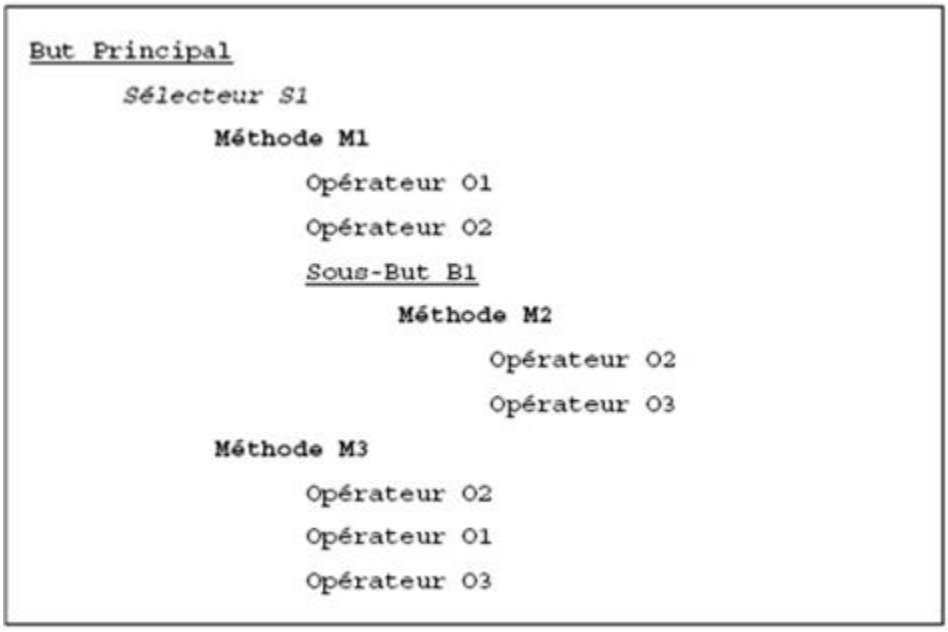


Figure.2.3 : Principe de décomposition de GOMS

Card, Moran et Newell**[38]** définissent quatre niveaux d'analyse : les niveaux tâche, fonctionnel, argument et physique. Le niveau tâche structure l'espace de travail en une hiérarchie de sous-tâches (*cf.* figure.2.3) **[39],** dont la nature dépend uniquement du domaine.Les éléments terminaux de la décomposition sont des tâches conceptuelles élémentaires. L'analyse fonctionnelle modélise les tâches élémentaires, en termes de fonctions du système. A ce niveau de modélisation, l'accomplissement d'une tâche est décrit par une suite de fonctions. Le niveau argument précise pour chaque fonction, sa réalisation par une suite de commandes. A ce niveau de modélisation, l'accomplissement d'une tâche est décrit par une suite de commandes. Le niveau physique décrit en termes d'actions physiques, la spécification des commandes.
Il existe différentes versions de GOMS :

- KLM **[40]** (*Keystroke-Level Model*) (littéralement : modèle au niveau des touches) est une version simplifiée de GOMS, qui n'utilise que les opérateurs concernant les touches, sans utiliser de buts, de méthodes ou de règles de sélection. Ce modèle ne fait que dresser la liste des touches, des mouvements de souris et des clics souris, accomplis pour effectuer une tâche, afin d'évaluer le temps nécessaire à son accomplissement;

- NGOMS **[41]** est une extension de GOMS dans laquelle il a été ajouté plus de rigueur, pour l'identification des composants GOMS, et de l'information comme le nombre d'étapes dans une méthode, la façon dont un but est atteint, ou quelle information est à retenir au cours de l'accomplissement d'une tâche..

- CPM-GOMS **[42]** ajoute (partiellement) à GOMS, la prise en compte des erreurs et la concurrence des tâches, implémentant une méthode de description de chemin critique, basée sur les diagrammes PERT.

1.3.1.2 UAN (User Action Notation)

UAN **[30]**a été créée pour répondre à la nécessité de formaliser la communication, entre les concepteurs d'interface utilisateur, et l'équipe de développement. À cette fin, UAN se concentre sur la fourniture d'un moyen de représenter les spécifications de l'interface utilisateur d'un système interactif, et plus précisément ce que les utilisateurs voient, et comment ils interagissent avec le système.UAN a été conçue pour être utilisée lors de l'analyse des besoins des utilisateurs, ainsi que lors de la conception d'interface utilisateur.

La technique de spécification UAN est un modèle linguistique pour la description de l'IHM. Il repose sur une notation textuelle orientée tâche, et centrée sur l'utilisateur, qui peut ainsi décrire une tâche dans un tableau à trois colonnes :

- Les actions utilisateur qui correspondent aux actions physiques exécutées par l'utilisateur, quand celui-ci agit sur les dispositifs (par exemple, enfoncer le bouton de souris) ;

- Les retours d'informations fournis par le système (élément sélectionné) ;

- L'état de l'interface qui donne la nouvelle situation de l'interface (élément sélectionné).

L'exemple suivant (Figure 2.4) **[30]**décrit la tâche de déplacement d'une icône de fichier dans une interface :

Tâche : Sélectionner Fichier		
Actions Utilisateur	Retour d'information	Etat de l'interface
~ [File icon] Mv [File icon] M^	[File icon]'!	selected=file

Figure.2.4 : Sélectionner une icône de fichier exprimé avec UAN

Dans cet exemple, "~ [File icon]" signifie « déplacer le curseur sur une icône de fichier non sélectionnée ». "Mv" signifie « presser le bouton de la souris ». Quand ces deux actions sont réalisées, l'interface doit faire apparaître en surbrillance l'icône de fichier (file icon'!), et positionner la variable "selected" à la valeur "file". "M^" signifie « relâcher le bouton de la souris et complèter la tâche ».

La capacité de description de la version initiale de UAN était limitée au niveau de la tâche élémentaire. Une extension XUAN fut proposée par Gray **[43]**. Dans XUAN, la structuration des tâches est plus complète, avec des variables locales et externes à la tâche, des pré- et post-conditions et quelques notions temporelles plus fines.

La structuration du modèle de tâches proposé par UAN/XUAN sous forme de tableau donne une certaine compréhension de ce modèle. Cependant, une représentation graphique hiérarchisée, offre une meilleure lisibilité des modèles de tâches. De plus, à notre connaissance, il n'existe pas d'outils qui supportent cette notation.

2.3.1.3 CTT (ConcurTaskTrees)

ConcurTaskTrees**[44]** (dénommée **CTT** par la suite) est une notation basée sur des diagrammes, pour la description des modèles de tâche. Contrairement aux approches précédentes comme UAN **[30]**, CTT offre un nombre important d'opérateurs, qui ont chacun un sens précis, issu principalement de la notation LOTOS (*Language Of Temporal OrderingSpecification*) **[45]**, et qui permettent de décrire de nombreuses relations temporelles, comme la concurrence, l'interruption, la désactivation, l'itération, *etc*. Cela permet au concepteur de représenter de façon concise les évolutions possibles d'une session utilisateur.

Un modèle de tâche, décrit avec ce modèle, se construit en trois phases :

- Une décomposition hiérarchique des tâches représentée sous la forme d'un arbre ;

- Une identification des liens temporels qui unissent ces tâches ;

- Une identification des objets et actions associés.

La figure 2.5 (*cf.* Figure.2.5) montre la notation des opérateurs pouvant lier deux tâches T1 et T2. La priorité des opérateurs est dans un ordre décroissant : opérateur unaire, [], |=|, |||, |[]|, [>, |>, >>, []>>. Les opérateurs temporels peuvent être combinés.

Notation	Signification
T1 [] T2	Le choix
T1 |=| T2	Ordre Indépendant
T1 ||| T2	Imbrication (interleaving)
T1 |[]| T2	Synchronisation
T1 >> T2	Composition séquentielle
T1 []>> T2	Composition séquentielle avec transfert d'information
T1 [> T2	Neutralisation (disabling)
T1 *	Itération infinie (opérateur unaire)
[T1]	Exécution optionnelle (opérateur unaire)
T1 |> T2	Suspendre/Reprendre

Figure.2.5 : Notations CTT des opérateurs entre deux tâches T1 et T2

La Figure.2.6 (*cf.* Figure.2.6)**[27]** représente une modélisation CTT d'une tâche de consultation des informations, concernant un étudiant. Cette tâche se divise en plusieurs sous-

tâches utilisateurs, permettant la formalisation de la demande (saisir_paramètres, envoyer_demande) et d'une tâche machine qui assure l'affichage des résultats.

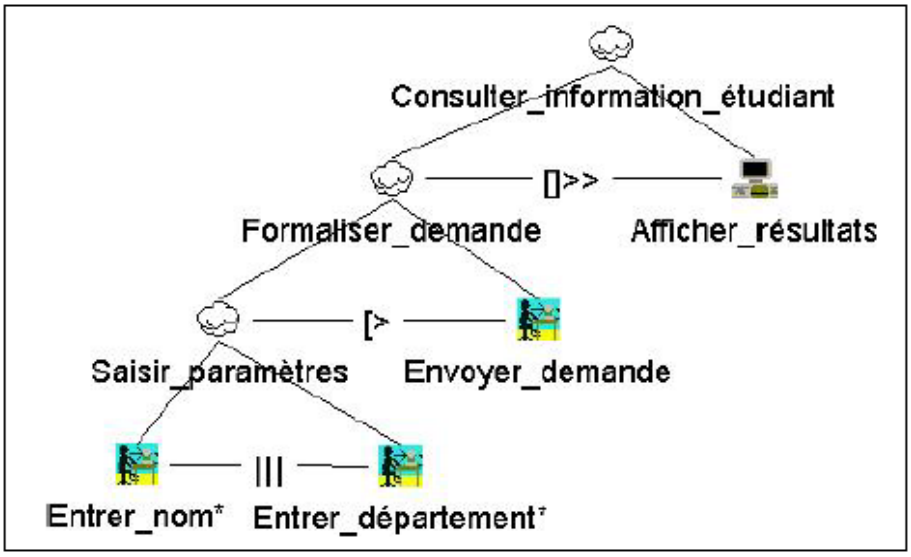


Figure.2.6: Modélisation CTT d'une tâche de consultation des informations d'un étudiant

2.3.2 Les modèles d'architecture d'un système interactif

La recherche en interaction homme-machine, a consacré beaucoup d'énergie à développer des modèles génériques et abstraits, de systèmes interactifs. L'objectif de ces *modèles de référence* est d'obtenir une meilleure compréhension des systèmes interactifs existants, de mettre en place une base commune de communication sur le domaine, et de guider le choix d'une architecture logicielle, lors du développement de nouveaux systèmes interactifs **[46]**.

Les modèles d'architecture logicielle formalisent la façon dont l'utilisateur peut interagir avec le cœur procédural des applications (*noyau fonctionnel),* via une interface.

Depuis les années 1970, différents modèles de ce type ont été proposés. Ces modèles ont été classés en trois catégories : les modèles globaux, les modèles à agents, et les modèles hybrides. Cette section se donne comme objectif d'en donner une liste concise.

2.3.2.1 Les modèles globaux (linguistiques ou centralisés)

Ce sont historiquement, les premiers modèles d'architecture à avoir vu le jour.

2.3.2.1.1Le modèle SEEHEIM

Le modèle de SEEHEIM **[47]** est né d'une réunion de travail, ayant eu lieu dans la ville de SEEHEIM en 1983. Ce modèle préconise la décomposition d'une application interactive en trois modules distincts, permettant de séparer la partie Interface, de la sémantique même de l'application.Ainsi, et comme l'illustre la figure (*cf.*figure.2.7), un système interactif est structuré en trois unités fonctionnelles :

*La Présentation*est chargée d'afficher les informations de l'application à l'utilisateur (vial'écran), et de récupérer ses « entrées » (toutes les saisies faites à partir des dispositifs mis à sa disposition, comme le clavier, la souris, etc.). C'est ici que doivent être prises en compte d'éventuelles règles d'ergonomie. Il est important de noter que cette couche logicielle, est indépendante du reste de l'application. Elle n'a aucune connaissance de la sémantique des actions effectuées par l'utilisateur, et ne connaît que le contrôleur de dialogue qu'elle avertit, lorsqu'un événement se produit à son niveau.

*L'interface du noyau fonctionnel*est une surcouche de l'application. Elle correspond à tout ce que le contrôleur de dialogue connaît de l'application. C'est également elle qui est chargée de transformer les données de la couche de présentation, transmises par le contrôleur de dialogue, en données de l'application.

*Le Contrôleur de Dialogue*fait le lien entre la Présentation et l'Interface de l'application. Il est prévenu par le premier, dès qu'un événement survient. Il peut alors demander à la présentation de lui fournir certaines données, d'en afficher d'autres, etc. Il est également capable de demander à l'Interface de l'application d'effectuer certains traitements.

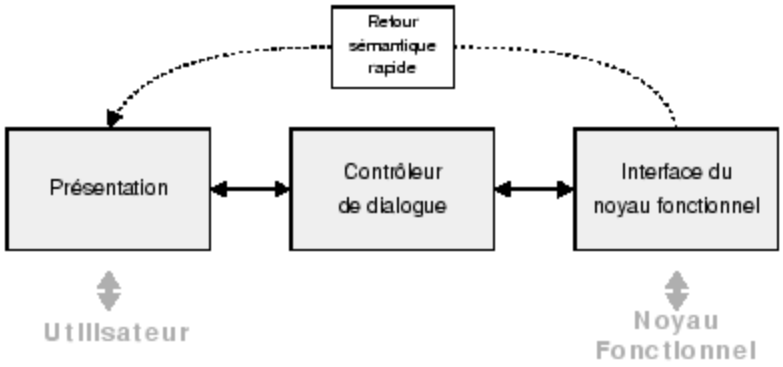


Figure.2.7 : Modèle de Seeheim

Cette architecture prévoit éventuellement, une rétroaction directe du noyau fonctionnel, vers la présentation physique, sans passer par le contrôleur de dialogue ‘’Retour sémantique rapide‘’ **[46]**.

2.3.2.1.2 Le modèle de référence Arch (Seeheim révisé)

A la suite des travaux sur le modèle SEEHEIM, un certains nombres de séminaires réunissant des développeurs d’interfaces ont débouché sur le modèle ARCH **[48]**. Le modèle Arch affine le modèle de Seeheim, en s'inspirant davantage des boîtes à outils graphiques actuelles. Ce modèle incorpore la notion de plate-forme de développement, et décrit la nature des données qui transitent entre les différents composants.

Le modèle s’appuie sur les composants conceptuels du modèle Seeheim (noyau fonctionnel, contrôleur de dialogue, et interaction physique) mais ajoute deux composants intermédiaires. Il reprend le contrôleur de dialogue de Seeheim comme centre de l’arche et il raffine la décomposition des deux autres piliers (*cf.* Figure.2.8).

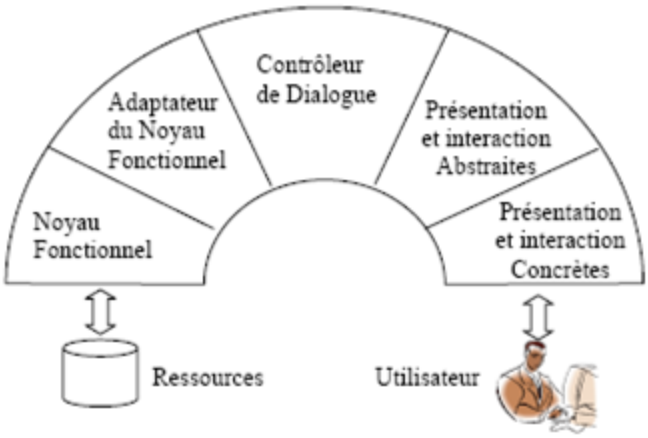


Figure.2.8: Composants du modèle Arch

La séparation entre l'interface Homme/Machine et le contenu fonctionnel (appelé aussi logique applicative), est le plus important aspect exploité par ce modèle. Cette séparation se fait au niveau de l'architecture, car d'après ce modèle, la conception applicative de l'interface, ne doit avoir aucun rapport avec les fonctionnalités présentées par le système, afin de favoriser la réutilisabilité, la modification et l'adaptabilité de point de vue logiciel **[49],** et du point de vue IHM. Pour cela, le modèle présente cinq composants :

Le côté du pilier applicatif distingue 2 niveaux, le noyau fonctionnel et l'adaptateur du noyau fonctionnel :

- Le *noyau fonctionnel*est la partie fonctionnelle qui présente tous les aspects de contrôle, manipulation, et recouvrement des objets du domaine manipulé par la structure du système , tout en restant indépendant de la présentation, et de la projection des structures applicatives auprès de l'utilisateur ;
- *L'adaptateur du noyau fonctionnel* est un composant médiateur, entre le contrôleur de dialogue et le noyau fonctionnel. Il est responsable de la réorganisation des données du domaine, dans des buts de manipulation interactive, ou dedétection des erreurs sémantiques.

Du côté du pilier présentation, il décompose l'interface utilisateur en deux niveaux, le composant abstrait d'interaction, et le composant concret d'interaction.

- Le *composant abstrait d'interaction* est le composant médiateur, entre le noyau fonctionnel et le composant physique d'interaction. Il s'agit d'une boîte à outils indépendante des objets

du système, utilisable par le composant de dialogue, comportant les objets de présentation et d'interaction indépendants de l'interface concrète, et donc des dispositifs d'interaction avec l'utilisateur.

- *Le composant concret d'interaction* est le composant qui implémente les interactions logicielles et matérielles vis-à-vis de l'utilisateur (widget). Il permet de générer les périphériques d'interaction logicielles, et les outils permettant la manipulation des structures de données présentées par le système : les menus, les boutons, les boîtes de dialogue, les icônes, les boîtes à outils.

Enfin, le contrôleur de dialogue est le composant principal qui lie les deux composants,présentation et adaptateur du noyau fonctionnel. Il se charge de manipuler tous les objets conceptuels, les présentations, l'enchaînement des tâches et surtout l'affectation des objets conceptuels aux objets de représentation.

2.3.2.2 Les modèles à agents

De nombreux modèles relèvent de l'approche multi-agent. Dans ce qui suit, nous en donnons les principes généraux des principaux modèles à agents, à savoir MVC et PAC.

2.3.2.2.1 Le MVC (Model - View - Controller)

Le modèle multi-agents MVC **[50] [51]** apparu dans les années 1970, permet de concevoir un logiciel interactif, de manière robuste, en séparant la partie de la logique applicative, de la partie de la logique de présentation.

MVC est le modèle d'architecture du langage Smalltalk**[52],** repris plus récemment par Swing, la boîte à outils du langage Java, puis par la nouvelle version d'ASP.NET appelée ASP.NET MVC **[53]**.

MVC décompose les systèmes interactifs en une hiérarchie d'agents **[46]**. Un agent MVC consiste en un *modèle* muni d'une ou de plusieurs *vues*, et d'un ou de plusieurs *contrôleurs (cf.*Figure.2.9)**[89]**. Les trois modules (modèle, vue et contrôleur) sont indépendants, et peuvent communiquer entre eux par message **[49] [55]**.

Le*Modèle*correspond au noyau fonctionnel d'une application et à ses données métier. Il peut représenter des données brutes (entier, chaîne de caractères) ou des objets ayant un comportement complexe. Le modèle notifie les vues qui lui sont associées, à chaque fois que son état se trouve modifié par le noyau de l'application, ou par ses contrôleurs.

La *Vue*se charge de visualiser les données qui sont retournées par le modèle (celles qui sont affichées à l'écran), pour l'utilisateur final. La vue met à jour la représentation graphique du modèle, à chaque changement d'état de ce dernier ; elle est en général constituée d'objets graphiques. Il faut noter que chaque vue peut avoir un ou plusieurs contrôleurs.

Le *Contrôleur*gère les « entrées » de l'utilisateur, transmises par l'intermédiaire de la vue ; il interprète ces entrées, et peut, en réaction soit modifier la vue associée, soit modifier le modèle.

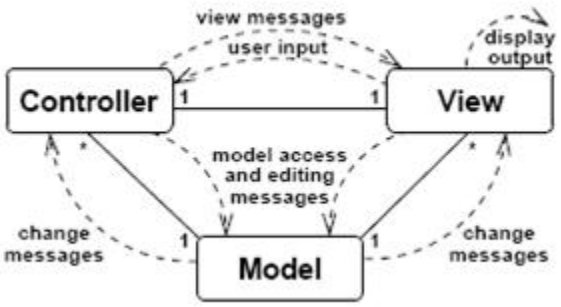


Figure.2.9: Modèle MVC

Cette séparation entre le modèle de données et la visualisation graphique est indispensable surtout si le système doit se présenter sur plusieurs plateformes : web, PDA, téléphone cellulaire, application réseau. Ainsi l'interface utilisateur est adaptable suivant les plateformes d'implémentation en gardant le même noyau fonctionnel.

Il est possible qu'un modèle possède plusieurs de ces couples vue-contrôleur. Ainsi, il est possible que le modèle se trouve modifié, soit par l'un quelconque des contrôleurs qui lui sont associés, soit par le noyau fonctionnel lui-même. Le modèle doit donc alors informer chacune de ses vues, que ses données ont changé, afin qu'elles puissent le cas échéant, mettre à jour leurs informations, par exemple en se redessinant.

2.3.2.2.2 Le modèle PAC (Présentation - Abstraction - Contrôle)

Le modèle PAC a été mis au point en 1987 par Joëlle Coutaz**[23] [55]**. Tout comme MVC, on peut voir PAC comme une architecture SEEHEIM répartie, à la différence que PAC modélise l'application dans sa totalité, là ou MVC ne s'occupe « que » de l'architecture de l'interface.

Le modèle PAC décompose l'application interactive en une hiérarchie d'agents interactifs (*cf.* Figure.2.10)**[56],** structurés en trois facettes : la Présentation, l'Abstraction et le Contrôle.

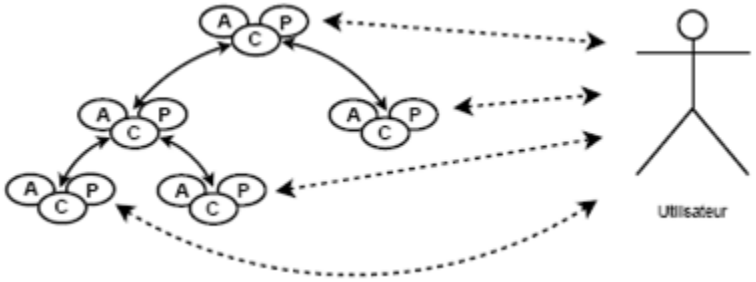


Figure.2.10 : Architecture du modèle PAC

La *Présentation* définit le comportement perceptible de l'agent, auprès d'un agent humain. Elle concerne à la fois les entrées et les sorties, c'est-à-dire les modalités d'actions accessibles à l'utilisateur, et la restitution perceptible. La Présentation interprète les événements résultants des actions physiques, que l'utilisateur applique à l'agent, via des dispositifs d'entrée, et engendre des événements qui traduisent des actions physiques du système, sur les dispositifs de sortie.

L'*Abstraction* avec ses fonctions et ses attributs internes, définit la compétence de l'agent, indépendamment des considérations de présentation. Elle constitue, au sens de Seeheim, le noyau fonctionnel de l'agent.

Quant au *Contrôle*, il a un double rôle : il sert de pont entre les facettes Présentation et Abstraction de l'agent, et gère des relations avec d'autres agents PAC. Tout échange d'informations entre l'Abstraction et la Présentation, s'effectue via le Contrôle. C'est aussi par leurs parties Contrôle que deux agents PAC communiquent.

Les agents PAC peuvent en effet, être organisés en une hiérarchie arborescente **[56]** (*cf.* Figure.2.10), un agent pouvant être composé de plusieurs agents fils, le spécialisant. Un agent parent a la responsabilité des agents fils ; par conséquent le niveau d'abstraction de chaque agent, est proportionnel à son niveau dans la hiérarchie. Un agent peut ainsi en utiliser plusieurs autres : lorsque sa présentation est modifiée, un contrôleur prévient son précédent dans la hiérarchie,

lequel, à son tour, avertit les autres subordonnés, afin qu'ils reflètent le changement dans leur propre présentation.

De manière générale, un événement utilisateur peut avoir un effet sémantique ou non **[52]**. Un événement sans effet sémantique est traité en local par la Présentation, qui produit un retour d'informations perceptible. Les événements à effet de bord sémantique, sont non seulement traités localement par la Présentation, pour les retours d'informations lexicaux, mais, en plus, sont transmis à l'Abstraction, par l'intermédiaire du Contrôle, pour complément de traitements. L'enrichissement sémantique du retour d'informations, peut se poursuivre lorsque l'agent fait appel à d'autres agents de la hiérarchie, y compris le noyau fonctionnel. L'utilisateur interagit avec les facettes de présentation des différents agents PAC, qui constituent un système (*cf.* Figure.2.10) **[56]**.

2.3.2.3 Les modèles hybrides

Dans certains cas, il paraît souhaitable de combiner les modèles à couches, avec les modèles multi-agents, pour essayer de tirer partie de leurs avantages respectifs.

2.3.2.3.1Le modèle PAC-Amodeus

Pour la modélisation des applications multimodales, le modèle hybride PAC-Amodeus**[57]** reprend le découpage en couches d'Arch, tout en exprimant le composant contrôleur de dialogue, sous la forme d'agents PAC (*cf.* Figure.2.11). Il peut ainsi facilement exprimer le parallélisme de contrôle, des différentes modalités. L'adaptateur de domaine et la présentation d'Arch communiquent directement avec chaque agent PAC, à travers son abstraction (pour le premier) et sa présentation (pour le second).

D'une façon plus générale, les composants *Abstraction* et *Présentation*du modèle PAC, peuvent être rapprochés des composants *Adaptateur de domaine* et *Présentation* du modèle Arch. Dans ce cas, ils constituent des adaptateurs combinant une modélisation multi-agents de l'application et l'utilisation de boîtes à outils spécialisées (pour le traitement du noyau fonctionnel ou pour l'interface utilisateur). Ainsi, il est possible de concevoir des logiciels

hétérogènes,utilisant à la fois l'approche multi-agents et l'approche en couches, et faisant appel à différents langages de programmation ou boîtes à outils.

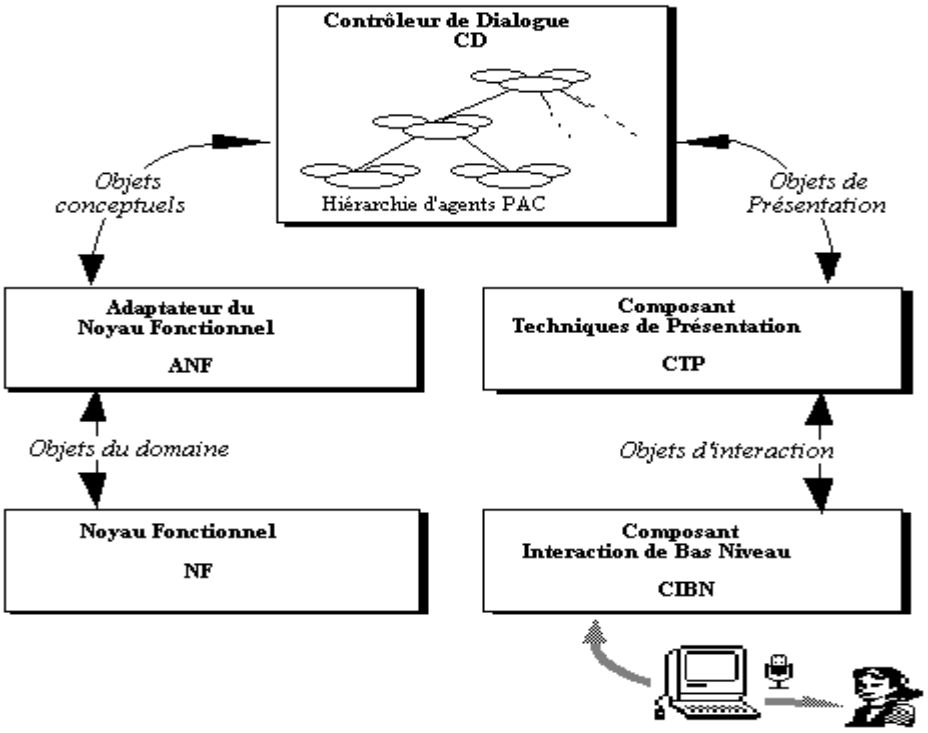


Figure.2.11 : Le modèle hybride PAC-Amodeus

2.3.3 Composants graphiques pour la définition d'IHM

Pour que la charge de travail des développeurs de systèmes interactifs, soit moins importante, un grand nombre de techniques et d'outils d'aide au développement, contenant des composants graphiques ont été proposés. Dans cette section, nous présentons brièvement les outils les plus utilisé actuellement, à savoir les boites à outils (toolkits), les éditeurs graphiques d'interface, et les langages de description d'IHM.

2.3.3.1 Composants graphiques dans les boites à outils

Une application interactive a essentiellement besoin d'animer des formes géométriques à l'écran, etde lire les données, en provenance des dispositifs d'entrée. Le développement d'une

application complète, avec des librairies graphiques et des librairies d'entrée bas-niveau, constitue cependant un travail énorme. C'est pourquoi les *boîtes à outils graphiques* ou*boîtes à outils d'interaction*,proposent des composants interactifs réutilisables (*widgets*), et un mécanismede gestion des entrées, basé sur la notion d'*événements*. La plupart du temps, l'implémentationde la partie interactive d'une application, se résume alors à l'instanciation et au positionnement déclaratifde ces composants « widgets », dont la manipulation est gérée automatiquement par des *comportements* prédéfinis, au sein des widgets.

Une boîte à outilsmet à disposition du programmeur d'interfaces, une collection detechniques d'interaction, sous forme de composants logiciels réutilisables, de petite taille. Ellescorrespondent à une façon de manipuler les périphériques physiques (souris, tablette àdigitaliser, clavier, écran...), pour faire le choix d'une commande (exemple : menu), d'uneacquisition de valeur (exemple : *scrollbars*), accéder à des fonctionnalités, effectuer unparamétrage (exemple : boutons), ou réaliser des affichages graphiques.

Les boîtes à outils fournissent un ensemble de modules dotés de fonctions communes (API :*Application Programming Interface*), produisant automatiquement des applications. Unegrande variété de boîtes à outils est proposée au développeur, nous citons à titre d'exemple AWT, Swing et HTML Kit.

L'Abstract WindowToolkit (AWT) **[58]** est une boîte à outils qui a été proposée pour le langage JAVA.La particularité de l'AWT, est que Java fait appel au système d'exploitation sous-jacent, pourafficher les composants graphiques. Pour cette raison, l'affichage de l'interface utilisateurd'une application peut diverger sensiblement: chaque système d'exploitation dessine à sa manière un bouton. L'AWT garantira que la fonctionnalité recherchée sera dans tous les casfournie, mais elle sera présentée différemment.

Comme le langage Java se veut être indépendant de la plate-forme utilisée, une autre API a été définie dans les versions qui suivent. Swing **[59]**est une boîte à outils proposée encore une fois pour le langage JAVA ; elle offre la possibilité de créer des interfaces graphiques identiques, quel que soit le système d'exploitation sous-jacent, au prix de performances moindres qu'en utilisant AWT.

HTML Kit **[60]** est un autre exemple de boîte à outils contenant des composants graphiques. Elle est dédiée pour tous ceux qui font du développement de pages web, peu importe les technologies utilisées (html, whtml, xml, css, javascript, php...).

2.3.3.2 Composants graphiques dans les éditeurs graphiques d'interfaces

Les éditeurs graphiques d'interfacesou GUIBuilders (*Graphic User Interface Builders*), sont des logiciels donnant aux concepteurs, la possibilité de créer la couche de présentation des interfaces graphiques. Cette création d'interface par les moyens visuels, exclut toute formede programmation graphique complexe par leur utilisateur. La plupart des éditeurs graphiques actuellement disponibles, fournissent des composants graphiques, comme les menus, les boîtes de dialogues, la gestion desfenêtres et de boutons, prêts à être utilisés. En revanche, ils offrent peu de supports, pour desinterfaces représentant les données de manière spécifique au domaine, et utilisant des stylesd'interaction variés.

Des éditeurs graphiques sont proposés par la plupart des boîtes à outils. On peut citer à titred'exemple: Visual C++ et Visual Basic de Microsoft pour les MFC, JBuilder pour Java deBorland.

2.3.3.3 Composants graphiques dans les UIDL

Un *langage de description d'IHM* (UIDL : *User Interfacedescription Language*) est un langage de haut niveau, dont le but est de décrire les interfaces, et plus précisément la composition des composants graphiques. L'atout majeur des UIDL, dans leur ensemble, est de rendre la définitiondes interfaces indépendantes de la plate-forme d'exécution, et ceci en fournissant une description d'interface abstraite.

Ces cinq dernières années, le nombre de langages de description des interfaces utilisateur ne cesse d'augmenter. Les efforts dans ce domaine se concentrentessentiellement autour des langages basés sur XML **[61]**. En effet, en plus del'interopérabilité apportée par ce langage, la flexibilité de XML donne la possibilité de créer sapropre grammaire et sa propre interprétation.

Dans ce qui suit, nous proposons de parcourir rapidement les principaux langages qui sont proposés dans la littérature, pour la description d'interfaces abstraites.

2.3.3.3.1 XForms

XForms**[62]** est un exemple de l'évolution des langages basés XML et deleur exploitation, pour la description des interfaces utilisateur génériques.Bien qu'initialement conçu pour s'intégrer à XHTML **[63]**, XForms n'estplus restreint à ce langage et peut s'intégrer à n'importe quel langage de balisage compatibleXML.

XForms offre principalement des fonctionnalités de formulaire, comme lescalculs de données. L'un des aspects essentiels de ce langage, est qu'il a pour objectif deséparer le formulaire de la présentation en trois parties : le modèle d'informations, le traitement d'informations et l'interface utilisateur. Cette séparation facilite la modélisation des données, sans se préoccuper de savoir, comment elles seront utilisées ou présentées.

XForms offre aujourd'hui un degré d'abstraction suffisant, pour la description des interfacesutilisateurs, indépendamment du support. Comme les éléments de l'interface sont définis demanière abstraite, c'est la plate-forme cible qui va choisir la manière dont elle va présenter leformulaire. Par exemple, <xform:select1> permet de définir un contrôleur où l'utilisateur doitchoisir un seul élément parmi une liste. Selon les capacités de la plate-forme, le navigateuraffichera cette liste sous forme de boutons radio, de liste déroulante ou autre.

Cependant, XForms se limite à la modélisation des formulaires, et il communique pourl'instant, exclusivement avec les navigateurs compatibles XML.

2.3.3.3.2 UIML (User Interface MarkupLanguage)

Le User Interface MarkupLanguage (UIML) **[64]** est l'un des premiers langages basés XML, pour décrire l'interfaceutilisateur. UIML est également le plus connu et le plus répandu.

UIML est un langage générique qui cherche à masquer la diversité des plates-formes, et deslangages de développement. Il définit des interfaces, sans pour autant les associer à un formatfigé. Une description UIML est composée de quatre parties (*cf.* Figure.2.12) **[29]**:

- la partie « *head*» qui donne une méta-description du document ;

-la partie optionnelle « *template*» qui marque des fragments d'UIML réutilisable ;

-la partie « *interface* » qui décrit les parties de l'interface utilisateur, à savoir : la structure,le contenu, le style et les comportements ;

- la partie « *peers*» qui décrit la correspondance, entre d'un côté, les éléments du document UIML, et de l'autre, les composants de l'interface utilisateur finale et la logiqued'application.

Les éléments de UIML spécifient les informations nécessaires pour la programmation, en séparant ce qui est générique à l'interface, de ce qui est spécifique aux supports. Aujourd'hui, une interface développée en UIML peut être interprétée via un moteur de rendu en Java, ou HTML.

Les transformations des composants abstraits en composants concrets, sont spécifiées dans la partie *peers* du document UIML. Cependant, cette partie du documentdoit être réécrite pour chaque plate-forme, ce qui nuit complètement à la notion de généricitédu langage. Une même description d'interface n'est utilisable que pour des supports proches,autorisant une même structure.

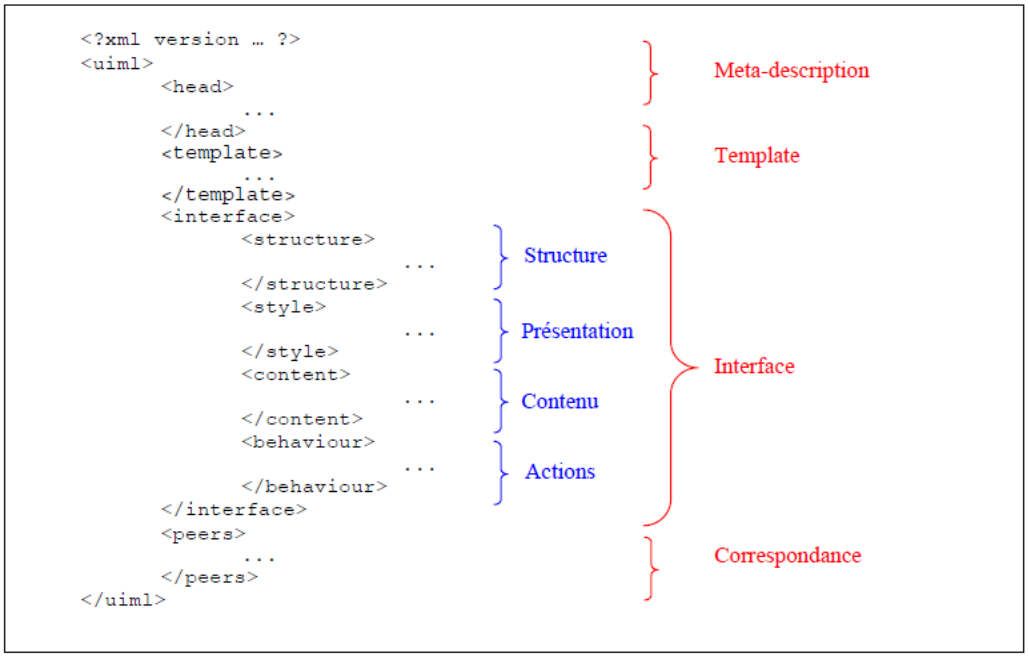


Figure.2.12 : Composition d'un fichier UIML.

2.3.3.3.3 AUIML (Abstract User Interface MarkupLanguage)

Le langage de balisage des interfaces utilisateur abstraites AUIML **[65]**est né de l'ambitieux projet d'IBM, qui avait pour but de définir unlangage de description d'interface utilisateur, indépendant de la plateforme.La particularité d'AUIML est de permettre de définir l'intention (ou lebut) d'une interaction, plutôt que de se concentrer sur son apparence. Ainsi,dans AUIML, l'accent est mis davantage sur la sémantique que sur la forme.Cette représentation étant sémantique, fait qu'une même description peutêtre utilisée sur plusieurs plateformes.

Une interface utilisateur AUIML est décrite en termes d'objets manipulés, d'éléments d'interaction (un modèle de présentation spéciale : liste à choix multiples, groupe, table, arbre,...), et d'actionsqui permettent de décrire un micro-dialogue, pour gérer les évènements entrel'interface et les données. En utilisant le modèle de présentation, l'utilisateurpeut choisir le niveau d'abstraction dans lequel il souhaite travailler. Un designer peut ainsi définir de manière précise ce qui doit être affiché, ou fourniruniquement le style global de l'interaction, laissant au moteur de rendu lesoin de faire le bon choix.

Le langage AUIML est dirigé par les données ; il définit le type d'information qui doit être présenté comme étant une balise, et fournit une correspondance pour chaque type utilisé (*cf.* Figure.2.13) **[29]**. Lestypes disposent également d'un lien vers le modèle de données (dans ce cas un ensembled'objets Java).

```
<GROUP NAME="NomDeLaPersonne">
<CAPTION><META-TEXT>Nom complet de la personne </META-TEXT></CAPTION>
<CHOICE NAME="TitreDeLaPersonne" SELECTION-POLICY="SINGLE">
<CAPTION><META-TEXT>Titre</META-TEXT></CAPTION>
<HINT>
<META-TEXT> Un ensemble de titres valables <META-TEXT>
</HINT>
<STRING NAME="MR">
<CAPTION><META-TEXT>Mr.</META-TEXT></CAPTION>
</STRING>
<STRING NAME="MELLE">
<CAPTION><META-TEXT>Melle.</META-TEXT></CAPTION>
</STRING>
</CHOICE>
</GROUP>
```

Figure.2.13 : Exemple de description d'une intention dans AUIML.

Une grande partie des informations concernant AUIML demeurent confidentielles, car celui-ci est destiné principalement à un usage interne à IBM.Le moteur de rendu, par exemple, reste confidentiel.

2.4 Discussion

Les modèles de tâches présentés apportent une contribution importante pour la conceptiond'un système interactif. Le système doit s'appuyer sur la connaissance que l'utilisateur a de satâche. Le modèle de tâche utilisateur s'utilise souvent pour la spécification des besoins dans le domaine des IHMs, et est insuffisant pour la spécification d'interfaces homme machine.

Avant de proposer un modèle de composants pour la spécification d'une IHM, il était important pour nous, de déterminer les composantes principales de toute interface. L'analyse des modèles d'architecture nous a révélé, que chaque modèle se propose d'identifier les éléments significatifs, qui entrent dans la composition de la plupart des systèmes interactifs, ainsi que les relations qui les lient. Bien que les terminologies employées diffèrent sensiblement, on retrouve souvent d'un modèle à l'autre, les mêmes principes de base.

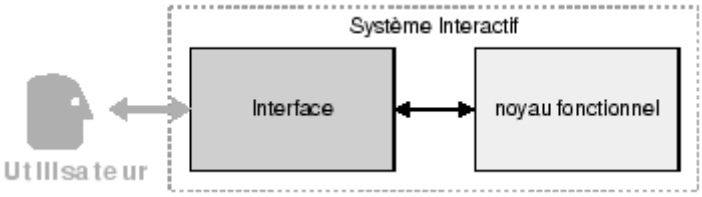


Figure.2.14: Modèle primitif d'un système interactif

Tous les modèles partent du principe qu'un système interactif comporte une partie « interface » appelée aussi « Composant de Présentation » **[66],** qui est identifiée comme entité externe au reste de l'application, et une partie « application pure » (*cf.*figure.2.14) dénommée *noyau fonctionnel ou noyau sémantique* **[67]**, telle que tout ce qui s'y réfère appartient au *domaine*. Le noyau fonctionnel est considéré comme préexistant, et les modèles de systèmes interactifs décrivent essentiellement la partie interface, ainsi que ses relations avec les objets du domaine.

La partie métier et la partie IHM des systèmes interactifs actuels ne sont pas conçues de la même manière ; la partie métier est conçue selon les modèles de composants présentés dans le premier chapitre, tandis que la méthode utilisée pour construire des IHMs, est de développer les IHMs en utilisant des boîtes à outils standards, par exemple, Swing ou HTML. Mais cela conduit à deux difficultés principales :

- Le premier problème est que ces IHMs sont moins portables que les composants qu'ils représentent. Contrairement à la logique métier qui est unique, une version de l'IHM doit êtredéveloppée pour chaque nouveau média visé : une version logicielle, web, PDA, … Cela multiplie letravail de développement par le nombre de versions à réaliser. De plus, toute modification devra êtrerépercutée sur chacune de ces IHMs. Le coût en maintenance et en développement est donc assezlourd.

- Le deuxième problème est que ces IHMs sont peu réutilisables, cela demande un travailparticulier pour rendre une IHM réutilisable. Il est difficile de réutiliser une IHM qui n'a pas été prévue pour.

Ces problèmes ne disposent actuellement que de solutions partielles, les UIDL proposent de rendre la définitiondes interfaces indépendantes de la plate-forme d'exécution, et donc de rendre celles-ci portables. Cependant, ces langages émergent, sans pour autant apporter des solutions aux différents problèmes. En effet, les UIDL proposés ne permettent pas de lier les composants métier aux composants graphiques correspondants, et donc la cohérence entre la partie métier et la partie IHM n'est pas assurée.De plus, ils ne permettent pas de décrire les liens de données et d'évènements qui existent entre le noyau fonctionnel et l'IHM. Enfin, ces langages sont surtout destinés à un usage orienté web.

2.5 Conclusion

Nous avons présenté dans ce chapitre, une rapide synthèse des solutions utilisées pour la spécification d'IHM et de systèmes interactifs. Ces solutions sont en effet toujours utilisées, mais présentent des inconvénients majeurs que l'on a déjà abordés.

Si l'adage « write once runeverywhere » a déjà trouvé sa place dans la partie métier d'une application, il est encore à l'étude, en ce qui concerne le développement des interfaces homme-machine. Il nous semble essentiel,que les applications soient construites par assemblage de composants, selon une approche architecture logicielle. Cet assemblage doit comprendre les composants métier, ainsi que leurs composants d'IHM associés.

CHAPITRE 3
SPECIFICATION DES IHMs EN ARCHITECTURE LOGICIELLE : UN ETAT DE L'ART

3.1 Introduction

Dans ce chapitre, nous allons effectuer un état de l'art, sur la prise en compte des interfaces homme machine, dans la définition d'une architecture logicielle, à base de composants. Pour cela, nous commençons par la détermination des caractéristiques fondamentales, d'un modèle de composants, qui supporterait efficacement la spécification des IHMs, que ce soit au niveau de la vue *présentation*, ou de la vue *architecture* (assemblage de composants à l'aide de connecteurs) de l'IHM. Cette action se basera sur les résultats du deuxième chapitre, renforcés par une analyse des IHMs dans le monde objet. Le monde objet étant actuellement l'espace ou la spécification d'IHMs, a atteint un haut degré de maturité. Les caractéristiques déterminées seront ensuite utilisées dans ce chapitre, pour l'évaluation des divers travaux autour des langages de description d'architecture, des modèles de composants académiques, et des modèles de composants industriels.

3.2 Eléments de base pour la construction des IHMs

L'objectif de notre travail est de permettre à l'architecture logicielle, d'arriver a un degré de maturité aussi élevé, que le degré de maturité atteint par la spécification des IHMs dans l'orienté objet, si ce n'est plus. En architecture logicielle, l'élément de base pour la construction d'IHM est le composant. Les connecteurs pourraient par la suite être utilisés pour atteindre des objectifs, qu'il n'est pas facile d'atteindre en orienté objet. A titre d'exemple, grâce au concept de connecteur, un composant d'IHM pourrait interagir avec un autre composant métier, ou d'IHM, déployé dans d'autres environnements distants. Le modèle de composants adapté à la spécification d'IHM, devrait permettre au moins le support des diverses actions de construction d'IHM, dans le contexte d'un IDE orienté objet, tels que Delphi, JavaBuilder, Eclipse ou NetBeans.

En orienté objet, les composants d'IHM, que nous appellerons des composant visuels, sont des classes d'objet. Le code de la classe est incorporé dans le programme, dès le placementde ce composant visuel, sur une des vues, de la partie présentation d'une application. A titre d'exemple, si on considère la conception d'une IHM, dans le contexte d'un IDE (*IntegratedDevelopmentEnvironment*) supportant le langage Java, le placement d'un bouton dans la vue présentation, entrainera l'incorporation dans le programme du code de la classe JButton, du package javax.swing.

Les composants visuels sont définis, en se basant sur un modèle de référence. A titre d'exemple, dans Java, JavaBean représente le modèle de référence de tous les composants visuels, du package javax.swing.

Souvent les composants visuels réagissent à des événements, initiés par une action sur la souris ou sur le clavier, ou par une action programmée. La réaction du composant visuel (donc de son code), entraine le lancement d'une opération, qui s'est au préalable inscrite au niveau du composant visuel, comme intéressée par un évènement bien précis. L'opération appartient à ce qui est communément appelé, un *écouteur d'évènement* sur un composant.

Les évènements sont par leur nature asynchrone. De plus, le nombre d'évènements auquel peut réagir un composant visuel, est bien connu. Cette connaissance s'est d'ailleurs traduite, par une sorte de normalisation, du nom des opérations d'écoute. Ces dernières sont souvent définies dans une interface, que l'écouteur devra implémenter. A titre d'exemple, une des interfaces d'écoute d'évènement de la souris, s'appelle *MouseListener* dans Java. Les operations de cette interface sont:mouseClicked(MouseEvent), mouseEntered(MouseEvent), mouseExited(MouseEvent), mousePressed(MouseEvent), et mouseReleased(MouseEvent).

Une IHM peut être composée de plusieurs fenêtres indépendantes, chacune composée de plusieurs composant visuels (sous fenêtre, conteneur doté d'un placeur de composants visuels, les boutons, les zones de texte, etc..). Lorsqu'un composant visuel subit une opération, qui nécessité le rafraichissement du dessin de son contenu, le composant composite initie un évènement programmé, qui consiste à redessiner tous les composant visuels qu'il contient.

Lors de la conception d'une IHM dans le contexte d'un IDE, nous distinguons deux vues relative à l'IHM: La vue *présentation* : qui est construite interactivement, par la sélection des composants visuels, et leur placement dans la fenêtre. La vue *code* : qui est parfois manipulée par le concepteur, pour lier la production d'un évènement, à un traitement bien précis.

La synchronisation entre la vue présentation, et la vue du code est nécessaire dans les IDEs. La modification d'une propriété du composant visuel, au niveau de la vue présentation (taille, titre, couleur etc..), est automatiquement reporté dans le code source de l'application. De même, toute modification au niveau du code, est répercutée au niveau de la vue présentation.

3.3 Les caractéristiques d'un modèle de composants adapté à la construction d'IHM

L'étude précédente et celle du chapitre 2, révèlent que tout modèle de composants, adapté à la construction des IHMs, doit supporter les différentes caractéristiques des ces dernières. Nous organisons ces caractéristiques en trois groupes :

3.3.1 Les caractéristiques liées à la structure de l'IHM

Un modèle de composants spécifiques aux IHMs, doit permettre la spécification, de tout élément faisant partie de ces dernières, ainsi que les relations existantes entre eux. Nous citons donc, les caractéristiques suivantes :

- La construction d'IHM se fait par la définition d'un ensemble de composants visuels. Chaque composant visuel est construit, en se référant à un modèle de référence ;

- Un composant visuel peut être simple ou composite. Dans ce dernier cas, il comprend plusieurs autres composants visuels.

- Un composant visuel possède des attributs, qui peuvent être modifiés.

3.3.2 Les caractéristiques liées au comportement de l'IHM

Un modèle de composants efficace, doit prendre en charge le comportement, et le traitement des composants visuels, comme dans le cas de l'acquisition de données, d'affichage de résultats, ou de lancement d'évènement. Un évènement estproduit soit par une action de l'utilisateur (cliquer sur un bouton, fermer une fenêtre, ...),soit par une directivevenant d'un autre composant. Les caractéristiques liées au comportement sont :

- Chaque composant visuel réagit à un ensemble fixe d'évènements ;

- Les évènements sont par nature asynchrone ;

- Le composant doit permettre la spécification d'un intérêt, pour un ou plusieurs évènements ;
- Le composant doit réagir aux évènements programmés, ou issus de dispositifs physiques, par lancement de l'opération inscrite comme intéressée par un ou plusieurs évènements ;

3.3.3 Les caractéristiques liées aux liens entre composant métier et composant d'IHM

L'utilisateur d'une application ou d'un site web, fait appel aux services de ses derniers, en manipulant l'IHM, elle est donc liée aux différents services métier d'une application. Il est important pour nous, de déterminer les liens qui existent entre un composant métier, et un composant visuel, ceci nous amène à citer la caractéristique suivante :

- Un composant visuel peut être lié à un ou plusieurs composants métier.

Dans la suite, nous passerons en revue les diverses propositions autour des langages de description d'architecture, des modèles de composant académiques, et des modèles de composants industriels. Ces travaux se basent essentiellement sur un modèle de composants et de connecteurs.

3.4 Les langages de description d'architecture

Les premiers travaux réalisés pour décrire les architectures logicielles,se sont concrétisé par l'apparition des langages de description d'architecture (ADLs). En accord avec la notion d'architecture logicielle, l'idée est de fournir une structure de haut niveau de l'application, plutôt que l'implantation dans un code source spécifique **[68]**. Il faut préciser qu'il n'existe pas une définition unique des ADLs, mais en général on accepte qu'un ADL fournisse un modèle explicite de composants, de connecteurs et de leur configuration, utilisé pour décrire la structure

d'un système comme un assemblage d'éléments logiciels. Cela est illustré par le diagramme de la Figure 3.1 (*cf.* Figure 3.1), où l'on voit un client et un serveur reliés par un appel distant de procédure (RemoteProcedure Call) **[69]**.

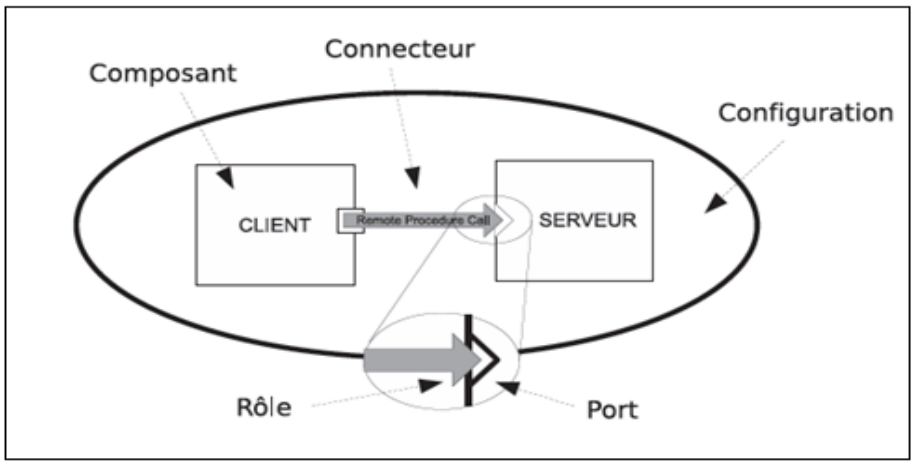


Figure 3.1 : Assemblage d'éléments architecturaux pouvant être décrit à l'aide d'un ADL

Durant la dernière décennie, une douzaine d'ADLs différents sont apparus. Chacun d'eux vise à décrire une configuration architecturale, mais selon un angle de vue différent. En outre, chacun de ces ADLs possède des règles de syntaxe très spécifiques et adaptées à la description de certaines propriétés de l'architecture.

3.4.1 L'ADL Darwin

3.4.1.1 Présentation du modèle

Le langage Darwin **[70][71]** est considéré comme un langage de description d'architecture, bien que celui-ci soit souvent appelé langage de configuration. Un langage de configuration favorise la description de la configuration d'une application, c'est-à-dire la description des interactions entre composants.La description d'un composant au niveau du langage Darwin, permet de créer de multiples instances d'un composant lors de l'exécution. Ainsi, ce type de langage se centre sur la description de la configuration, et sur l'expression du comportement d'une application, plutôt que sur la description structurelle de l'architecture d'un système, comme le font de nombreux ADLs.

Le concept principal de Darwin est le composant. Un composant est décrit par une interface, qui contient les services fournis et requis. Ces services s'apparentent plus aux entrées, et sorties de flots de communication, qu'à la notion de fonction. Deux types de composants existent :

- les primitifs intègrent du code logiciel, leur granularité est établie au niveau d'un processus,
- les composites sont des interconnexions de composants, ils représentent des unités de configuration.

La sémantique associée à un composant est celle du processus. Ainsi, une instance de composant, correspond à un processus créé. Darwin permet de fixer le nombre d'instances d'un composant, lors de son initialisation. Les services requis ou fournis correspondent à des types d'objets de communication, que le composant utilise pour respectivement communiquer avec un autre composant, ou recevoir une communication d'un autre composant. Ainsi, les services n'ont pas de connotation fonctionnelle. Ils décrivent les types d'objets de communication utilisés, ou autorisés à appeler une fonction du composant. Ces types d'objets sont définis par le support d'exécution réparti, appelé Regis**[70],** et sont limités. Ainsi, à l'exécution, un composant Darwin est un processus qui communique avec d'autres composants, grâce à des objets de communication créés et gérés par le système d'exécution Regis.

Parmi les types d'objet, le port est le plus courant : il s'agit d'un objet envoyant des requêtes de manière synchrone, ou asynchrone entre composants répartis ou non.

Les composites sont des entités de configuration. Ils contiennent les descriptions des interconnexions de l'application. Une application est décrite comme un composant composite. Deux constructions syntaxiques permettent de définir des schémas d'instanciation :

– l'opérateur *inst*, qui déclare une instance de composant sur un site particulier. Cet opérateur permet de décrire la phase d'initialisation ;

– l'opérateur *bind*, qui relie un port requis d'un composant, à un port fourni d'un autre composant. Cet opérateur permet de décrire les liens entre composants, au moment de

l'exécution. Cet opérateur peut servir à lier un port d'un composant composite, avec un port d'un composant primitif faisant partie du composite.

L'exemple de la figure 3.2(*cf.* Figure 3.2) **[72]**illustre un composant de type *pipeline*, composé par une liste d'instances de composant *filter*. L'entrée input de chaque instance est connectée à la sortie output de son prédécesseur. L'itérateur*forall* permet d'instancier chacune des instances (inst), et de les connecter (bind). Lorsqu'une interface n'est pas satisfaite à l'intérieur du composant, le composant peut l'exposer comme un besoin à satisfaire par l'extérieur, c'est-à-dire par d'autres composants compatibles. Ceci est le cas, par exemple dans : F[n-1].next – output.

```
componentpipeline (int n) {

// Interface
provideentrée;
requiresortie;

// Implantation
arrayF[n] : filtre;
// Définition d'un ensemble d'instances de filtre
forallk: 0..n-1 {
instF[k]; // création d'une instance de filtre
bindF[k].sortie -- sortie;
// Lien avec le composant pipeline
whenk<n-1
bindF[k].droite -- F[k+1].gauche;
// Lien des composants entre eux
}
bindentrée -- F[0].gauche;
F[n-1].entrée -- sortie;
}
```

Figure 3.2 : Architecture pipeline à base d'un composant composite dans Darwin

3.4.1.2 Evaluation de Darwin

Le langage Darwin fournit une syntaxe riche, permettant de décrire les interactions entre composants. Ce langage considère un élément de l'architecture (le composant), comme une entité pouvant être instanciée. Ceci a pour conséquence de spécifier de manière plus précise, les interactions entre composants visuels. Cependant, Darwin est limité par le fait qu'un composant, n'est associé qu'à une seule sémantique, qui est le processus. Ainsi, un composant ne permet pas

d'exprimer un autre élément, tel qu'un composant visuel, et donc ne peut pas exprimer l'architecture d'une IHM. De plus, les évènements spécifiques au clavier et souris ne peuvent pas être associés à un composant, et le déclenchement de services lors de la réception d'un évènement d'un autre composant est très limité.

3.4.2 L'ADL Rapide

3.4.2.1 Présentation du modèle

Rapide **[18]**, est un langage de description d'architecture, dont le but est de vérifier, par la simulation, la validité d'une architecture logicielle donnée.

Avec le langage Rapide, une application est construite à partir de composants, communiquant par échange de messages, ou d'événements. Rapide fournit également un environnement, composé d'un simulateur permettant de vérifier la validité de l'architecture. Les concepts de base du langage Rapide sont les événements, les composants, et l'architecture.

L'événement est une information transmise. Il permet de construire des expressions appelées *event patterns*, qui caractérisent les événements circulant entre composants. La construction de ces expressions se fait avec l'utilisation d'opérateurs, qui définissent les dépendances entre événements. Parmi ces opérateurs, on trouve l'opérateur de dépendance causale (A → B si l'événement B dépend causalement de A), l'opérateur d'indépendance A||B (A est indépendant de B), l'opérateur de différence (A~B si A et B sont différents), et l'opérateur de simultanéité (A and B si A et B sont vérifiés). Ainsi, l'événement correspond à une information, permettant de spécifier le comportement d'une application.

Le composant est défini par une interface. Cette dernière est constituée d'un ensemble de services fournis et d'un ensemble de services requis. Les services sont de trois types :

- Les services *Provides*: fournis par le composant,et appelés de manière synchrone par d'autres composants ;

- Les services *Requires*: demandés par le composant, et appelés de manière synchrone ;

- Les *Actions*:correspondent à des appels asynchrones entre composants. Deux types d'actions existent : les actions *in,* et *out* qui sont des événements acceptés, et envoyés par un composant.

L'interface contient également, une section de description du comportement (clause *behavior*) du composant. Cette dernière correspond au fonctionnement observable du composant comme, par exemple, l'ordonnancement des événements ou des appels aux services. Ainsi, l'environnement Rapide peut simuler le fonctionnement de l'application.
De plus, Rapide permet également de spécifier des contraintes (clause *constraint*), qui sont des patrons d'événements, qui doivent ou non se produire pour un composant lors de son exécution. Par exemple, une contrainte peut fixer un ordre obligatoire, pour une séquence d'événements d'un composant. En général, ces contraintes permettent de spécifier des restrictions, sur le comportement des composants.

L'architecturecontient la déclaration des instances de composants, et les règles de connexions entre ces instances. Toutes les instances sont déclarées sous forme de variables. La règle d'interconnexion est composée de deux parties. La première est la partie gauche, elle contient une expression d'événements qui doit être vérifiée, la seconde est la partie droite, elle contient également une expression d'événements qui doivent être déclenchés, après la vérification de l'expression de la partie de gauche. Les contraintes (*clause constraint*) peuvent être utilisées pour décrire l'architecture. Elles permettent de restreindre le comportement de l'architecture, en définissant des patrons d'événements à appliquer pour certaines connexions entre composants. Les parties gauches et droites peuvent être connectées par trois types d'opérateurs :

- L'opérateur « To » : connecte deux expressions d'événements simples. Il ne peut y avoir qu'un événement possible vers un composant. Si la partie gauche est vérifiée, alors l'expression de la partie droite, permet le déclenchement de l'événement, vers l'unique composant désigné par cette expression.

- L'opérateur de diffusion « ||> » : connecte deux expressions quelconques. Dès que la partie gauche est vérifiée, tous les événements contenus dans la partie droite sont déclenchés. Ils sont envoyés vers l'ensemble des destinataires désignés dans cette expression. L'ordre d'évaluation de cette règle de connexion est quelconque.

- L'opérateur pipeline « => » : est identique au précédent, mais l'ordre d'´evaluation des règles est contrôlé. Un déclenchement de cette règle, est causalement dépendant des déclenchements antérieurs de cette même règle.

Un exemple d'architecture producteur/consommateur est décrit par la spécification suivante de la figure 3.3 (*cf.* Figure 3.3).

```
type Producer (Max : Positive ) is  interface
action  out   Send (N: Integer ) ;
action   in   Reply (N : Integer ) ;
behavior
        Start =>send (0);
        (?X  in  Integer ) Reply (?X) where ?X<Max =>Send
(?X+1);
end Producer ;

type  Consumer is interface
        action  in  Receive (N: Integer ) ;
        action  outAck(N : Integer ) ;
        behavior
                (?X in Integer ) Receive (?X)  =>Ack (?X) ;
end Consumer

architectureProdCon ( )   return   SomeType   is
        Prod  :  Producer (100) ;    Cons  :  Consumer ;
        connect
        (?n  in Integer )    Prod.Send (?n) =>Cons.Receive (?n ) ;
        Cons.Ack(?n)  =>Prod.Reply (?n ) ;
end architecture ProdCon ;
```

Figure.3.3 : Exemple d'architecture producteur/consommateur dans Rapide

3.4.2.2 Evaluation de Rapide

Rapide est un langage permettant une expression forte de la dynamicité d'une application. Ceci est assuré grâce aux concepts de base de ce langage, qui sont les composants, les évènements et l'architecture. L'expression du comportement d'une IHM, peut se faire

efficacement avec ce langage. Cependant, l'expression de la structure de l'architecture de l'IHM, ne peut pas se faire à cause de l'absence de la notion de composant composite. De plus, les liens qui existent avec les composants métier, ne peuvent pas être exprimés. Ceci est dû à l'absence d'un modèle explicite, pour exprimer les interactions.

3.4.3 L'ADL Wright

3.4.3.1 Présentation du modèle

Wright **[17]** est un langage d'architecture logicielle, qui se centre sur la spécification de l'architecture, et de ses éléments. Il repose sur quatre concepts qui sont : le composant, le connecteur, la configuration et le style.

Un composant en Wright est une unité abstraite, localisée, et indépendante. La description d'un composant contient deux parties importantes, qui sont l'interface (interface), et la partie calcul (computation). L'interface consiste à décrire les ports, c'est-à-dire les interactions auxquelles le composant peut participer. Un port peut également être perçu, comme une facette d'un composant. A chaque port, est associée une description formelle par le langage CSP (CommunicatingSequentialPrecesses) **[73],** spécifiant son comportement. La partie calcul, quant à elle, consiste à décrire le comportement du composant, en indiquant comment celui-ci utilise les ports. Ainsi, les ports qui sont décrits indépendamment dans l'interface, sont utilisés pour décrire le comportement du composant dans le calcul.

Un connecteur en Wright, représente une interaction entre une collection de composants. Il possède un type. Il spécifie le patron d'une interaction, de manière explicite, et abstraite. Ce patron peut être réutilisé dans différentes architectures. Par exemple, un protocole de base de données comme le protocole de validation à deux phases (2PC : two-phase commit 2PC) **[74],**peut être un connecteur. Il contient deux parties importantes, qui sont un ensemble de rôles et la glue. Chaque rôle indique comment se comporte un composant qui participe à l'interaction. Le comportement du rôle est décrit par une spécification CSP. La glue décrit comment les participants, c'est-à-dire les rôles, interagissent entre eux pour former une interaction. Par exemple, la glue d'un connecteur *appel de procédure,* indiquera que l'appelant doit initialiser l'appel, et que l'appelé doit envoyer une réponse en retour.

La description d'une configuration Wright, est composée de trois parties, qui sont : la déclaration des composants et des connecteurs utilisés dans l'architecture, la déclaration des instances de composants et de connecteurs, et les descriptions des liens entre les instances de composants, par les connecteurs.

Wright supporte la composition hiérarchique. Ainsi, un composant peut être composé d'un ensemble de composants. Il en va de même pour un connecteur. Lorsqu'un composant représente un sous-ensemble de l'architecture, ce sous-ensemble est décrit sous forme de configuration, dans la partie calcul du composant.

Chaque partie d'un élément de l'architecture sous Wright (glue du connecteur, rôles du connecteur, calcul, ou ports d'un composant, ..), a une spécification décrivant le comportement de celle-ci. Le modèle utilisé pour la spécification est le modèle CSP. Ainsi, à chaque comportement, est associé un *process* CSP. Le *process* est un patron de comportement, formé d'événements observables et déclenchés par ce *process* (*events*). Les événements déclenchés par le *process* possèdent une barre au dessus, les événements observables n'en possèdent pas ; de plus, les événements peuvent fournir (x!e) ou recevoir (x?e)des données.

L'exemple suivant correspond à celui d'une architecture client/serveur présenté dans **[75]**. Il s'agit d'un client et un serveur interagissant via un connecteur *link*(*cf.* Figure 3.4).

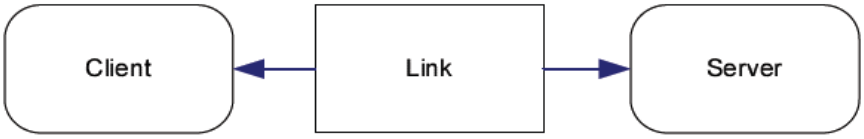


Figure. 3.4 : Architecture Client / Serveur dans Wright

Dans cet exemple, un client fait une requête, elle est reçue par le serveur ; ensuite le serveur fournit une réponse, qui est communiquée au client. Cette séquence d'actions peut se répéter plusieurs fois.

La spécification Wright présentée ci-dessous (*cf.* Figure 3.5), montre les patterns d'interactions, et en plus fournit des détails importants sur les règles d'interactions. L'idée de base est de traiter les composants, et les connecteurs en tant que processus qui se synchronisent.

Component Client
Port p = $\overline{request}$ → reply → p Π §
Computation = internalCompute → $\overline{p.request}$ →p.reply → Computation Π §
Component Server
Port p = request → reply → p □§
Computation = p.request → internalCompute →$\overline{p.reply}$ → Computation□§
Connector Link
Role c = $\overline{request}$ → reply →p Π §
Role s = request → $\overline{reply}$ →p □§
Glue = c : request → $\overline{s:request}$ →Glue
□s : reply→ $\overline{c:reply}$ → Glue
□§
Configuration Client-Server
Instances
C: Client; L: Link; S: Server
Attachments
C.p as L.c ;S.p as L.s
End Configuration

Figure 3.5 : Exemple de description d'architecture Client/Serveur dansWright

L'utilisation du choix interne (Π), dans la spécification du client indique que c'est le client qui décide sur la génération de requêtes. L'utilisation du choix externe (□), dans la spécification du serveur indique qu'on attend que le serveur réponde, à un nombre déterminé de requêtes, et qu'il ne doit pas s'arrêter avant que ceci passe. Une terminaison correcte est indiquée par § (le processus vide).

3.4.3.2 Evaluation de Wright

Wrightprésente un point essentiel, c'est qu'il sépare la notion de composant et de connecteur. Ceci facilite la spécification des composants visuels, et rend plus claire les liens entre instances de composants, ainsi qu'entre composants métier et composants visuels. Cependant, Wright présente l'inconvénient d'être couplé à l'outil de spécification, qui est difficile à assimiler et à utiliser, et qui s'avère inadapté à exprimer le comportement d'une IHM, à cause de l'absence de la notion d'évènement.

3.4.4 L'ADL C2

3.4.4.1 Présentation du modèle

C2 **[76] [77]** est un langage de description d'architecture, conçu par l'université UCI en Californie. Il fut conçu, à l'origine pour la conception des interfaces graphiques. Aujourd'hui, son but est la conception d'autres applications. C2 propose une manière de spécifier une architecture similaire à la plupart des ADLs. En effet, il possède trois abstractions principales, qui sont : le composant, le connecteur et la configuration d'une architecture.

Le but actuel du modèle C2, est de définir une application sous forme de réseau de composants, s'exécutant de manière concurrente, liés par des connecteurs, et communiquant de manière asynchrone par envoi de messages. Chaque composant C2 a son propre état, et son propre fil de contrôle (*cf.* Figure 3.6) **[76]**. Il est composé :

- D'une partie appelée *haut* (*top*) : qui spécifie l'ensemble des événements, que le composant reçoit par le connecteur situé au-dessus de lui, et pour lesquels il exécute des actions, et l'ensemble de services requis par celui-ci.

- d'une partie appelée *bas* (*bottom*) : qui spécifie un ensemble d'événements, que le composant émet à travers l'architecture, pour les composants situés en dessous de lui, et un ensemble de services fournis.

Il est à noter que le haut d'un composant, doit être connecté au bas d'un seul connecteur, et que le bas d'un composant, est connecté au haut d'un connecteur unique. Ainsi, un composant est rattaché à l'architecture d'un système, grâce à deux connecteurs, dont l'un lui permet d'envoyer à travers sa partie « *bottom* », des événements à d'autres composants, et dont l'autre, lui permet d'en recevoir à travers sa partie « *top* » (*cf.* Figure 3.6).

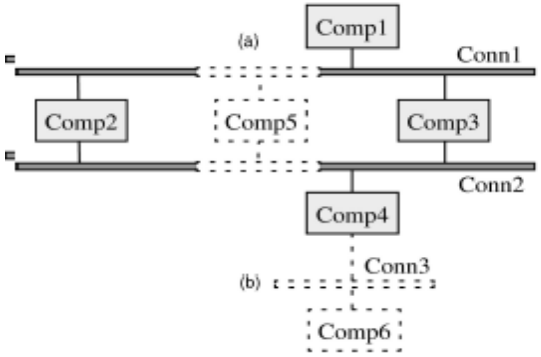


Figure.3.6: Architecture définie par C2

Les connecteurs permettent de lier plusieurs composants. Il n'y a pas de limites du nombre de composants liés à un connecteur. Ils permettent d'une part, de diffuser les messages (demandes de requêtes ou événements) envoyés, et destinés aux composants, et d'autre part de filtrer certains messages.

Une architecture C2 est décrite par un langage prototype, appelé C2 SADL. La topologie de l'application, décrit l'assemblage statique des connecteurs, avec les composants. Elle offre aussi la possibilité d'exprimer une partie de la dynamique d'une architecture.

L'exemple suivant pris de **[78]** spécifie un système, proposant une interface graphique, permettant de manipuler une pile. L'utilisateur peut, grâce à cette interface, empiler ou dépiler un élément dans la pile. Le schéma de l'architecture logicielle en C2 de cette application, est représenté par la figure suivante (*cf.* Figure.3.8).

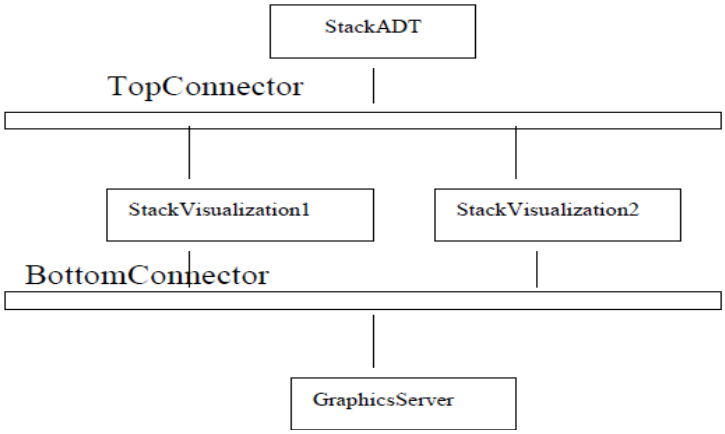


Figure.3.8 : Exemple d'architecture C2

Dans cet exemple, il y a quatre composants :

- Le composant StackADT, qui représente la pile en tant que type de donnée abstraite, et qui gère les fonctionnalités de celle-ci ;

- le composant StackVisualization1, et le composant StackVisualization2, qui gèrent deux représentations graphiques différentes de la pile ;

- Le composant GraphicsServer, qui affiche la pile sur la station de travail.

Et deux connecteurs : TopConnector et BottomConnector.

Le composant StackADT, avec le langage C2 SADL, est défini comme ci-dessous (*cf.* Figure 3.9).

```
componentStackADTis
interface
top_domain
in
null;
out
null;
bottom_domain
in
PushElement (value :stack_type);
PopElement ();
GetTopElement ();
out
ElementPushed (value :stack_type);
ElementPopped (value :stack_type);
TopStackElement (value :stack_type);
StackEmpty ();
parameters
null;
methods
procedure Push (value : stack_type);
function Pop () return stack_type;
function Top () return stack_type;
functionIsEmpty () return boolean;
behavior
received_messagesPushElement;
invoke_methodsPush;
always_generateElementPushed;
received_messagesPopElement;
invoke_methodsIsEmpty, Pop;
always_generateStackEmptyxorElementPopped;
received_messagesGetTopElement;
invoke_methodsIsEmpty, Top;
always_generateStackEmptyxorTopStackElement;
context
top_most ADT
endStackADT;
```

Figure.3.9 : Spécification C2 du composant StackADT

3.4.4.2 Evaluation de C2

Le modèle de composant C2, fût conçu à l'origine pour la conception d'interface graphique. Ceci le rend plus fort que les modèles déjà présentés. C2 a l'avantage de prendre en considération les éléments graphiques dans une spécification, et de prendre en charge tout les évènements qui se produisent dans une IHM. De plus, il permet d'exprimer les liens d'évènements, qui existent entre un composant métier, et un composant d'IHM, comme déjà vu dans l'exemple. Cependant, C2 met beaucoup l'accent sur la spécification d'évènement, en omettant un peu la spécification de la structure d'une IHM, de la composition d'éléments visuels,

et des liens de données qui existent entre composants. La notion de connecteur, bien que celle-ci soit définie de manière explicite, s'apparente plus à un filtre d'événements asynchrones, et donc à une notion d'implantation (médiateur ou bus), qu'à un patron de connexion réutilisable, et donc à une notion de conception.

3.4.5 Le modèle Fractal

3.4.5.1 Présentation du modèle

Le modèle de composant Fractal **[79],** a été défini par France Télécom R&D, et l'INRIA, dans le cadre du consortium ObjectWeb, pour le middleware open source. Sa spécification n'est pas liée à un langage de programmation, ou une technologie particulière, mais son implémentation de référence, Julia, est écrite en Java.

Fractal a été développé conjointement, avec son langage de description d'architecture, Fractal ADL. Son modèle de composants intègre ainsi de façon explicite, les concepts des ADL comme le montre la figure 3.9 (*cf.* Figure 3.9).

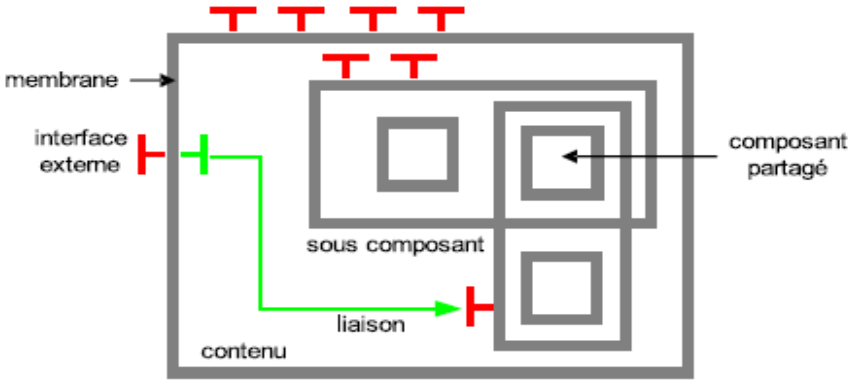


Figure.3.9: Exemple d'un composant Fractal

Dans le modèle de composants Fractal, les composants possèdent une membrane, qui délimite clairement leur contenu, de leur environnement extérieur, afin de structurer l'application. Cette membrane dispose d'interfaces externes (resp. internes), permettant la communication des composants avec l'extérieur (resp. au sein de leur contenu). Leur contenu consiste en un ensemble fini de sous-composants. Le modèle est donc hiérarchique, ou récursif.

Les connecteurs Fractal sont assimilés à des liens d'assemblage, qui relient les composants par leurs interfaces. Les interfaces peuvent être soit des interfaces clientes, c'est-à-

dire des interfaces exposant les services fournis par le composant, soit des interfaces serveurs, c'est-à-dire des interfaces qui correspondent aux services requis par le composant.

Une configuration correspond à l'assemblage des composants, sélectionnés pour constituer l'application.

Le caractère hiérarchique du modèle Fractal, entraîne la distinction entre trois types de composants: les composants composites, les composants primitifs, et les composants partagés.

Les composants composites, sont des composants qui contiennent des sous-composants. Ils permettent d'obtenir une vue uniforme de l'application, à différents niveaux d'abstraction.

Les composants primitifs sont des composants, dont le contenu n'est pas exprimé en termes d'assemblage de composants. Ces composants peuvent servir à encapsuler des composants, qui n'appartiennent pas au modèle de composant Fractal, ou être construits simplement à base d'objets.

Les composants partagés sont des composants, qui sont sous-composants de plusieurs composants composites. Ils permettent de modéliser notamment, le partage des ressources entre composants.

L'exemple ci-dessous, (*cf.* figure3.10*)* montre une application très simple, constituée d'un composant composite, contenant deux composants primitifs. Le premier composant primitif est *"server"*, et qui fournit une interface s de type S. Le deuxième composant primitif est *"client"*,lié à l'interface du serveur précédent.

3.4.5.2 Evaluation de fractal

Fractal est un modèle simple et léger. Sa prise en main est aisée pour les programmeurs et les architectes issus de l'objet. Fractal permet de construire des applications par composition, un composant Fractal peut être considéré comme composant visuel. Fractal permet donc la spécification de la structure d'une IHM. Cependant, Fractal souffre du manque d'information liée au comportement de l'application, ceci fait de Fractal ADL un langage pauvre et difficile à utiliser pour exprimer le comportement d'une IHM.

```
//Le composant Serveur
@Component(provides=@Interface(name="s",signature=Service.class) )

public class ServeurImpl implements Service {
public void print( String msg ) {
System.out.println(msg);
}
 }
//Le composant Client
@Component(provides=@Interface(name="r",signature=Runnable.class) )
public class ClientImpl implements Runnable {
@Requires(name="s")
private Service service;
public void run() {
service.print("Hello world!");
     }
}
//L'assemblage
<definition name="HelloWorld">
<interface name="r" role="server" signature="java.lang.Runnable" />
<component name="client" definition="ClientImpl" />
<component name="serveur" definition="ServeurImpl" />
<binding client="this.r" server="client.r" />
<binding client="client.s" server="serveur.s" />
</definition>
```

Figure 3.10 : Exemple de description Fractal

3.4.6 L'approche IASA

3.4.6.1 Présentation de l'approche

IASA **[80]** (IntegratedApproach to Software Architecture), est une approche intégrée pour l'architecture logicielle, visant à généraliser l'approche architecture logicielle, à la conception de logiciels de tailles diverses, dans les divers domaines de l'informatique.

Les objectifs principaux d'IASA sont :

- L'expression facile de topologies logicielles diverses ;
- l'expression simultanée des aspects structurels et comportementaux ;
- l'isolation des technologies d'interconnexion ;
- l'offre de support d'une approche de conception homogène.

Les éléments fondamentaux de l'approche IASA sont le point d'accès, le composant et le connecteur.

Les points d'accès dans IASA, représentent les concepts de base échangés, entre deux ports de composants. Le point d'accès IASA est destiné à supporter deux concepts : le transfert

de données, et le transfert de flux. Chacun de ces deux concepts est supporté par un point d'accès spécifique (*cf.* Figure.3.11) :

- Les points d'accès dédiés aux transferts de données,sont appelés des DOAP (Data Oriented Access Point) ;

- Les points d'accès dédiés au transfert de flux de contrôle, (invocation d'une action, reprise du flux de contrôle) sont appelés des ACTOAP (Action Oriented Access Point).Un ACTOAP indique la présence d'un service, qui peut être initié à partir de ce point.

Il existe aussi des points d'accès dédiés au contrôle. L'objectif du contrôle, est de permettre la spécification de la disponibilité ou non, d'une ressource, et la spécification du démarrage, arrêt, pause, reprise et redémarrage de service.

Un port IASA est un regroupement de points d'accès, étroitement associés dans le contexte de la réalisation d'un objectif commun.

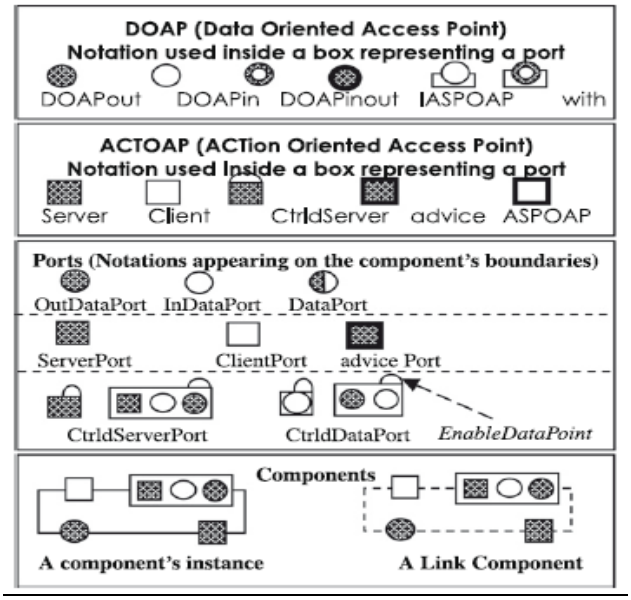


Figure.3.11 : Principales notations graphiques d'IASA

IASA offre quatre types de ports, chacun a un propre usage : les ports ordinaires, contrôlés, orientés aspects, et les ports standards. Il existe quatre types de ports ordinaires, qui répondent à une grande variété d'applications : *ClientPort, ServerPort, PeerPort*et le *DataPort* (*cf.* Figure.3.11) **[81]**. Quand aux ports contrôlés, ils supportent la spécification des diverses

opérations de contrôle, définies dans le contexte des actions de contrôle du langage 3ADL **[80] [82]** (*Enable, Disable, Start, Stop,Pause, Resume, et Restart)*. Le langage 3ADL est l'ADL d'IASA.

Le modèle de composant dans l'approche IASA, distingue entre deux catégories de composants: les primitifs et les composites. Il définit également une organisation d'une vue interne, et une pour la vue externe.

La vue externe est représentée par le concept d'enveloppe, ou sont localisés les ports modélisant le composant (*cf.* Figure.3.12). Le concept d'enveloppe a été introduit dans IASA, pour permettre l'isolation totale de la vue interne d'un composant du monde externe.

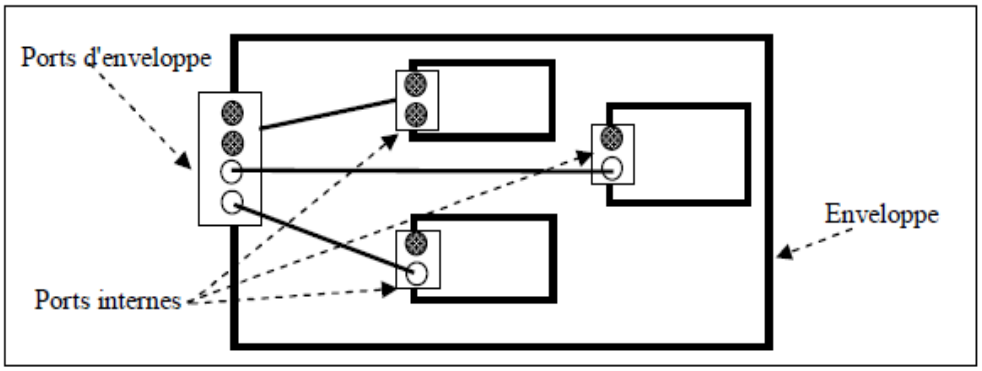


Figure 3.12: Enveloppe, et ports d'enveloppe de l'approche IASA

La vue interne est organisée en deux parties. Une partie opérative, et une partie contrôle (*cf.* Figure.3.13). La partie opérative contient les composants, représentants par leurs interconnexions la logique générale du composite, à un moment bien précis. Elle est représentée par le type *OperativePart***.**

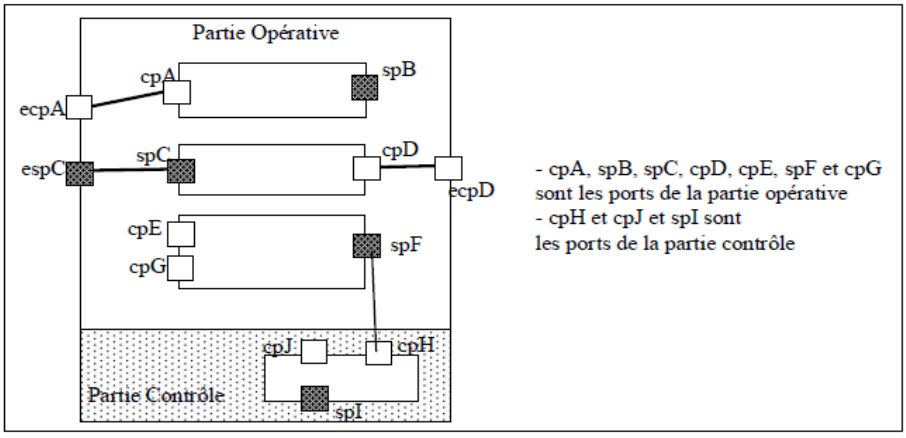


Figure.3.13: Vue interne d'un composant d'IASA

Quand à la partie contrôle, représentée par le type *ControlPart*, réalise les diverses opérations de contrôle, sur les composants de la partie opérative, tels que la gestion du flux de contrôle des divers services (arrêt, lancement en séquence ou en parallèle), le contrôle de l'évolution structurelle, la gestion des exceptions, l'exportation des états du composant, et la génération de log. Les opérations de la partie contrôle sont assurés par quatre composants spécifiques: L'*OpPartController*, l'*OpPartStateCmp***,** l'*OpPartExceptionCmp*et l'*OpPartLogCmp*.

Le modèle de connecteur de l'approche intégrée est basé sur deux vues: une vue abstraite, et une vue concrète. La vue abstraite (resp. concrète), peut comporter plusieurs connecteurs abstraits (resp. concrets). La correspondance entre connecteur abstrait et concret est assurée au niveau du connecteur.

Afin de pouvoir spécifier librement diverses topologies, l'approche propose deux catégories de connecteurs: les connecteurs de transport, et les connecteurs de services. Les deux vues abstraite, et concrète de connecteurs sont basées sur ces deux catégories de connecteurs. Les connecteurs de transport s'occupent du transport des données, et des flux de contrôle. Ils sont composés uniquement de deux rôles, et ne permettent de spécifier que des topologies point à point, permettant de lier deux ports ou deux points d'accès. Un connecteur de service peut disposer de deux ou plusieurs rôles. Il offre les services de communication complexes, tels les services de coordination, de distribution, de facilitation et d'équilibrage de charges.

3.4.6.2 Evaluation de l'approche IASA

L'approche IASA constitue un outil puissant en architecture logicielle, grâce à l'ensemble de modèles qu'elle propose. Ces modèles peuvent considérer simultanément plusieurs niveaux, plusieurs vues et plusieurs aspects : le niveau abstrait, le niveau implémentation, la vue structurelle, la vue comportementale, l'aspect propriétés non fonctionnelles. L'outil IASA Studio de l'approche manque d'outils, permettant la spécification de l'architecture d'une IHM, et qui montrent comment une IHM est organisée réellement par composition d'éléments visuels.

3.4.7 Le langage ArchJava

3.4.7.1 Présentation du modèle

ArchJava**[83]** se situe à la frontière entre le langage de description d'architectures, et les modèles de construction d'applications à composants. Il a été créé à l'université de Washington par Jonathan Aldrich et Craig Chambers. ArchJava a pour objectif d'améliorer la compréhension des programmes, de garantir l'architecture de l'application, de permettre une meilleure évolutivité des applications.

ArchJava travaille sur une partie encore peu explorée des ADLs. En effet, beaucoup d'ADLs décrivent l'architecture logicielle d'une application dans leur propre langage. Au contraire Archjava étend l'implantation d'un langage, en l'occurrence Java, afin d'incorporer les éléments d'architectures, à l'intérieur du code. Comme la plupart des ADLs, les concepts les plus importants d'ArchJava sont le composant, le connecteur et la configuration.

Les composants sont les seules unités d'encapsulation, définies dans le langage ArchJava. Les composants communiquent avec d'autres composants via des ports. Un port définit un ensemble de méthodes, au sens Java classique, qui définissent des points d'accès à un composant. Dans ce sens, les ports ArchJava peuvent être considérés comme des interfaces des composants, au sens large. Un composant peut avoir plusieurs ports. Contrairement à la plupart des modèles qui définissent un sens associé à chaque interface, (serveur/client ou fourni/requis), aucune propriété de ce type n'est associée aux ports dans ArchJava. En effet, ce sont les méthodes associées à un port qui vont donner ce type d'informations. Dans ces termes, une méthode peut être soit de type fourni (implantée par le composant), soit de type requis ou diffusion (appelée par le composant).

La communication entre les composants, se fait à l'aide d'un appel de méthode. Les interactions entre composants connectés, et entre un parent et ses sous-composants immédiats sont autorisées. Par contre, les appels externes à un sous-composant, et les appels entre composants non connectés sont interdits. ArchJava interdit aussi les appels violant la structure hiérarchique de l'architecture.

Pour illustrer la programmation en ArchJava, nous proposons de reprendre l'exemple du compilateur présenté dans **[83]**. La figure 3.15(a) (*cf.* figure 3.15(a)) présente l'architecture d'un compilateur formé de trois composants. L'implantation d'un composant primitif, tel que l'exemple du composant parser (*cf.* figure3.14) est constituée de la définition de ses ports d'entrée et de sortie, et de l'implantation des méthodes fournies.

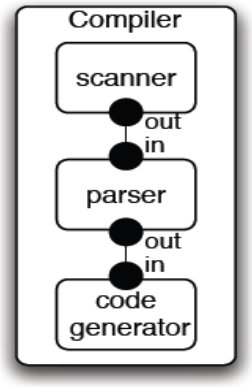
```
public componentclass Parser {
public port in {
provides voidsetInfo ( Token symbol , SymTabEntry e ) ;
requires Token nextToken ( ) throwsScanException ;
}

public port out {
providesSymTabEntrygetInfo ( Token t ) ;
requiresvoid compile (AST ast ) ;
}

voidsetInfo ( Token t , SymTabEntry e ) { . . . } ;
SymTabEntrygetInfo( Token t ) { . . . } ;
. . .
}
```

Figure.3.14 : Description du composant primitif Parser en ArchJava

Dans le cas d'un composant composite, tel que l'application compilateur (*cf.* figure 3.15 (b)), les sous-composants sont déclarés comme des champs privés et finaux. De plus, les interconnexions entre les ports des sous-composants sont définies à ce niveau. Il est à noter qu'un composant composite peut exporter, ou importer les interfaces de ses sous-composants, et peut invoquer directement des opérations de ses sous-composants de premier ordre.




```
public component class Compiler {
private final Scanner scanner = . . . ;
privatefinal Parser parser = . . . ;
privatefinalCodeGencodegen = . . . ;

connect scanner. out, parser.in;
connect parser. out , codegen.in ;
public static void  main ( String args [ ] ) {
newCompiler( ).compile ( args ) ;
}

public void compile ( String args [ ] ) {
/ / for each file in argsdo :
. . . parser.parse( file ) ; . . .
}
}
```

(a) (b)

Figure.3.15 : L'architecture d'un compilateur à base de trois composants principaux (a), et sa description en ArchJava (b).

3.4.7.2 Evaluation d'ArchJava

Le langage ArchJava présente un avantage important, qui est la prise en compte au niveau du développement de l'application, des concepts architecturaux, ceci est une qualité qui permet une meilleure lisibilité du programme. Cependant, pour la spécification des IHM,

ArchJava ne peut être utilisé que pour la spécification de la structure d'une IHM, le comportement de l'IHM est directement implanté. Dans ce sens, ArchJava ne propose pas d'abstraction du comportement d'un composant d'IHM, et aucune analyse ne permet de garantir la compatibilité comportementale, entre des composants d'IHM qui interagissent entre eux ou avec les composants métier.

3.4.8. Le modèle SOFA

3.4.8.1 Présentation du modèle

SOFA (*SOFtware Appliance*) **[84]** est un modèle de composant, développé par l'université Charles (Prague). Comme pour tous les autres modèles, l'objectif principal du modèle Sofa, est la construction d'application à partir d'une composition de composants.

SOFA propose les éléments caractéristiques des langages de description d'architecture, à savoir le composant, le connecteur, et la configuration.

Un composant SOFA peut être primitif, ou composite. Un composite est défini comme un assemblage de sous-composants, alors qu'un primitif ne peut contenir de sous composants. Un composite ne contient pas de fonctionnalités propres, toutes les fonctionnalités sont définies au sein des primitifs. Les composants de SOFA sont définis par leur *frame*,et leur *architecture*. C'est une approche boîte noire du composant. Une*frame* définit les interfaces requises et fournies par le composant. Elle déclare aussi de manière optionnelle, certaines propriétés, permettant de paramétrer le composant. SOFA est un modèle hiérarchique. L'*architecture* est une approche boîte grise du composant. Elle définit soit un assemblage de sous-composants permettant de réaliser une*frame*,soit elle indique que le composant est un primitif. Dès lors, aucun sous-composant ne concourt à la réalisation des services de la*frame*.

Les connecteurs dans SOFA sont des éléments de premier ordre, au même titre que les composants. Ils n'ont pas fondamentalement de différence avec les composants. Ils sont définis à l'aide de *connector-frame* pour la vision boîte noire, et *connector-architecture* pour la vision boîte grise. L'objectif des connecteurs de SOFA, est de permettre de capturer uniquement,la logique métier au sein des composants. Les *connector-frames* sont définis à partir d'un ensemble de rôles. Un rôle est une interface générique pouvant être liée à une interface de composant. Le *connector-architecture* décrit la partie interne du connecteur. Comme pour le composant, cette

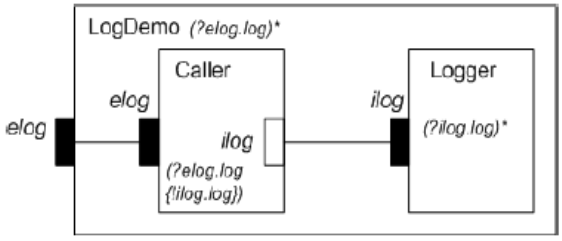
partie peut être déclarée primitive directement implantée ou elle peut être déclarée composée, elle contient alors un ensemble de composants et de connecteurs.

SOFA fournit trois types de connecteur prédéfinis : *CSProcCall* avec une sémantique d'appel de procédure,*EventDelivery* avec une sémantique d'envoi de message et *DataStream*pour une sémantique d'échange de flux. Il fournit aussi un type de connecteur utilisateur, qui permet de définir sa propre sémantique. Cette dernière sera alors réalisée à l'aide d'un assemblage de composants et de connecteurs. De manière implicite, si aucun connecteur n'est spécifié, un connecteur possédant une sémantique d'appel de procédure est introduit.

Dans une configuration SOFA, il y a quatre manières de lier deux composants. La première appelée le *binding,*permet de créer une liaison entre une interface requise, et une interface fournie, de deux composants de même niveau. La deuxième appelée *delegating,*lie une interface fournie du composite, à une interface fournie d'un de ses sous-composants. La troisième nommée *subsuming,*permet de lier un port requis d'un sous-composant à un port requis du composite. Enfin, la dernière nommée *exempting,*permet de déclarer qu'une interface d'un composant n'est pas liée. Les trois premiers types de liaisons, sont réalisés par des connecteurs.

DCUP **[84]**, est lamise en œuvre de référence, du modèle SOFA.

Dans l'exemple ci-dessous (*cf.* Figure3.16), il y'a deux composants primitives qui sont *Caller* et Logger. Logger fournit l'interface LogInterface, Caller requière cette dernière. Ces deux composants sont instanciés.



(a)

```
module logdemo {
interface LogInterface {
void log( in string number);
};
frame Caller {
requires:
LogInterface Log;
};
frame Logger {
provides:
LogInterface Log;
};
};
```

(b)

Figure.3.16 : L'architecture du composant LogDemo(a), et sa description en Sofa (b)

3.4.8.2 Evaluation du modèle SOFA

SOFA est un modèle de composants issu d'une équipe, qui a beaucoup travaillé dans le domaine des objets répartis. Bénéficiant des différents travaux dans le domaine de l'architecture logicielle, l'équipe propose un modèle classique avec une bonne séparation entre la notion de type de composant (appelé *frame*), et la notion d'implantation (appelé *architecture*). L'inconvénient majeur de Sofa est qu'il propose au niveau du modèle, la notion d'interface requise ou fournie,plutôt que la notion de port. Dès lors, si cette distinction ne pose pas de problème au niveau de la mise en œuvre, elle limite la pertinence du modèle pour la conception d'une IHM.

3.5 Les modèles de composants industriels

La communauté CSBE part du principe selon lequel, il est possible de développer une application en assemblant des composants préfabriqués, provenant de fournisseurs différents **[85]**. Pour mener à bien un tel développement, il est nécessaire que le concepteur d'application à base de composants, puisse effectivement interconnecter les différents composants sélectionnés pour composer l'application. Il est donc crucial que chaque parti (concepteurs et fournisseurs) s'accorde sur un modèle de composants, c'est-à-dire un standard commun permettant la manipulation uniforme des composants et leur interopérabilité.

« A component model specifies the standards and conventions imposed on developers of components.» **[86]**

Un modèle de composants établit donc une norme, définissant un ensemble de règles, et de contraintes techniques, imposées sur la construction et l'utilisation des composants. Pour réaliser son application, le concepteur peut alors s'appuyer sur le modèle de composants utilisé par son application, afin d'établir des hypothèses sur les composants qu'il réutilise.

Cette section présente une étude de divers modèles à composants. Chaque modèle est présenté, en décrivant son domaine d'application, et ses caractéristiques. Un exemple permettant de voir la manière suivant laquelle les concepts sont réalisés, au niveau du code est aussi présenté.

3.5.1 Le modèle COM

3.5.1.1 Présentation du modèle

Le modèle COM (Component Object Model) de Microsoft **[8] [87]** a été conçu, pour résoudre le problème de l'interopérabilité au niveau binaire, entre des composants appartenant à des applications écrites par différents vendeurs. Un exemple typique d'interopérabilité, est l'incorporation de fonctionnalités fournies, par une application dans une autre ; par exemple le composant moteur HTML d'un navigateur, peut être incorporé dans un reproducteur de médias, qui nécessite d'afficher des pages web, même si ces deux composants sont implémentés dans des langages différents.

Un composant est une entité de réutilisation de logiciels au niveau du binaire. Il n'est pas nécessaire d'avoir accès au code source du logiciel, pour pouvoir l'utiliser de manière cohérente. L'accès au «binaire » est facilité par la présence d'une ou plusieurs interfaces associées au composant.

Le modèle de composants COM, permet de séparer l'utilisation des opérations implantées dans des modules logiciels, de leurs détails d'implantations. Bien que COM impose une séparation entre les interfaces fonctionnelles, et leur implémentation (*cf.* Figure3.17), il ne propose pas de moyen de décrire la vue externe d'un composant, du fait que le nombre d'interfaces fonctionnelles peut varier. Un client d'une instance de composant, doit toujours vérifier la présence d'une interface qu'il compte utiliser. Une interface dans COM, contient un ensemble de méthodes, est décrite dans un langage de description spécifique, (IDL : Interface DefinitionLanguage) et est identifiée de façon unique.

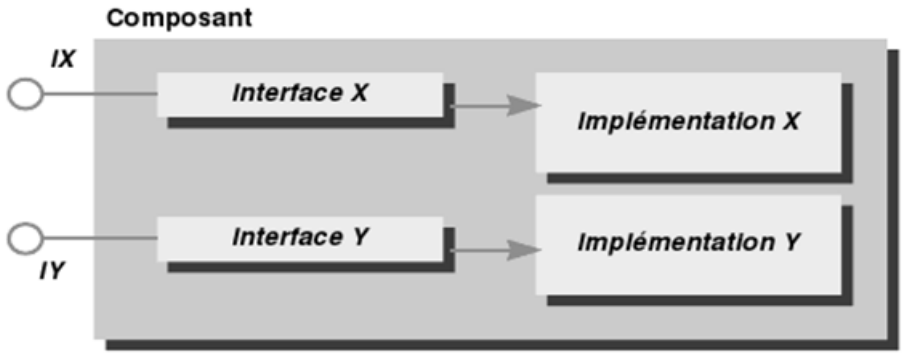


Figure.3.17 : Structure d'un composant COM

L'encapsulation est la base du modèle COM, les données sont cachées, et accessibles par le biais unique de services définis dans une interface.

Pour permettre l'évolution des interfaces, un composant COM peut implémenter simultanément plusieurs versions d'une même interface (*cf*. Figure 3.18). Toutes les interfaces de COM héritent d'une interface racine, appelée *IUnknown*, qui contient des méthodes permettant aux clients d'introspecter une instance de composant, pendant l'exécution. Les méthodes fournies par *IUnknown* sont employées par un client pour deux raisons : pour tester la présence d'une interface particulière, ou bien pour voir si le client implémente une nouvelle version d'une interface.

```
a) Interfaces fonctionnelles
interfaceIClient : IUnknown {
HRESULT receive([in] string Data) ;
} ;

interfaceIServer : IUnknown {
HRESULT addClient([in] Client *client) ;
HRESULT GetClients([out] int clients) ;
} ;
b) Instantiation
IServer *pServer = NULL ;
HRESULT hr = CoCreateInstance (
CLSID_ServerImpl, // identifie implementation
NULL,
CLSCTX_SERVER,
IID_IServer,
reinterpret_cast<void**> (&pServer)) ;
if( !SUCCEEDED(hr))
{
// L'objet COM n'a pas pu être créé...
}
d) Composition
IClient *pClient = NULL ;
hr = CoCreateInstance (...) ;
if(SUCCEEDED(hr)) // Le client à pu être créé
{
pServer->addClient(pClient) ; // connexion des instances
}
```

Figure.3.18 : Exemple de COM

3.5.1.2 Evaluation du modèle COM

COM propose avant tout un modèle binaire, pour mettre en place des composants logiciels. Ce modèle augmente en effet l'implantation binaire des objets (en particulier celui de C++), pour en faire des composants qui peuvent fournir plusieurs interfaces distinctes. Les problèmes majeurs de COM, sont la difficulté d'écriture des composants, et le manque d'informations relatives aux relations, et aux interdépendances entre les composants. Ceci n'aide pas à concevoir la structure d'une IHM.

3.5.2 Le modèle JavaBeans

3.5.2.1 Présentation du modèle

Sun Microsystems a mis au point le modèle de composants JavaBeans **[88],** en particulier pour la construction d'interfaces graphiques utilisateur (GUI : Graphical User Interface). Sa simplicité et son efficacité en font cependant un modèle de choix, pour de nombreuses autres applications.Sun**[89]** propose la définitionsuivante des JavaBeans :

« A Java Bean is a reusable software component that can be manipulated visually in a builder tool.»

Cette définition met en avant le caractère réutilisable de ces composants, ce qui est clairement dans l'esprit de la définition de Szyperski. Un composant JavaBean n'est donc pas conçu, pour fonctionner seulement dans une application. Il peut être composé par des tiers, et doit même pouvoir l'être visuellement. Sun démontre les possibilités de son modèle de composants, avec le logiciel Bean Builder qui se distingue notamment par :

- des mécanismes d'introspection, qui décrivent les interfaces et les propriétés des composants ;
- la possibilité de configurer manuellement les propriétés des composants ;

- une communication événementielle permettant l'assemblage des composants, grâce à un mécanisme de publication/abonnement de notifications ;

Les JavaBeans sont donc explicitement conçus, pour interagir dans deux contextes différents, au moment de la composition dans l'outil de construction, et à l'exploitation réelle dans l'environnement d'exécution **[85]**. Cela est principalement mis en œuvre grâce aux mécanismes de réflexion de la plate-forme Java, (paquetage *java.lang.reflect.**) qui permettent à l'exécution de retrouver la classe d'un objet, ses méthodes, ses attributs et de les manipuler. Ainsi les JavaBeans sont de simples objets Java, sur lesquels le modèle de composants impose un certain nombre de conventions de nommage, pour spécifier leurs propriétés, leurs événements, ou encore leurs méthodes. Par exemple, une simple propriété est définie sur un composant par une méthode portant le nom de la propriété, préfixé par les termes *get,*ou *set* selon que la propriété doit être accessible en lecture ou en écriture (*cf.* Figure 3.19).

```
a) Classe de composant
// Vue externe et implémentation
packageorg.examples ;
interfaceServerConfig
{
public intgetMaxClients() ;
public voidsetMaxClients(int max) ;
public voidaddClient(Client c) ;
}
public class Server implementsServerConfig
{
privateintmaxClients = 2 ;
private List clients = new ArrayList() ;
Server() {  }
public intgetMaxClients() // propriété de configuration
{  returnmaxClients() ;
}
public voidsetMaxClients(int max)
{  maxClients = max ;
}
public voidaddClient(Client c) // interface requise
{  clients.add(c) ;
}
} ;

b) Instantiation
Server server = (Server)
Beans.instantiate(this.getClassLoader(),"org.examples.Server") ;

d) Composition
// Exemple de composition impérative
Client c1 = new Client() ; // Création d'un client
c1.setName("Client1") ; // Configuration
server.addClient(c1) ; // Connexion des instances
```

Figure.3.19 : Exemple de JavaBeans

3.5.2.2 Evaluation de JavaBeans

Les JavaBeans constituent un modèle simple, tout à fait approprié pour le développement d'applications clientes. L'intérêt de cette approche est de permettre la construction de la partie « visuelle » de l'application. C'est le modèle le mieux adapté pour les IHM, par rapport aux modèles déjà présentés dans ce chapitre. L'utilité de JavaBeans trouve ses limites pour la description de l'architecture, ce qui est très important pour tout système. Il n'y a pas de moyens de visualiser l'architecture de l'application, ni les liens entre les différents composants, ou entre partie métier et partie IHM, ceci n'est possible qu'en consultant le code source des Beans.

3.5.3 Le modèle Entreprise Java Bean

3.5.3.1 Présentation du modèle

Le modèle à composants EJB de Sun **[90],** se place dans un contexte d'applications, construites selon une architecture répartie en trois tiers **[91]**: un tiers de présentation, qui réside principalement dans une machine du côté de l'utilisateur, un tiers applicatif, qui réside dans un serveur, et un tiers de données, correspondant par exemple à une base de données. L'idée de cette architecture, est de permettre l'interaction des utilisateurs avec les données, en passant par le tiers applicatif (appelé aussi *tiers milieu*). L'interaction des utilisateurs avec les instances du tiers applicatif, est souvent réalisée à partir d'un navigateur, et donc cette interaction est de type sessionnel. Ces applications trois tiers nécessitent d'un support de plusieurs caractéristiques non fonctionnelles, telles que la transaction, la sécurité, et la distribution. Le modèle EJB est spécifiquement orienté à la construction du tiers applicatif.

Il est intéressant de noter que, bien que l'architecture trois tiers est de nature répartie, le modèle EJB ne suppose pas que les applications du tiers du milieu soient distribuées, bien que cela reste réalisable.

La vue externe des classes de composant EJB, est constituée d'une unique interface fonctionnelle fournie, appelée l'interface *remote*(*cf.* Figure 3.20). EJB ne permet pas de décrire de façon explicite, les interfaces requises par les instances du composant, ou les propriétés de configuration, un composant EJB n'a qu'une seule interface serveur, et n'a aucune interface client faisant partie de son type.

Il existe trois types de composant EJB : les composants session, message et entité. Les composants session sont destinés à être instanciés pour chaque session d'interaction, entre un client et le serveur. Leur durée de vie est équivalente à la durée de validité de l'interaction. Les composants session sont dédiés à l'implantation de la logique (du comportement) de métier. Les composants message ont essentiellement le même rôle que les composants session, mais sont compatibles avec le mode de communication asynchrone. Enfin, les composants entité sont de durée de vie indéfinie, et sont dédiés à la réification des données du serveur.

Les instances de composant session, message ou entité sont créées à partir d'une fabrique appelée un *home,*qui permet soit de créer une nouvelle instance, soit de retrouver une instance à partir d'une clé qui l'identifie. Il est important de noter que les clients du tiers présentation, n'interagissent pas de façon directe avec les composants de type entité, mais seulement à travers les composants de type session.

Développer une application à partir de composants EJB de base, consiste à créer de nouveaux composants, responsables d'instancier, et connecter des composants de base. La composition est normalement impérative à travers le langage Java. Les composants EJB sont assemblés et exécutés au sein de conteneurs. Les conteneurs implantent la création, et la destruction des composants EJB, et sont en charge de leur fournir un environnement d'exécution comprenant des services non-fonctionnels de type gestion des transactions, etc. Un conteneur s'exécute au-dessus d'une machine virtuelle Java, sur un serveur physique, et peut gérer plusieurs types d'EJB. L'instanciation de composants se fait par l'interface Home, fournie par les conteneurs.

L'interface de base pour un composant EJB s'appelle EJBObject. Via cette interface, on peut obtenir une référence unique pour l'instance de composant, et l'on peut accéder au conteneur qui l'accueille. Cette interface de base doit être étendue, pour définir des types de composants EJB, avec leurs fonctions métiers (*cf.* Figure 3.20).

L'interface de base pour mettre en œuvre un conteneur EJB, s'appelle EJBHome. Cette interface fournit une référence unique pour le conteneur, et permet d'effectuer des recherches, parmi les composants lui appartenant. Cette interface doit être étendue, pour spécifier les opérations spécifiques à un type de conteneur donné.

a) Classe de composant

```
// Interface fonctionnelle (vue externe)
packageorg.ejbexamples ;
interfaceEJBServer extends javax.ejb.EJBObject
{ …….
public void addClient(Client c) throws RemoteException ;
public void startServing() throws RemoteException ;
}
// Méthodes de l'interface fonctionnelle (vue externe)
public voidaddClient(Client c)
{     clients.add(c);              }
} ;
```

b) Instantiation

```
// Obtenir référence vers le home
….
Object obj = ctx.lookup("ServerHome") ;
ServerHomesrvHome = (ServerHome)
Server server = (Server) srvHome.create() ;
....
server.remove() ; // destruction de l'instance
```

c) Composition

```
server.addClient(client) ; // Connexion des instances
```

Figure.3.20 : Exemple du modèle EJB

3.5.3.2 Evaluation du modèle EJB

Le modèle EJB facilite la programmation des applications d'entreprise, pour lesquelles il a été conçu. Il a l'avantage de prendre en charge les aspects non fonctionnels d'un composant, comme la sécurité, et les transactions. Le modèle EJB a été conçu pour un type d'application précis, et ne propose aucun aspect pour prendre en compte l'architecture d'une IHM, surtout qu'un composant EJB n'a qu'une interface serveur.

De plus, les conteneurs fournissent un modèle de composition assez limité (pas hiérarchique), et laissent aux programmeurs de composants, la gestion des liaisons entre les composants. Ces propriétés nous semblent limitatives, dans le cadre de la mise en place d'une IHM, surtout dans le cas des IHM complexes et à grande échelle.

3.5.4 Le modèle Corba Component Model (CCM)

3.5.4.1 Présentation du modèle

CORBA (Common Object Request Broker Architecture) **[7]** est un standard proposé par l'OMG (Object Management Group), pour la programmation des objets répartis. CORBA spécifie des interfaces de programmation, et des protocoles de communication de type client/serveur, qui permettent de déléguer la mise en œuvre de tous les aspects liés à la communication, et à la synchronisation des objets répartis, avec leurs environnements d'exécution sous-jacents. Conceptuellement, le CCM peut être considéré comme étant une extension des EJB, pour rendre ce modèle indépendant des langages de programmation, utilisés pour développer les composants.

CCM spécifie essentiellement une structure de composant, et un environnement d'exécution fournissant des services non-fonctionnels, comme dans le cas des EJB.

Un composant, d'après le modèle CCM, est une entité logicielle qui peut fournir, ou recevoir des opérations à travers des interfaces, appelées *ports*. Comme illustré dans la figure 3.21 (*cf.* Figure 3.21), les composants peuvent avoir de multiples ports, ainsi que des attributs qui donnent accès à leurs paramètres configurables. CCM distingue quatre sortes de ports : les *facettes* et les *réceptacles* sont respectivement, les interfaces serveur et client en mode de communication synchrone, alors que les *puits* et les *sources*, sont utilisés en mode de communication asynchrone. Il est à noter que la notion de type de composant, est présente dans ce modèle et correspond à l'ensemble des ports du composant.

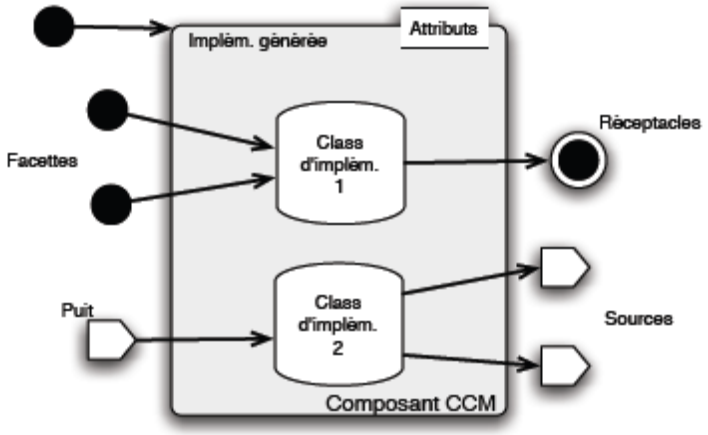


Figure.3.21 : Structure interne d'un composant CCM

Les composants CCM sont décrits dans le langage IDL3**[7]**, qui est capable de spécifier le concept d'interfaces fournies (*provide*) et requises (*use*) (*cf.* Figure 3.22). La description d'une application (instanciation des composants, et de leurs éventuelles connexions) est réalisée dans le langage OSD (*Open Software Descriptor*)**[7]**, qui est un langage basé sur *XML*.

a) Classe de composant

```
// Vue externe
interface Server {...} ;
interfaceServerConfig {...} ;
interface Client {...} ;
componentMyServer
{
attributeintMaxClients ;
provides Server server ;
providesServerConfigserverCfg ;
uses multiple Client clients ;
}
//implémentation
composition process MyServerImpl
{
home executor MyServerHomeImpl
{
implementsMyServerHome ;
managesMyServerImpl ;
} ;
} ;
```

b) Balises de composition

```
<componentassembly>
<description>
<componentfiles>
<partitionning>
<connection>
<connectinterface>
<usesport>
<providesport>
</connectinterface>
<connectevent>
<consumesport>
<publishessport>
</connectevent>
</connection>
</componentassembly>
```

Figure.3.22 : Exemple de CCM

Les composants CCM évoluent dans des conteneurs, définis dans le contexte du bus CORBA. Les conteneurs exploitent les divers services de CORBA, tels que les services de transactions, et de notifications.

3.5.4.2 Evaluation de CCM

L'OMG propose ici un cadre unifié pour concevoir des composants. Les langages de description comme CORBA IDL, accompagnent le modèle pour permettre la mise en œuvre des descriptions de haut niveau, et facilitent la programmation en générant une partie du code nécessaire à l'implantation des composants. Cependant, le modèle de composition des composants CCMau sein des conteneurs, reste à un niveau équivalent à celui des EJBs, et nous semble limitatif pour la mise en place d'IHM. De plus, l'absence de connecteur comme pour le cas d'EJB ne va pas permettre la spécification d'IHM et de liens existant entre les différents composants.

3.6 Bilan sur les approches à base de composants

Cette étude des différents ADLs, et de modèles de construction d'applications à base de composants (CMs), permet de constater la grande diversité dans les caractéristiques, et les objectifs de ces approches. En effet, même si la partie structurelle du composant, semble maintenant relativement stabilisée avec l'utilisation de la notion de port, d'interface, d'opération ou de méthode, la sémantique du composant varie encore beaucoup selon les modèles. Ainsi, un composant peut être assimilé à un processus, une zone de mémoire partagée, ou même un élément matériel. De ces différences d'objectifs, et de caractéristiques découlent de nombreux critères d'évaluation possibles d'un modèle. Nous avons choisi les critères déjà présentés dans la section 3.3, et qui sont liés à la spécification d'interface homme machine.

3.6.1 Caractéristiques liées à la structure de l'IHM, et des liens entre les différents composants

La figure 3.23 (*cf.* Figure.3.23) représente un tableau, qui récapitule les caractéristiques des travaux présentés, sous les critères de la structure des IHMs déjà définis. Nous remarquons qu'à part les modèles C2 et JavaBeans, aucun modèle n'aborde la spécification d'IHM, et donc ne propose un modèle de référence pour les IHM. De même, beaucoup de modèle souffre du manque de mécanisme de liaisons, entre différents composant, ceci est très important pour la spécification d'architecture des IHMs. Par contre, beaucoup de langages offrent le mécanisme de composition hiérarchique, qui est un concept puissant pour la mise en place d'architecture des

IHMs. Enfin, la plupart des propositions permettent à un composant, d'avoir des attributs qui lui sont propres.

	Modèle de référence pour IHM	Composition	Attributs de composant	Liens entre composants
Darwin	Non	Non	Oui	Oui
Rapide	Non	Non	Non	Non
Wright	Non	Oui	Non	Oui
C2	Oui	Non	Oui	Oui
Fractal	Non	Oui	Oui	Oui
IASA	Non	Oui	Oui	Oui
ArchJava	Non	Oui	Oui	Non
Sofa	Non	Oui	Oui	Oui
COM	Non	Difficile	Non	Non
JavaBeans	Oui	Oui	Oui	Non
EJB	Non	Oui (conteneur)	Oui	Non
CCM	Non	Oui (conteneur)	Oui	Non

Figure.3.23 : Tableau comparatif synthétisant les caractéristiques de la structure d'IHM

3.6.2 Caractéristiques liées au comportement de l'IHM

Le tableau 3.24, (*cf.* Figure 3.24)résume comment il est possible à l'aide des approches étudiées dans ce chapitre, de spécifier le comportement d'une IHM. Nous remarquons principalement que la majorité des propositions, se focalise soit sur la structure de l'architecture, soit sur le comportement. Il n'existe pas de modèle qui couvre tout les critères cités. Darwin par exemple satisfait tout les critères de spécification de comportement, par contre ne couvre aucun critère de spécification de la structure d'un système. Il est à remarquer que l'approche IASA, couvre beaucoup de critères, c'est une approche assez flexible, ceci est du aux objectifs intéressant de l'approche qui ont été fixés.

	Réaction à un évènement	Communication asynchrone	Déclenchement de service	Spécification d'intérêt à un évènement

Darwin	Non	Oui	Limité	Non
Rapide	Oui	Oui	Oui	Oui
Wright	Non	Non	Oui	Non
C2	Non	Oui	Oui	Oui
Fractal	Oui	Difficile	Oui	Non
IASA	Oui	Oui	Oui	Non
ArchJava	Non	Oui	Oui	Non
Sofa	Oui	Oui	Oui	Non
COM	Non	Non	Non	Non
JavaBeans	Oui	Oui	Oui	Oui
EJB	Oui	Oui	Oui	Non
CCM	Oui	Oui	Oui	Non

Figure.3.24 : Tableau comparatif synthétisant les caractéristiques du comportement d'IHM

Nous tirons les conclusions suivantes de cette analyse d'état de l'art :

- Il existe de nombreux modèles de composants, qui permettent de simplifier le développement d'une application par abstraction des services systèmes. Les modèles de composants fournissent des notations, pour décrire et analyser les systèmes logiciels. Ils sont généralement accompagnés par divers outils destinés à analyser, simuler, et parfois générer le code des systèmes modélisés. Un certain nombre d'ADLs, offrent également un support pour modéliser le comportement, et les contraintes sur les propriétés des composants et des connecteurs.

- Dans les modèles de composants, et les ADLs vus, les composants sont principalement considérés comme des entités *métier*. Ils sont l'incarnation de la description par le concepteur, de la logique applicative du système en concepts clés clairement identifiés. Ces composants sont nommés dans la littérature « composants métier » **[92]**.

- Les modèles de composants et les ADLs actuels, proposent de nombreux services qui font la valeur ajoutée de ces solutions. On trouve par exemple des services pour gérer les aspects non fonctionnels d'une application, tels que la sécurité, les transactions. Ces modèles n'abordent

pas le problème de composition des IHMs. Ainsi, une application construite par assemblage de composants, verra ainsi son IHM conçue par les méthodes que nous avons abordées au deuxième chapitre.

- Les modèles de composants proposés pour l'assemblage de composants métier, sont très difficiles à utiliser pour les IHMs, et parfois même inadaptés.

Grace à l'étude des modèles actuels, nous avons pu identifier les causes, qui rendent ces derniers inefficaces pour la spécification d'architecture d'une IHM. Ces raisons sont les suivantes :

- Un composant n'est associé qu'à une seule sémantique. Dans le cas de DARWIN par exemple, un composant est un processus. Ainsi, un composant ne permet pas d'exprimer un autre élément, tel qu'une IHM qui ne correspond pas à un processus.

- L'ADL est basé sur un langage inadapté à la spécification d'IHM, et difficile à assimiler, comme dans le cas de Wright. En effet, l'outil de spécification CSP couplé à Wright, n'est pas facile à comprendre, de plus, il ne propose pas non plus un moyen, de raffiner l'architecture d'une application, en proposant plusieurs niveaux de description, et en sauvegardant les traces de passage de l'une à l'autre, ce qui cause un problème de complexité lorsque l'IHM est de grande taille.

- Les connecteurs ne sont pas décrits de manière explicite, comme dan beaucoup de cas comme COM, EJB, CCM ou de Rapide. Dans ce cas, les interactions ne peuvent pas être décrites dans l'architecture, surtout les liens de données et d'évènements qui existent entre les composants métier et les composants d'IHM.

- Le modèle de composants est orienté vers un domaine spécifique, très spécialisé, et ne peut pas être utilisé dans d'autres domaines (specialized component model). Medvidovic et Taylor**[16],** et**]** Vestal**[68]** proposent respectivement une étude comparative entre les différents ADLs, et CMs, dans lesquelles on trouve une classification des modèles. Il existe

beaucoup de modèles à usage général (generalpurpose), comme pour le cas de Fractal. Comme il existe des modèles de composants spécialisés, comme pour le cas d'AUTHOSAR **[93]**, qui est dédié pour le développement des standards logiciels communs, pour les unités de contrôle électronique (UCE) des automobiles.

- Les modèles de composants actuels, ne mentionnent pasles composants d'IHM, et n'indiquent pasde méthodes permettant leur intégration, dans une spécification d'architecture logicielle.

- Les travaux sur la composition au niveau de la partie métier, décrivent de manière implicite les différentes opérations, sans se préoccuper de l'enchaînement des tâches utilisateurs. quelque soit le modèle utilisé, on a juste la description des liens qu'il y a entre les différentes données utilisées, ainsi que les opérations. Pour une IHM, on a besoin de décrire les tâches utilisateur.

3.7 Conclusion

Nous avons présenté dans ce chapitre, les différentes propositions pour la programmation à base de composants, qui ont été introduites dans la dernière décennie. La programmation à base de composants, propose de bonnes briques de bases pour aborder la programmation d'applications, et de systèmes complexes. Néanmoins, il n'y a pas de modèle de composants idéal pour satisfaire l'ensemble des besoins. En d'autres termes, des propriétés qui semblent très intéressantes pour un certain cadre applicatif, peuvent constituer un inconvénient pour d'autres. Pour cette raison, le modèle doit être flexible, pour être adapté à chaque contexte applicatif. La communauté des chercheurs du domaine d'architecture logicielle, a consacré beaucoup d'efforts pour le développement d'application à base de composants, ces efforts se situent surtout dans la partie métier d'une application. La partie IHM n'a pas bénéficiée d'un grand nombre de propositions, JavaBeans et C2 constituent les seules propositions, souffrant de quelques lacunes déjà citées. Nous proposons au cours de cette thèse, de définir un modèle de composant pour la spécification de l'architecture d'une IHM, ce modèle sera proposé dans le cadre de l'approche IASA, qui est une approche très flexible, et qui peut être adaptée dans plusieurs contextes applicatifs.

CHAPITRE 4
UN MODELE DE COMPOSANTS POUR LA SPECIFICATION D'IHM SELON L'APPROCHE IASA

4.1 Introduction

Dans ce chapitre, nous présentons notre solution, permettant l'utilisation d'une approche *architecture logicielle,* pour la conception et la réalisation d'IHM, par assemblage de composants. Notre solution a été élaborée, dans le contexte d'un projet, dont l'objectif est la mise en place d'une approche intégrée d'Architecture Logicielle, appelée IASA (the IntegratedApproach to Software Architecture) **[80].** IASAest une approche orientée aspect d'Architecture logicielle. Elle vise à généraliser l'approche *architecture logicielle*, à la conception de logiciels de tailles diverses. En plus de permettre le raisonnement avec des composants, représentant des sous systèmes d'une application (gros composant), l'approche IASA permet de supporter un raisonnement d'assemblage de composants, ou ces derniers sont très fins, tels qu'un operateur arithmétique ou logique.

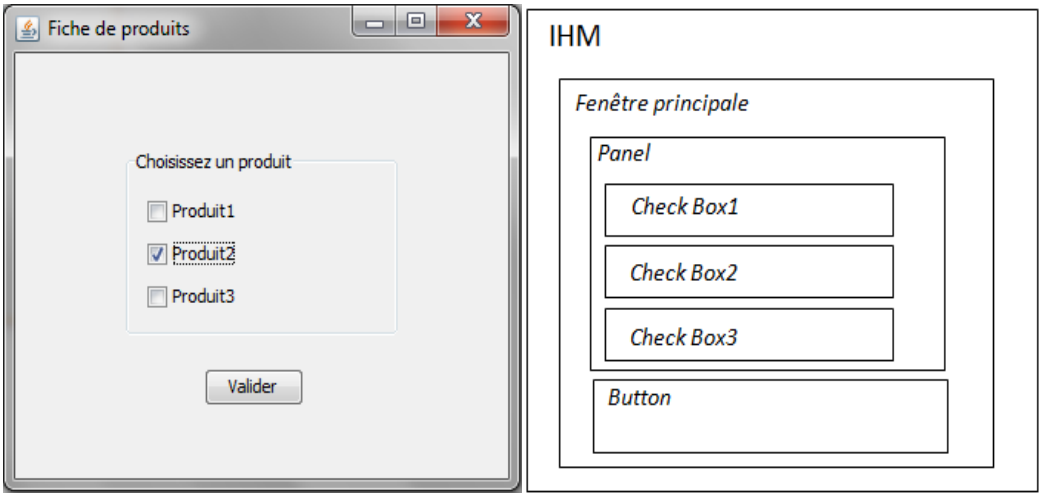
Notre solution exploite fortement les résultats obtenus, dans le contexte de l'exploration de l'état de l'art. L'originalité de notre solution,c'est de permettre de spécifier une l'IHM, comme un assemblage de composants d'IHM. Notre objectif dans le contexte du projet IASA, consiste principalement à transformer l'approche IASA, en une approche très bien adapté à la spécification d'IHM, par assemblage de composants. L'impact direct de notre travail, concernera principalement l'outil IASASTUDIO, qui devrait être revu pour intégrer nos résultats.

4.2 Eléments essentiels pour compositions d'interface homme machine

Après avoir analysé les causes, qui rendent les modèles de composants actuels, peu efficaces pour la spécification d'architecture d'IHM, nous présentons dans cette section quelques propositions, visant la mise en place d'un modèle de composants, permettant une spécification efficace d'IHM. Le but de notre travail, étant de permettre la spécification des deux vues d'une IHM, tout en gardant la cohérence entre les deux (*cf.* Figure4.1). Les deux vues sont :

- La vue présentation : similaire à celle que nous trouvons dans les environnements intégrés, de développement rapide d'application, tels que Delphi ou JBuilder(*cf.* Figure4.1(a)).

- La vue logique : dans laquelle l'IHM est représenté, sous forme d'un assemblage de composants (*cf.* Figure4.1(b)).



(a) (b)

Figure.4.1 : La vue *présentation* d'une IHM(a), et sa vue logique (b)

Actuellement, la majeure partie des applications, utilisent les composants visuels, dans le contexte de la réalisation de leurs interfaces homme machine. Ces composants sont parfois appelés des composants de présentation**[94],**ou des composants d'IHMs. Les résultats obtenus de notre état de l'art, révèlent queles modèles de composants actuels, ne mentionnent pas ces composants, et n'indiquent pasde méthodes permettant leur intégration dans une spécification d'architecture logicielle.

Notre objectif étant de transformer l'approche IASA, en une approche supportant la spécification aisée, et efficace d'IHM. De ce fait, nous nous sommes concentré principalement, sur l'étude du modèle de composant IASA, pour le transformer en un modèle, pouvant supporter la représentation des diverses caractéristiques d'un composant visuel. Le défi auquel nous étions confronté, est la nécessité de rester dans le contexte de ce modèle de composant, qui possède notamment une organisation interne bien précise, et sur laquelle est basée entièrement l'approche IASA. La transformation ne doit pas remettre en cause les applications déjà spécifiée en IASA.

Dans ce contexte, nous devons relever un fait important, qui nous a permis de relever ce défi. C'est ainsi que nous avons bénéficié hautement, du concept d'aspect que supporte l'approche IASA. L'approche *aspect* d'IASA recommande, lors de la conception de composant, de les concevoir, et de les réaliser de manière totalement libre, de tout contexte de mise en œuvre. De tels composants sont dits : *des composants inconscients (oblivious component)*. L'assemblage de ces composants dans une application, consiste à les connecter, soit par des connecteurs ordinaires, ou des connecteurs *aspect*.

Nous présentons dans ce qui suit, les critères les plus importants, que nous devons prendre en considération, lors de la proposition du modèle de composants.

4.2.1. Type de composant pouvant représenter une IHM

Vu la grande différence entre la spécification de la partie métier, et la spécification de la partie IHM, dans une application, il est très difficile d'utiliser un même modèle de composants, pour représenter simultanément un composant métier, et un composant d'IHM.

Comme première solution, pour permettre à IASA la spécification d'IHM, nous avons introduit la nécessité de séparer, entre deux types de composants :

-Les composants métiers :qui représentent les composants utilisés, lors de la réalisation de la partie métier d'une application, ces composants peuvent être simples ou composés. Leur composition sera effectué dans notre travail, selon le modèle de composants de l'approche intégrée IASA;

- Les composants visuels :qui représentent les composants utilisés, lors de la spécification de la partie interface homme machine, et qui déclenchent les services des composants métier. Ces composants seront adaptés aux tâches associées aux composants métier. Nous allons considérer dans notre travail, qu'un composant métier peut être associé à un ou plusieurs composants d'IHM.

Malgré cette distinction, il faut rappeler que les composants visuels, ne doivent en aucun cas remettre en cause les principes de l'approche IASA, notamment l'organisation interne des composants. Le composant visuel sera ainsi un composant IASA spécifique. C'est d'ailleurs ce

type d'approche qui est suivi en orienté objet, ou les composants visuels sont supportés par des classes, ayant une organisation propres. A titre d'exemple, dans java, les composants visuels de la bibliothèque javax.swing sont supportés par des JavaBeans. Ces derniers sont des classes, ayant une structure, et un mode de définition bien précis.

4.2.2 Les niveaux d'abstraction d'un composant visuel

L'objectif de la détermination des divers niveaux d'abstraction, est en relation avec le processus d'élaboration, et de génération de code d'un composant visuel, et qui dépend dans le jargon IASA, des facteurs de déploiement d'un composant. Dans ce contexte, IASA préconise un processus d'élaboration par raffinement successif (processus iteratif) de composant (recursif), qui permettrait de guider, pour une technologie cible particulière, la définition d'un processus de génération de code particulier. A titre d'exemple, pour la génération de code, une instance d'un type de composants, à déployer comme un EJB (c'est un composant ayant un modèle particulier et dépendant du langage Java), aura un processus de génération de code, pouvant être partiellement ou totalement différent, d'une autre instance du même type de composant, à déployer comme une Servlet Java ou un Service Web.

Nous avons constaté que la communauté IHM, consacre beaucoup d'efforts, pour développer des interfaces indépendantes du support, afin de produire des IHMs de qualité, et exploitables dans divers contextes (différents systèmes d'exploitation, différents langages de programmation, différents supports tels que les Pcs, les PDAs ou encore les téléphones, etc) **[94]**. A partir de n'importe lequel de ces contextes, le développeur peut obtenir différentes variantes d'une même IHM, selon le contexte d'utilisation. Ainsi, Bandelloni, et Paternò **[95]**ont proposé dans le cadre du projet Cameleon, les étapes par lesquelles passe une IHM (*cf.* Figure4.2).

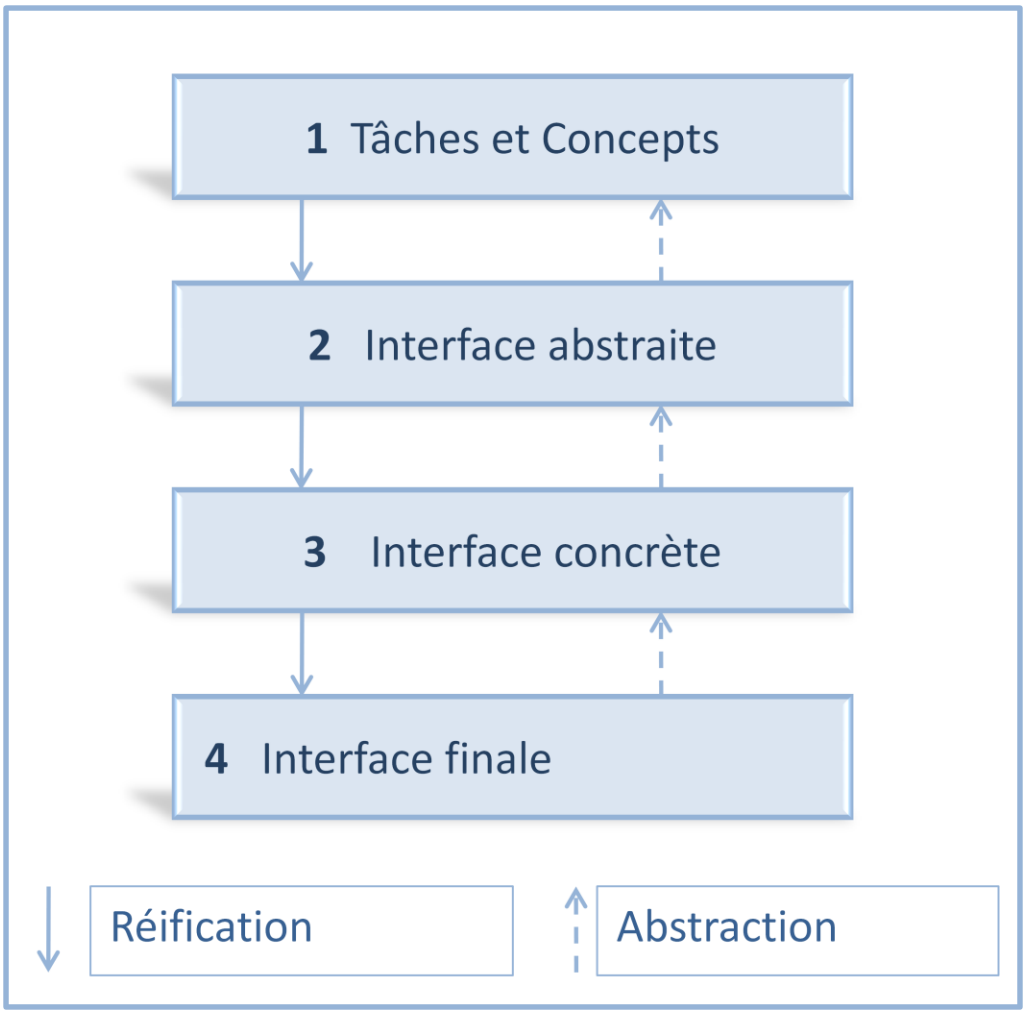


Figure.4.2 : Les quatre principales étapes de développement du projet cameleon

Le cadre de référence distingue quatre niveaux d'abstraction. Le niveau *tâches et concepts* du domaine, décrit l'IHM selon une perspectivedomaine, sans préjuger, d'une quelconque représentation. Les concepts du domaine sont les entités manipulées par les tâches. Le niveau Interface Abstraite (AUI), structure l'IHM en espaces de dialogue,et fixe la navigation entre ces espaces. A ce niveau, on reste indépendant de tout contexte d'utilisation. Le niveau Interface Concrète (CUI), affine l'IHM en introduisant les choix d'interacteurs (boutons, menus, etc.), en fonction d'un contexte d'utilisation donné (l'action « valider » peut être associée à un clic sur un bouton dans une IHM pour un poste de travail). Enfin, le niveau Interface Finale (FUI), modélise l'IHM dans une technologie particulière (JAVA-SWING, PHP-HTML, etc.), et un système d'exploitation spécifique.

Comme nous considérons une IHM comme un composant, nous allons considérer les étapes présentées ci-dessus comme étant les étapes par lesquelles, passe tout composant visuel. Ainsi, nous parlons de composant visuel abstrait, de composant visuel concret, et de composant visuel final.

4.2.3 Le processus d'élaboration d'IHM

La majorité des recherches, vise à manipuler l'interface utilisateur, sans se préoccuper de la partie fonctionnelle, comme dans le cas des UIDLs, et des modèles à composants, qui s'occupent de la partie métier seulement. A l'opposé, nous allons prendre en compte le processus métier dans notre travail, pour faire la composition d'interface homme machine, ceci est décrit ci-dessous dans notre démarche de composition (*cf.* Figure4.3).

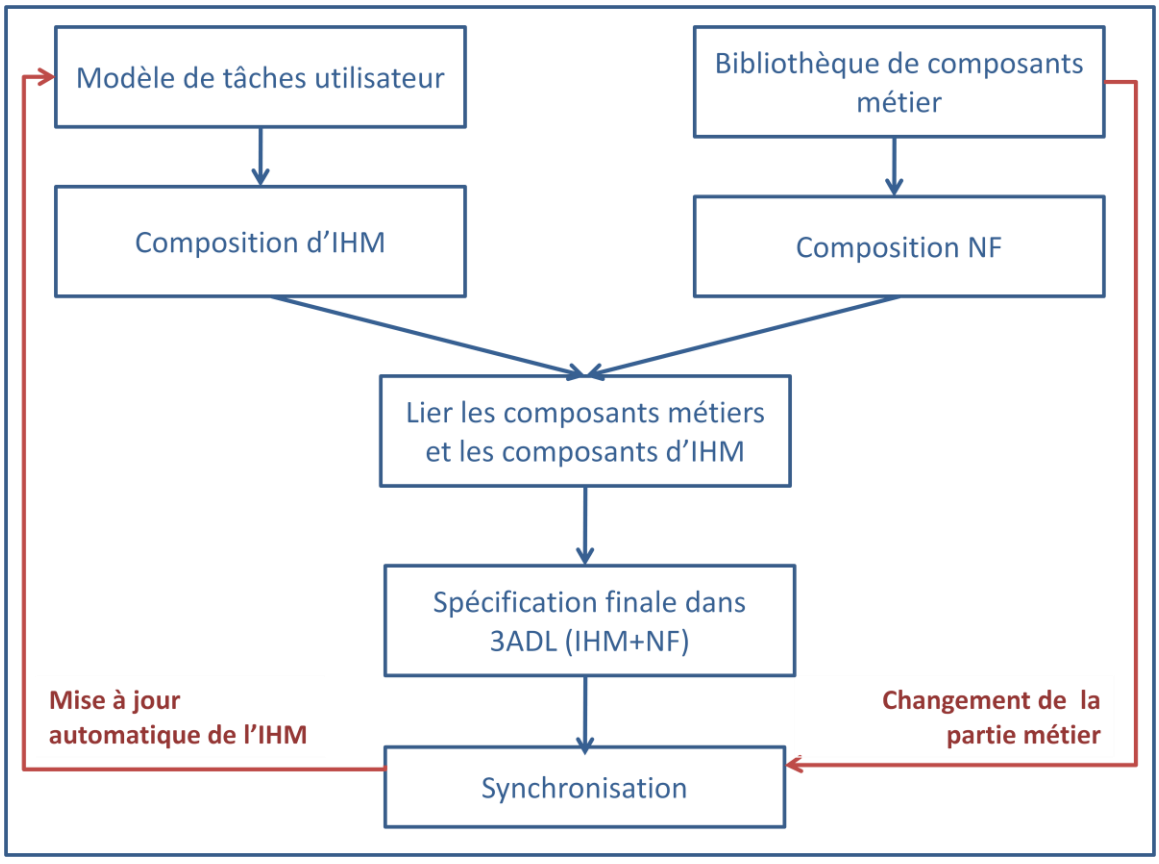


Figure.4.3 : Démarche proposée pour la spécification de systèmes interactifs

4.2.4 Autres caractéristiques

Le modèle de composant pour les IHMs sera caractérisé par les éléments suivants :

- La sémantique : le composant dans le modèle que nous allons proposer, aura une sémantique bien précise, qui pourra représenter un composant d'IHM.

- Communication avec les autres composants : la communication entre les différents composants sera assurée par le connecteur. Ce dernier est décrit de manière explicite dans l'approche IASA.
- Le modèle de composants supportera la composition hiérarchique.
- La composition au niveau de la partie IHM, doit décrire de manière implicite des tâches utilisateur. Cette propriété est importante au niveau des interfaces homme machine. C'est pour cela que la composition des éléments visuels, sera déduite à partir du modèle de tâches utilisateurs. Ainsi, nous allons la vérifier à la fin de ce chapitre.

4.3 Modèle de composants pour les interfaces homme machine

Le modèle de composant d'IHM que nous proposons, distingue entre deux catégories de composants: Les composants primitifs, et les composants composites. De ce fait, l'objectif de l'élaboration d'une IHM, consiste à réaliser un composant composite, et une IHM pourrait être considérée comme composant à part entière.

Comme dans le modèle de base de l'approche IASA, le modèle de composant visuel que nous avons introduit, définit une organisation de la vue externe, à laquelle doit adhérer n'importe quel composant visuel, primitif ou composite, et une organisation de la vue interne, qui est presque la même pour un primitif et un composite, sauf que le primitif ne comporte pas de partie réservée à la composition. Cette dernière est conforme à l'organisation de la vue interne du modèle de composant IASA.

4.3.1 La vue externe d'un composant visuel

La première caractéristique de la vue externe, est la spécialisation des faces d'un composant, dans sa notation graphique, sous forme de boite noire. Chaque face est ainsi associée à des types de ports particuliers. Cette approche de spécialisation des faces, a été déjà introduite dans le modèle de composant C2 (voir chapitre3).

La vue externe d'un composant visuel possède trois groupes de ports(*cf.* Figure.4.4), chacun étant associé à une des faces de la notation graphique d'un composant : les *ports*

d'évènements,les *ports de gestion d'évènement*, les ports d'accès aux attributs publiquesd'un composant visuel (couleur, position, police de caractère etc..)

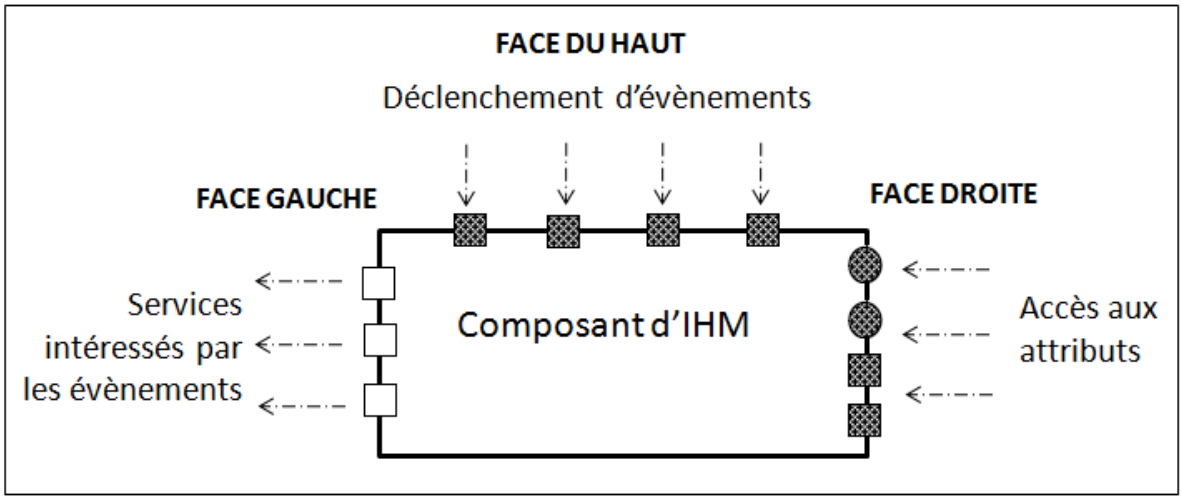


Figure.4.4: Modèle de la vue externe d'un composant d'IHM

La face du haut comporte les *ports d'évènements*.Ces ports renseignent sur les évènements,que peut reconnaitre un composant visuel. En général, l'événement est déclenché par le composant visuel lui-même, suite à une action qui lui aurait été appliqué (click de souris, presser une touche du clavier, fermer une fenêtre etc.). Ainsi, il pourrait sembler que l'exposition de ces ports est en trop dans le modèle. En fait, l'exposition de ces ports dans la vue externe, permettra un déclenchement programmé des événements, à partir d'autres composants connectés au composant visuel, à travers ces ports d'événement, et non pas suite à une action physique au niveau de l'IHM. Le nombre de *ports d'évènements* dans un composant d'IHM, dépend du nombre d'évènements que ce dernier peut reconnaitre. Les *ports d'évènements* d'un composant visuel,sont tous du type *ServerPort*d'IASA. Le point d'accès de service de ce port, fonctionne en mode asynchrone.

La face gauche contient les *ports de gestion d'évènements.* C'est à travers cesports, que seront déclenchés les services, associés à la logique de prise en charge d'un événement. Le nombre de ports nécessaires sera décidé, lors de l'instanciation du composant d'IHM. Les ports de *gestion d'évènements* sont des types *ClientPort et DataPort*.

Enfin, le dernier groupe, composé de *ports* d'accès, est associéà la face droite. Ces ports permettent l'accès aux spécificitésdu composant visuel (accès aux attributs). Le nombre de ports

d’accès dépend des attributs, et actions du composant visuel associé. Ce type de ports est représenté par des *DataPort,*et des *ServerPort.*

Le diagramme de classe de la vue externe d’un composant visuel, est représenté comme suit (*cf.* Figure.4.5) :

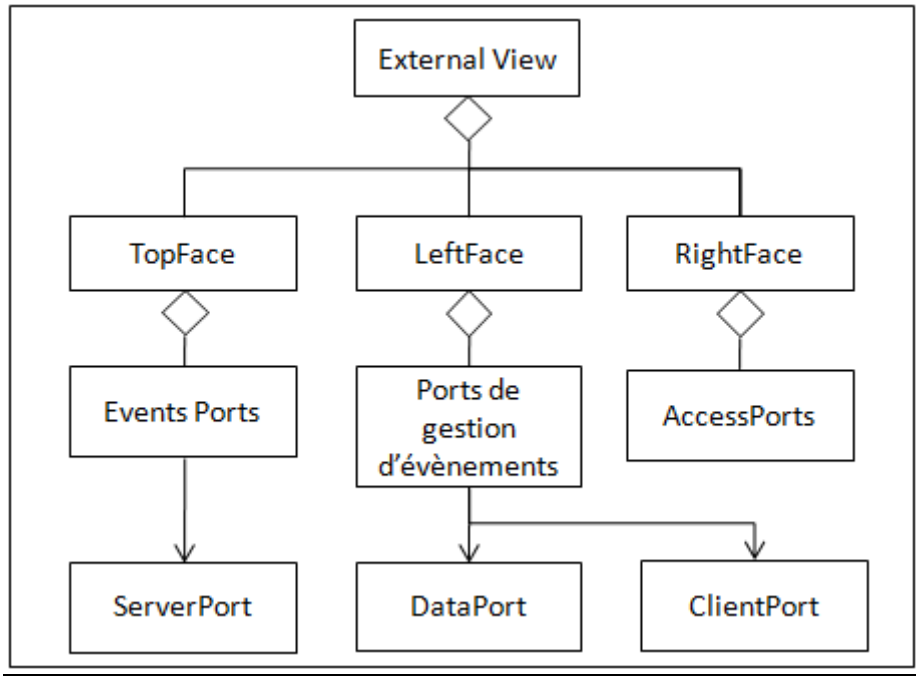


Figure 4.5 :Diagramme de classe de la vue externe d'un composant d’IHM

4.3.2 La structure interne du composant

La vue interne de notre modèle de composants, conserve l’organisation préconisée par l’approche IASA. Nous retrouvons les deux parties du modèle de composant de IASA, à savoir la partie opérative, et la partie contrôle (*cf.* Figure.4.6). C’est dans la partie opérative,qu’est spécifiée la structure d’une IHM complète. La partie opérative contient diverses instances de composants visuels interconnectés, représentant toute ou une partie de l’IHM, à un moment bien précis.

La partie opérative contient obligatoirement, un composant comportemental spécifique appelé*VisualCmpController*. C’est à travers les ports de ce composant, qu’il est possible de spécifier par des connecteurs, les opérations de modification d'attributs, et d'actions sur le composant d’IHM concret (i.e. activation, visualisation, déplacement).

Quand à la partie contrôle, elle n'est pas modifiée. Elle conserve sa tache définie dans IASA. Elle est constituée d'au moins un composant qui est du type *OpPartController*. La partie permet de réaliser les diverses opérations de contrôle, sur les composants de la partie opérative. Le composant *OpPartController*, se charge essentiellement des opérations d'initialisation des composants d'IHM.

La partie opérative est organisée en deux parties :

- Une partie qui interagit directement avec le composant visuel associé.
- Unepartie représentant les instances de composants rentrant dans la définition structurelle du composant visuel. En général, un composant visuel n'est composé que par d'autres composants visuels.

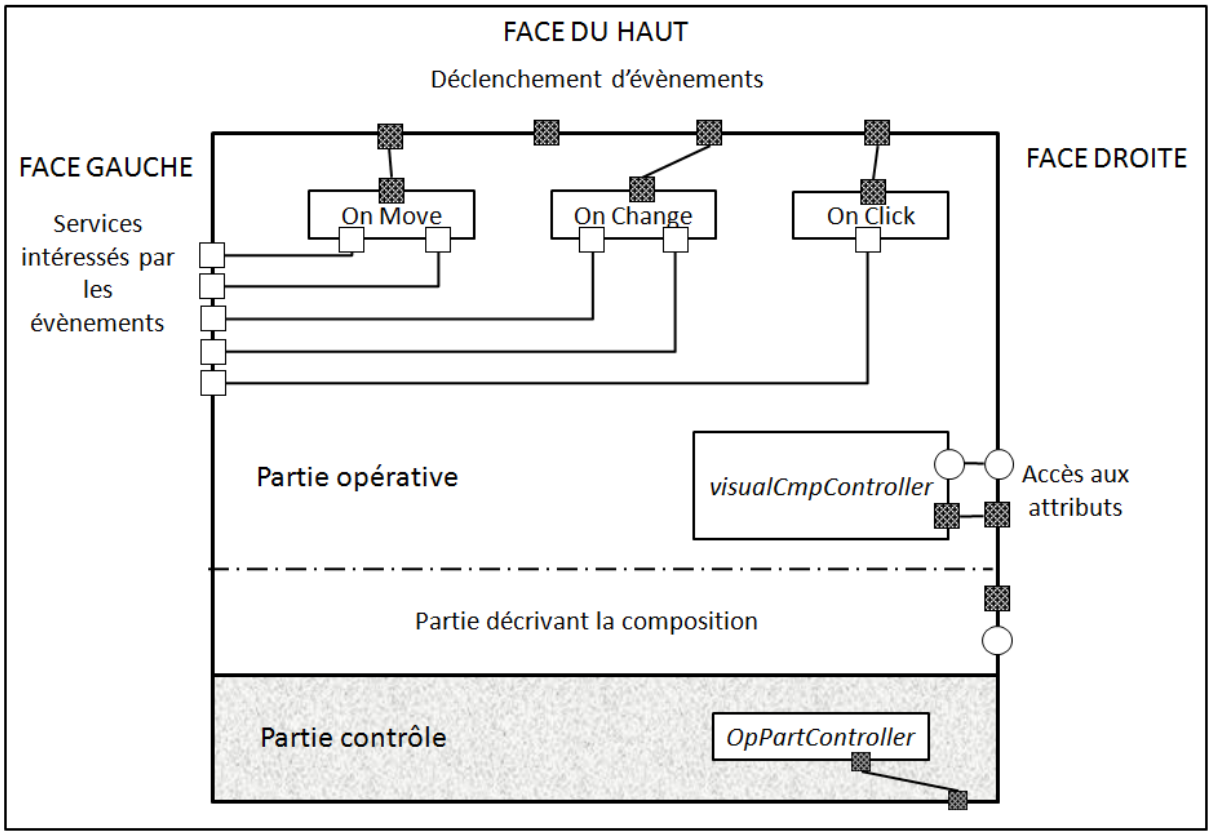


Figure.4.6: Modèle de la vue interne d'un composant d'IHM

Un composant primitif possède la même structure que celle du composite, sauf qu'il ne contient pas de partie réservée à la composition.

Le diagramme de classe de la vue interne d'un composant visuel, est représenté comme suit (*cf.* Figure.4.6) :

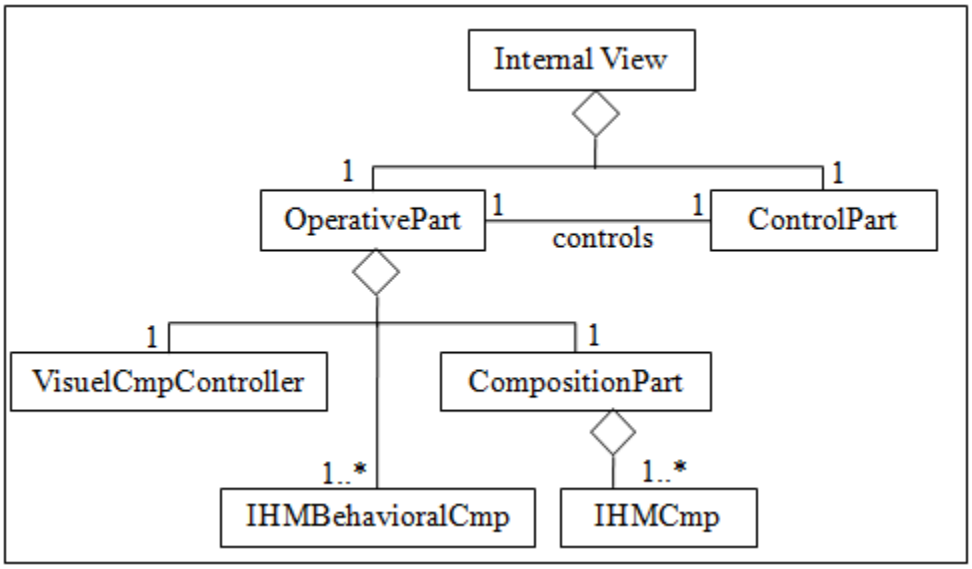


Figure.4.7: Diagramme de classe de la vue interne d'un composant d'IHM

4.3.3 La gestion des évènements

La gestion des événements au niveau de l'architecture d'une IHM, permet de construire une interaction conviviale entre l'utilisateur, et le programme. Les événements proviennent souvent de l'appui d'une touche au clavier, du clic de la souris ou du mouvement de celle-ci. Un évènement peut aussi provenir d'un autre composant dans l'architecture.

L'évènement est l'information qui est issue de la source, à destination du consommateur de l'événement, qui permet de déclencher une action.

L'affectation des événements aux services intéressés, décrite dans l'ADL 3ADL, est prise en charge par des composants comportementaux, de la partie interne des composants d'IHM. On associe à chaque événement, un composant comportemental (*cf.* Figure.4.8) qui possède des écouteurs (*listners*), ces derniers se chargent d'écouter s'il y'a un éventuel évènement. Les écouteurs sont aussi le principe de gestion d'évènements, du modèle JavaBean. Il est possible d'avoir plusieurs ports clients pour un même événement. Ces ports seront activés selon la logique du composant comportemental associé à l'événement.

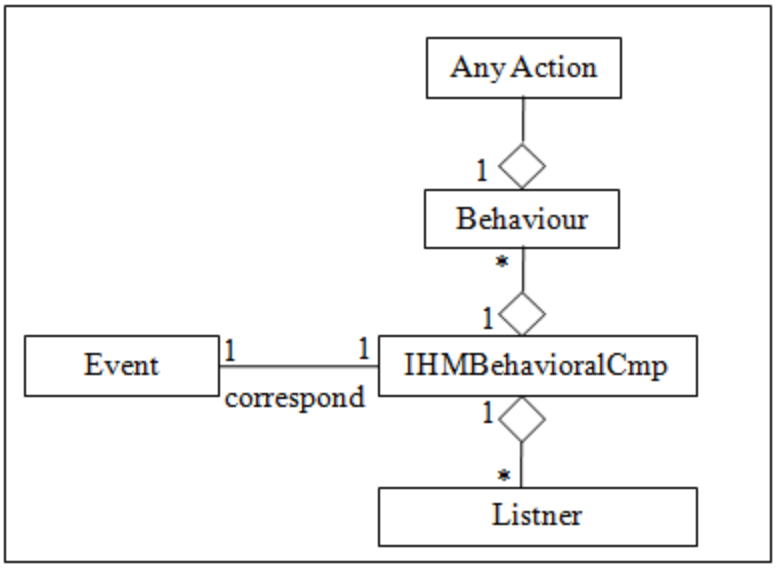


Figure.4.8 : Diagramme de classe d'un composant comportemental, faisant partie d'un composant d'IHM

4.3.4 Les composants primitifs dans le composite

Dans le modèle de composants proposé, les composants internes d'un composite, interagissent directement avec les composants métier, et peuvent agir sans faire intervenir le composite (déclenchement de services,..) (*cf.* Figure.4.9). Ils peuvent aussi interagir avec le composite. Le composant comportemental *visualCmpController*du composite, peut gérer les attributs des composants primitifs, et effectuer des actions sur ces derniers (show, hide,…).

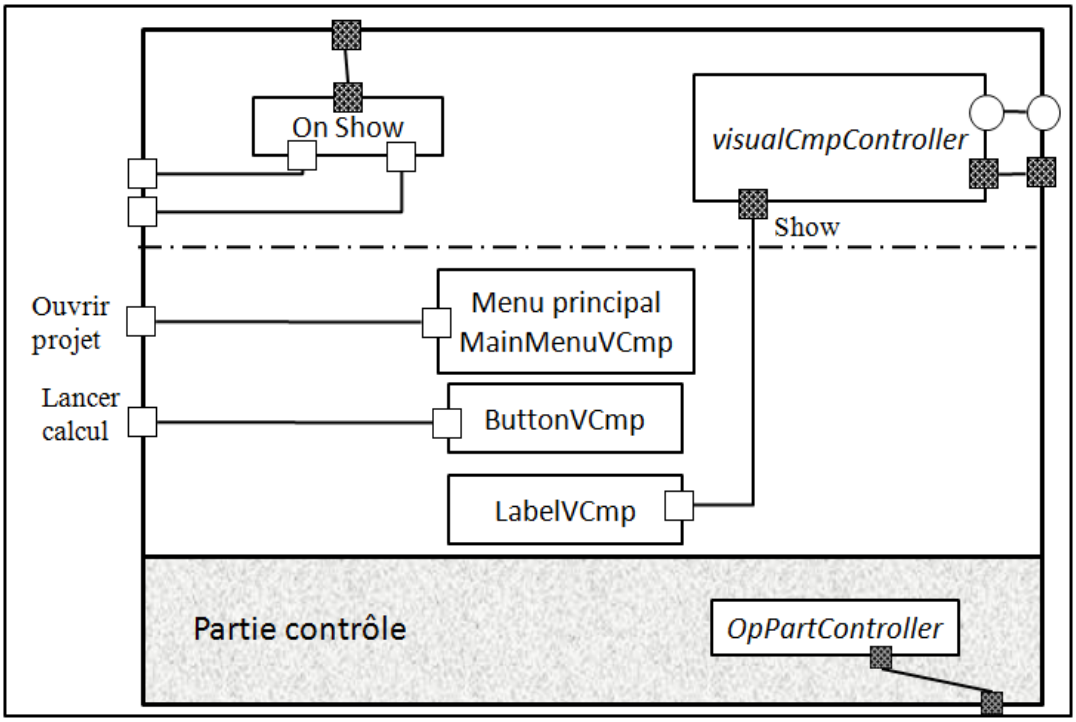


Figure.4.9: Exemple de composant composite

4.4 Liaison entre les composants métiers et les composants visuels

L'approche IASA permet de décrire la partie métier d'une application, dites aussi le noyau fonctionnel (NF). En effet, IASA s'appuie sur la description des différentes entrées, sorties, et actions disponibles. Nous proposons dans le contexte de l'approche IASA, la description de la partie IHM de l'application, en s'appuyant sur les mêmes paramètres (entrées, sorties et actions), sauf que celles-ci possèdent des significations différentes, selon que l'on décrit le NF ou l'IHM. Dans le cas de la description du NF, les entrées (resp. sorties) correspondent aux différentes entrées (resp. sorties) des opérations disponibles. Les opérations disponibles sont quant à elles fournies par les "actions". Tandis que pour la description de l'IHM, les entrées correspondent aux éléments sur lesquels l'utilisateur peut agir, pour fournir de l'information au NF, les sorties correspondent aux retours du NF, et enfin les "actions" sont des éléments qui permettent de déclencher l'appel d'une opération d'un service.

Afin de conserver une cohérence entre la partie métier (le noyau fonctionnel), et la partie IHM d'une application, il est important de décrire clairement dans le contexte de l'approche, les liens qui existent entre l'IHM et le NF. Ces liens décrivent aussi bien les données qui transitent entre les deux niveaux, que les événements qui vont déclencher les appels aux services du NF.

La liaison entre le NF et l'IHM, est assurée grâce au concept du connecteur de l'approche IASA. Nous utilisons dans notre travail, le type de connecteurs prédéfini dans IASA, à savoir les connecteurs de transport. Les connecteurs de transport s'occupent du transport des données, et d'évènements. Ils sont composés uniquement de deux rôles, et ne permettent de spécifier que des topologies point à point, permettant de lier deux ports, ou deux points d'accès. Ce type de connecteur se révèle suffisant à ce niveau, pour la spécification d'IHM, ainsi que pour la liaison du NF avec la partie IHM. Le diagramme de classe suivant (*cf.* Figure.4.10), montre les rôles associés au connecteur utilisé :

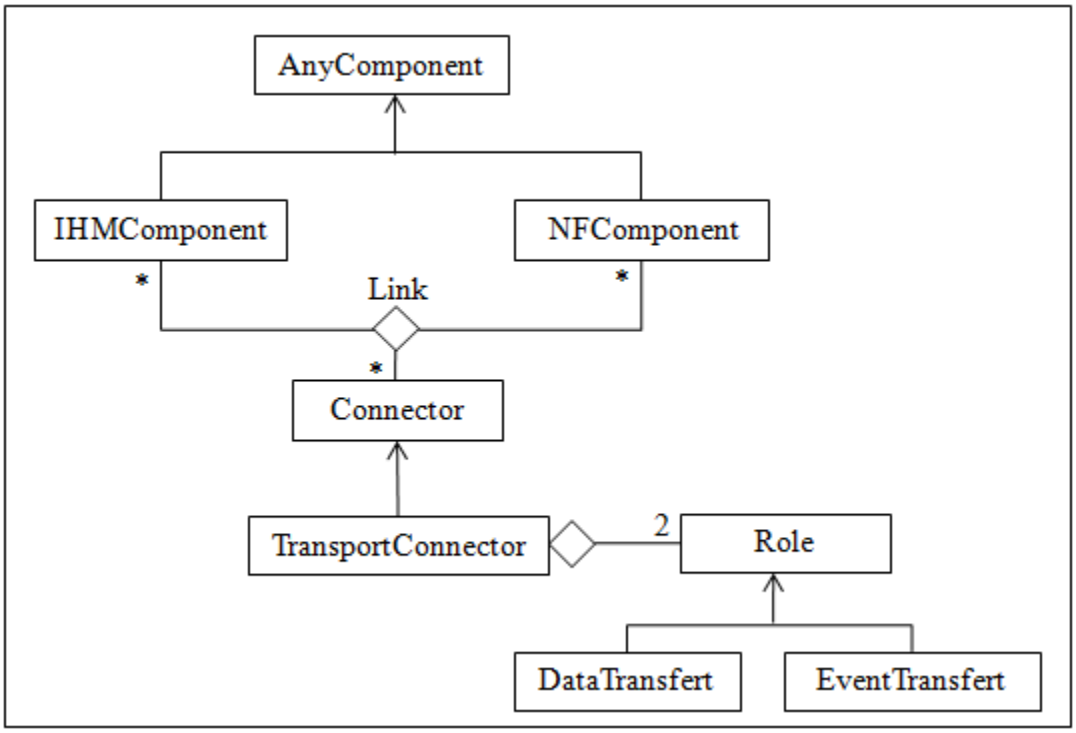


Figure.4.10 : Diagramme de classe montrant la liaison entre un composant métier et un composant d'IHM

Les connecteurs de transport que nous utilisons, ne nous permettent pas seulement de décrire les liens de données qui existent entre l'IHM et le NF, mais également les liens d'évènements. La Figure.4.11 (*cf.* Figure.4.11) illustre sur une représentation à composants, de la partie IHM, et de la partie métier, ainsi que les liens entre les deux. Notre modèle disjoints les flots de données, et d'événements, ceci rend la présentation schématique plus lisible. L'envoi de toutes les données doit se faire avec l'appel du service, ou de l'opération, tous les attributs doivent être affectés avec l'appel.La transmission effective d'une donnée, se fait dans le contexte

du déclenchement d'une action. Celle ci pourrait être une action pure de transfert de données, (envoyer, recevoir) ou à une action engendrant la réalisation d'un service particulier.

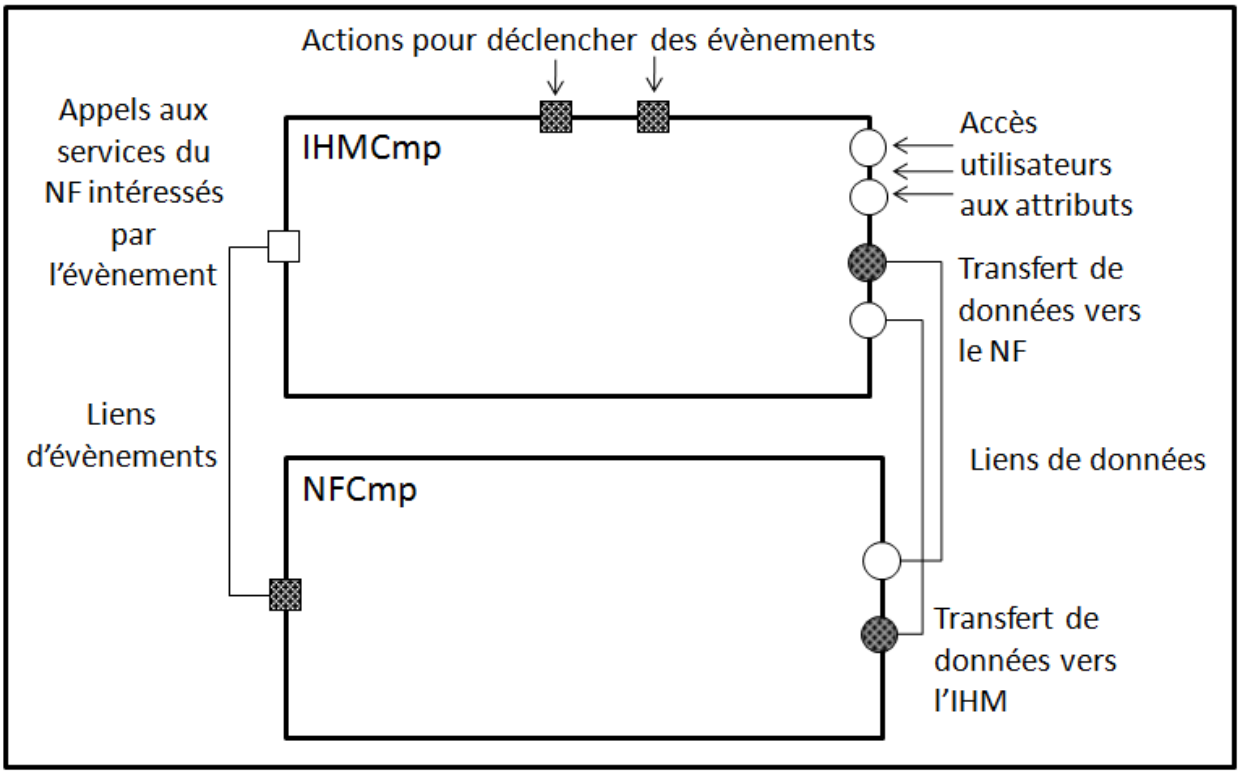


Figure.4.11 : Composants NF et IHM et liens entre les deux

La description des liens, entre les composants métier, et composants d'IHM, nous donne la possibilité d'ajout, et de retrait de composants d'IHM, correspondants aux évolutions fonctionnelles.

Il est important de noter que lorsqu'on veut faire la spécification d'un système quelconque, la notion du niveau d'abstraction est importante. Le niveau de granularité du composite « noyau fonctionnel » et celui du composite « IHM » n'est pas forcement le même, selon les besoins. De ce fait, il est possible d'ajouter des ports de délégation sur le composite, pour montrer le lien avec le noyau fonctionnel, sans montrer schématiquement le composant réel qui communique.

4.5 Description textuelle de l'architecture de l'IHM

La description architecturale de l'interface homme machine se fera à l'aide de 3ADL. Un langage de description d'architecture logicielle, défini dans le contexte de l'approche IASA, et étendu dans notre travail pour prendre en charge les interfaces homme machine, ainsi que les liens entre la partie IHM et la partie métier.

Le langage 3ADL nous permet de décrire l'architecture de l'interface homme machine de façon abstraite (*cf.* Figure.4.12), indépendamment de toute technologie et de tout langage de programmation, cela va nous donner la possibilité de projeter la description dans différents langage de programmation, et donc de concevoir des IHMs indépendantes du langage.

Pour permettre à 3ADL de prendre en charge la description des IHMs, nous proposons de factoriser en des concepts très simples, les principes fondamentaux des IHMs conventionnels. L'extension de 3ADL sera fondée sur un nombre de composants fondamentaux, qui sont les composants graphiques (widgets) connus. Chaque composant visuel est instancié d'un type bien précis appartenant à 3ADL. On peut différencier les composants visuels en deux catégories : les composants visuels composites comme les fenêtres, et les composants visuels primitifs comme les boutons. Les composants visuels composites peuvent contenir d'autres composants visuels. Le principe de base est de composer ces différents éléments ensembles,pour obtenir de nouveaux composants composites. Pour décrire son interface graphique, il suffit de composer dans le bon ordre, les bons composants visuels. Cette définition initiale doit évoluer pour construire des IHMs plus complexes.

Chaque composant d'IHM est associé à un type de composant, le représentant dans une spécification architecturale textuelle. A titre d'exemple, un bouton (i.e. le *JButton*de Java/Swing ou *Button*d'Eclipse SWT) est représenté dans l'approche intégrée par le type de composant *ButtonVCmp,*et une fenêtre est représentée par le composant *FormVCmp.* Il est à noter que chaque type de composant possède:

- Un nombre fixe de *ports d'évènements*,de type *ServerPort*(ports de la face haute du modèle), représentant chacun un événement pouvant se produire sur le composant visuel.

- Un nombre fixe de ports d'accès, de types *DataPort,*et *ServerPort,* associés aux attributs du composant visuel, et à ses méthodes.

- Un nombre variable des ports de gestion d'évènements, de type *ClientPort* (les ports de la face gauche du modèle). Cette variation est du à l'environnement d'instanciation du composant visuel. Sur une même fenêtre, un même composant visuel peut être instancié plusieurs fois. Chaque instance peut correspondre à une logique bien précise, dans laquelle le nombre de port de la face gauche peut être différent.

```
package nomDePackage;
import listeDePackage // package de composant et de connecteurs
component nomDuTypêDeComposant {
accesspoint{ // Définition de types internes de points d'accès de données }
port { // Définition de types internes de port }
connector{ // Définition de types internes de connecteurs. }
component { // Définition de types internes de composant.}
/// Définition des instances
ports{ // définition des ports du composant
}
operativepart{ // description de la partie opérative:
// instance de composant, type  de composant, et connexions
}
controlpart{// description de la partie contrôle: instance de composant et connexion inter
// composant de la partie contrôle et avec les composants de la partie opérative
}
behavior{ // Comportement du composant; Ce comportement représente
// celui du composant OpPartControllerde la partie control
}
properties{ // Spécification des propriétés non fonctionnelles. Pour l'instant, seules les
// sont décrites les informations de déploiements
}
} // Fin description type de composant
```

Figure.4.12: Les clauses de la description textuelle d'un composant

Par défaut, le nombre de ports de gestion d'évènements, est égal au nombre d'événements. Chaque port client (face gauche) correspond à un port de service (face haute). Le comportement par défaut du composant comportemental, liant un port d'évènement à un port de

gestion d'évènement, consiste à déclencher de manière séquentielle tous les services qui lui sont connectés. Par défaut, le composant comportemental possède un seul port client, connecté au port client correspondant situé sur la face gauche.

4.6 Les niveaux d'abstraction d'un composant visuel

Comme nous l'avons déjà présenté dans ce chapitre, un composant d'IHM est un type. Il possède une vue externe et une autre interne, et est crée pour but d'offrir une structure, et un comportement en même temps. Les composants d'IHM sont prévus pour interagir avec les composants métier. Ces deux types de composants sont liés grâce aux connecteurs. Il existe deux types de liens, liens de données et liens d'évènement.

Un composant d'IHM tel que nous l'avons définit, peut être soit abstrait, soit concret, ou finale. En suivant la démarche que nous avons proposée au départ (*cf.* Figure.4.2), un composant visuel est abstrait au départ.

Le niveau abstraitcorrespond à celui de l'interface abstraite, il est construit à partir du modèle de tâche utilisateur (*cf.* Figure.4.13). La spécification architecturale à ce niveau, comprend les composants d'IHM abstrait, déduit à partir du modèle de tâche. L'objectif, dans cette étape, est de transcrire un modèle abstrait, en une spécification d'architecture logicielle d'une IHM abstraite. Pour arriver à cela, il suffit d'associer des composants composites (de type FormVCmp par exemple) à la tâche principale, et aux sous-tâches mères. L'attribution d'un composite comme PanelVcmp, peut se faire pour les sous-tâches mères, mais pas toutes, ceci est un choix de l'architecte des logicielles. Enfin, des composants primitifs sont associés aux tâches interactives terminales (*cf.* Figure.4.14). Ainsi, les composants primitifs seront placés dans les composites, associés à leurs tâches mères. S'il se trouve qu'il y'a des composants composites inutiles, une suppression peut être effectuée.

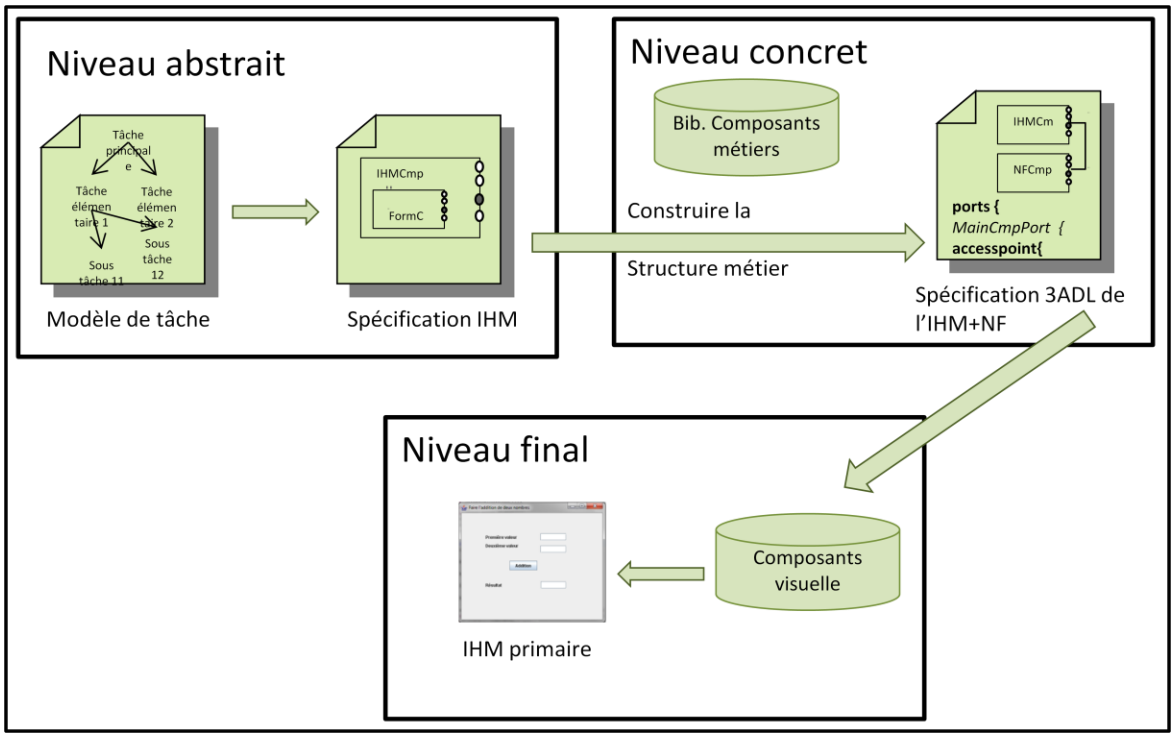


Figure.4.13: Les trois niveaux de spécification d'IHM

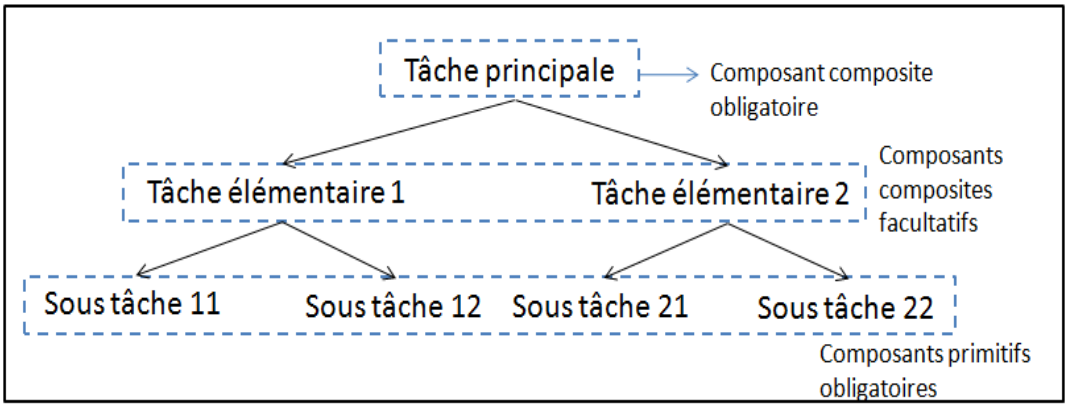


Figure.4.14: Exemple de modèle de tâche

La réification de l'IHM, qui est le passage au niveau concret,passe par la construction de la structure métier. Ce niveau est équipé d'une bibliothèque de composants métier, qui ont été élaborés avec l'approche IASA. Les composants visuels, et les composants métier seront liés à l'aide des connecteurs (*cf.* Figure.4.15).Dans cette étape, l'IHM et la partie métier sont décrit à l'aide du langage de description 3ADL, et à l'aide de la notation graphique d'IASA.

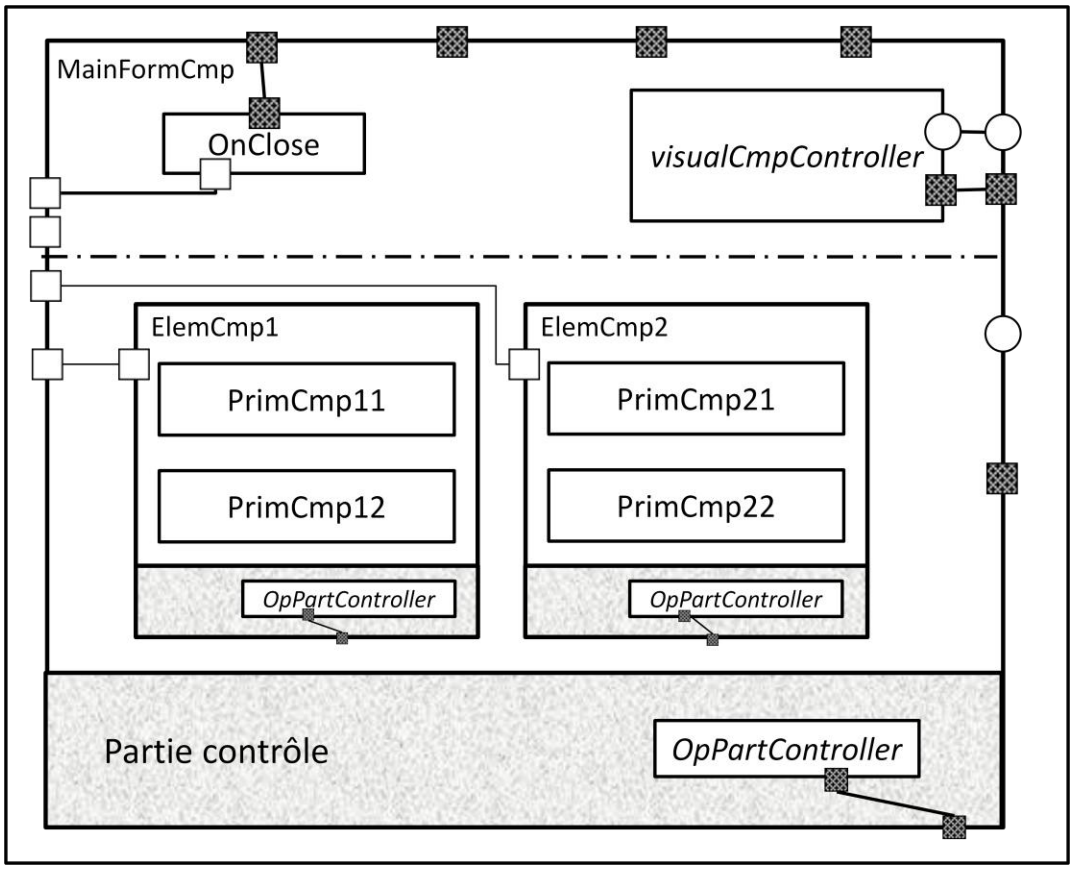


Figure.4.15: Modèle de composants décrivant implicitement les tâches utilisateur

Le niveau final sera obtenu, par le choix des composants de présentation qui sont les composants visuels concrets, et donc le choix d'une technologie précise pour avoir l'IHM finale. L'avantage de cette démarche, est que l'architecture peut être projetée vers plusieurs technologies.La projection consiste à parcourir tout les composants d'IHM, et associer chacun à un élément graphique, de père en fils, dans une technologie particulière.

4.7 Conclusion

Nous avons présenté dans ce chapitre, notre contribution, à savoir, la proposition d'un modèle de composants, pour la spécification d'architecture d'une IHM. Nous avons tout d'abord commencé, par aborder les problèmes de composition d'IHM, dans le monde des composants, pour motiver notre approche. Nous avons ensuite guidé notre travail, par des propositions, et une démarche générale, tout en exploitant les résultats de notre état de l'art.

Nous avons ensuite présenté une ébauche de solution, permettant de composer des IHM, de la même manière qu'une composition du NF. Cette solution permet de décrire à la fois l'IHM, ainsi que les liens entre l'IHM et le noyau fonctionnel.

Nous avons vérifié à la fin que le modèle proposé, décrit de manière implicite l'enchaînement des tâches utilisateurs. Notre démarche nous a permis d'arriver à la spécification d'un système interactif, avec ses deux parties IHM, et métier. Ce modèle mis en placepermet de réconcilier le monde de l'IHM, et du métier enproposant une vision commune, de ces deux niveaux.

CHAPITRE 5
IMPLEMENTATION DU MODELE DE COMPOSANTS

5.1 Introduction

Au cours du chapitre précédent, nous avons présenté un modèle pour la spécification d'IHM selon une approche d'architecture logicielle. Face à la difficulté d'évaluer quantitativement les propositions liées à notre mémoire, IASASTUDIO permet de concrétiser l'ensemble des propositions autour du modèle proposé. Nous allons donc dans ce chapitre, proposer une réalisation, et mener une expérimentation autour de notre proposition. Ce chapitre est divisé en deux parties :

- Nous allons présenter d'abord notre réalisation. Le logiciel IASASTUDIO a été amélioré, et réorganisé en plusieurs vues. Ces dernières serviront à l'élaboration de l'architecture des systèmes interactifs.

- Enfin, nous allons dérouler un exemple complet de spécification d'architecture. Nous allons suivre la démarche proposée dans le chapitre précédent. La composition sera déduite à partir du modèle de tâche utilisateur. Tandis que pour la construction de la partie métier, nous allons utiliser des composants métier déjà définit avec IASA.

5.2 Présentation d'IASASTUDIO

Nous avons organisé l'outil IASASTUDIO en trois vues (*cf.* Figure5.1), chacune d'elles communique avec les deux autres. Tout changement effectué sur une vue, se répercute systématiquement sur les autres, ceci est nécessaire pour garder la cohérence entre les différentes vues d'un même système.

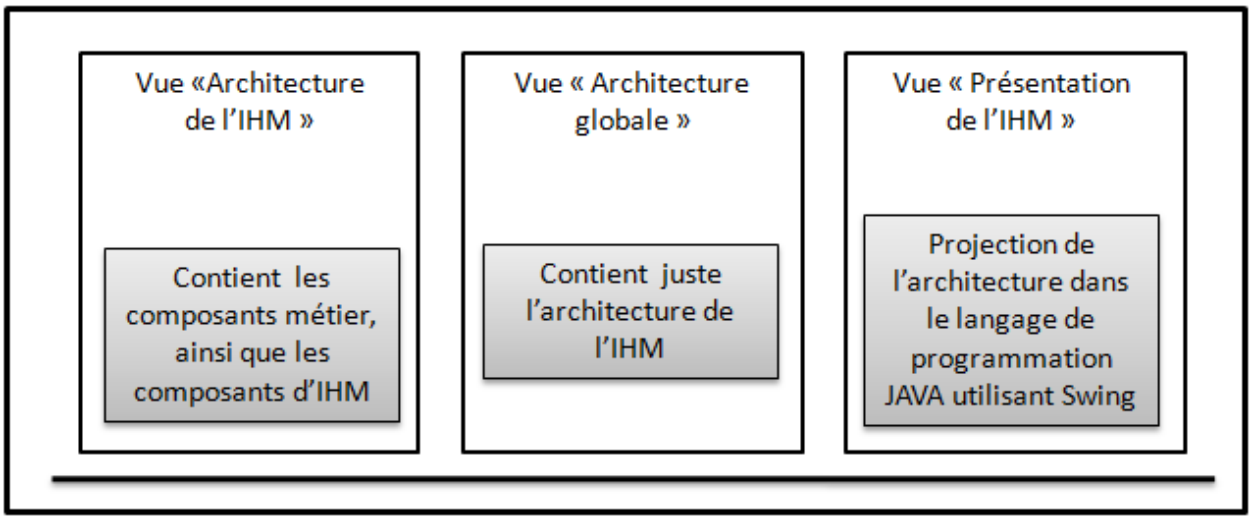


Figure 5.1 : Les trois vues d'IASASTUDIO

L'espace de travail d'IASASTUDIO (*cf.* Figure.5.2), est composée des éléments suivants :

- Un *menu principal*, qui permet d'ouvrir un nouveau projet, ou un projet existant, d'enregistrer un projet, de supprimer, de copier, et de coller des composants sélectionnés. De plus, ce menu permet de entre les deux modes de spécification d'architecture, qui sont le mode *textuel* et le mode *graphique*.

- Un *panneau graphique*, qui contient des raccourcis vers les fonctionnalités les plus importantes du menu principal.

- Une *barre d'outils*, qui permet de choisir un élément d'architecture, et de le dessiner dans la *feuille de spécification*.

- Un *panneau de navigation*, qui regroupe tout les éléments d'une architecture, et qui sont présents dans différentes vues. Ce panneau permet d'obtenir une vue arborescente des différents éléments, toute sélection dans l'arbre permet choisir les objets équivalents sur les autres vues, ceci nous permet de garder la cohérence entre les différentes vues.

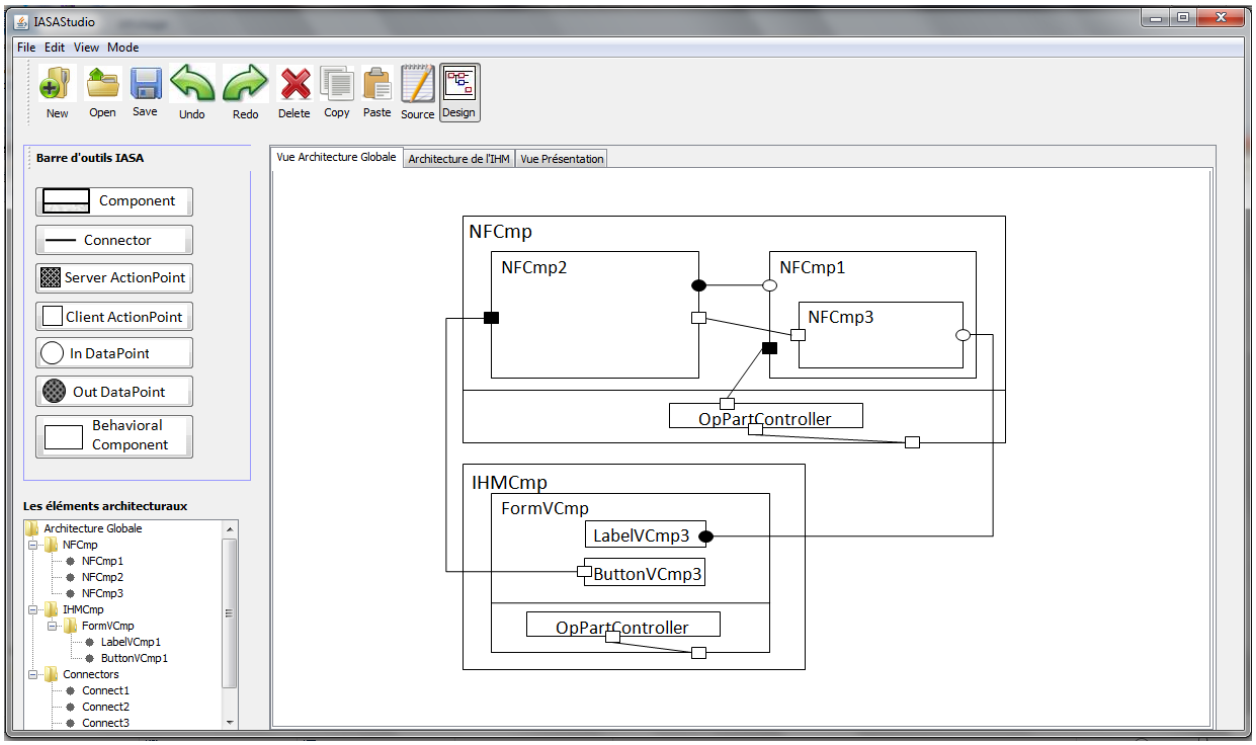


Figure.5.2 : Copie d'écran d'IASASTUDIO en mode design de la vue « architecture globale »

- Une *feuille de spécification*, elle englobe le diagramme de composants et de connecteurs d'un système, si le mode sélectionné est le mode *graphique*, elle contient sinon la spécification de l'architecture en 3ADL, et c'est le mode *textuel* qui est activé dans ce cas. Une feuille de spécification est associée à un onglet, chacun correspond à une vue parmi les trois.

- Un onglet « *architecture de l'IHM* », qui permet à l'utilisateur la spécification des différents composants d'IHM interconnectés, et donc de montrer la structure de l'interface. Les composants présents dans cette vue sont les composants *abstraits*, car ils sont indépendants de la structure métier.

- Un onglet « *architecture globale* », qui peut comporter les différents types de composants, composants métier, les composants visuels, ainsi que les connecteurs. Les composants d'IHM présents dans cette vue sont dits concrets, car à l'inverse de l'onglet précédent, ils sont associés aux composants métier.

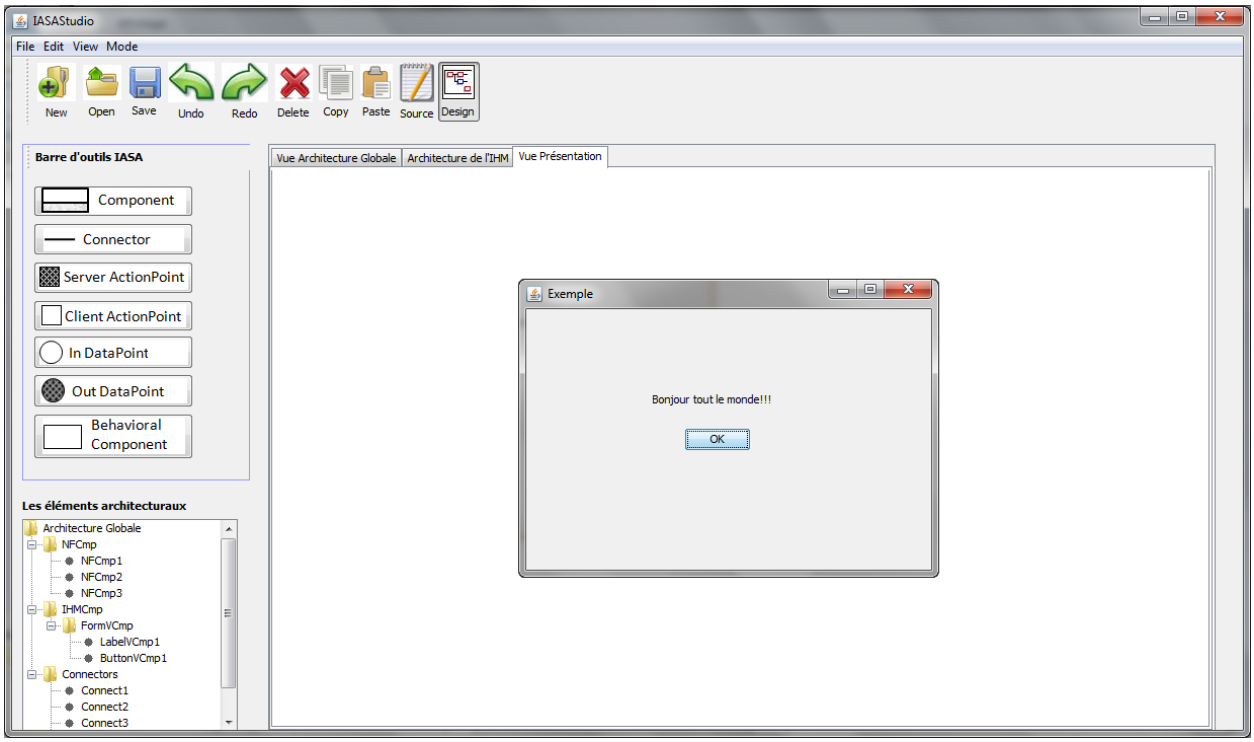


Figure.5.3 : Copie d'écran d'IASASTUDIO en mode design de la vue « *Présentation* »

- Un onglet « *présentation de l'IHM* », qui est la projection de la spécification précédente dans un langage de programmation, donnant lieu à une IHM *« finale »*, pour notre cas, nous avons choisi le langage *JAVA*, et la boite à outils *Swing* comme cibles. Il est à noter que la cohérence entre les deux vues architecture et présentation est assurée.

L'avantage principal de cette organisation, est que le nombre d'onglets (de vues) peut augmenter plus tard, permettant la spécification de l'architecture selon plusieurs ongles.

5.3 Exemple d'élaboration d'architecture logicielle

Notre travail a été évalué dans le cadre de la réalisation d'une application interactive simple, dont le but est l'implémentation de la fonction mathématique suivante :

$$f(x, y, z) = (x + y) * z$$

Nous allons dans ce qui suit, suivre la démarche proposée dans le chapitre précédent, pour la spécification de la partie métier de l'application, ainsi que pour la partie IHM.

5.3.1 Modélisation de la tâche utilisateur

Cette étape constitue la première dans l'élaboration de l'architecture d'une IHM. C'est à partir du modèle de tâche que la composition d'éléments visuels se fera. Il est clair que pour l'évaluation de la fonction mathématique *f*, l'utilisateur finale fera entrer les valeurs de x, de y, et de z, lancera le calcul, et récupérera le résultat (*cf.* Figure5.4).

Pour les tâches de saisies de valeurs, et de récupération de résultats, nous

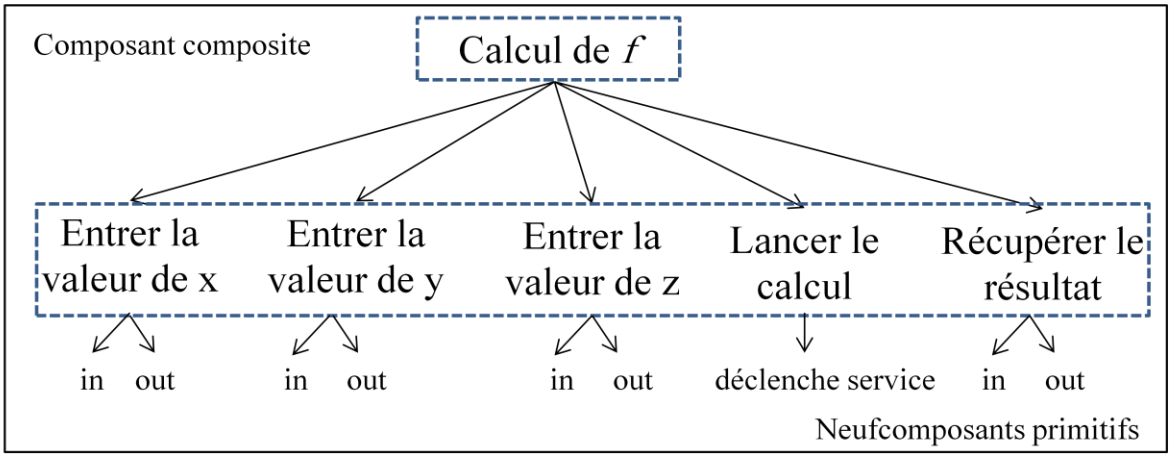


Figure.5.4 : Modèle de tâche pour le calcul de *f(x,y,z)*

5.3.2 Spécification de l'IHM abstraite

L'IHM abstraite est déduite à partir du modèle de tâche utilisateur. La tâche utilisateur dont le but est la saisie d'une valeur, sera exprimée par deux composants primitifs, l'un représente un champ de saisie, tandis que l'autre comporte une étiquette sur ce dernier. Pour notre exemple, nous avons un seul composant composite, et neuf composants primitifs. Les trois tâches de saisie, ainsi que la tâche de récupération du résultat, seront représentées par les composants *LabelVCmp* et *EditVCmp*. Quand à la tâche de lancement de calcul, elle sera représentée par le composant ButtonVCmp.

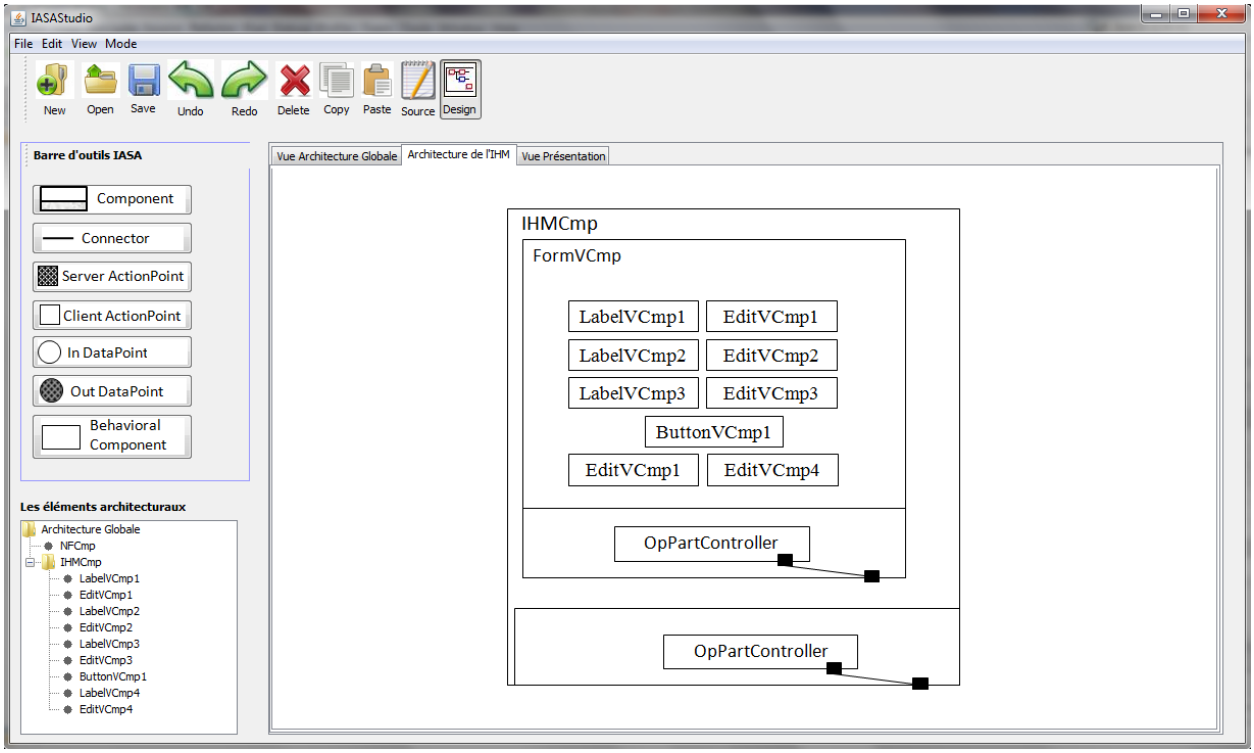


Figure.5.5 : Spécification de l'IHM abstraite pour la tâche de calcul de *f*

5.3.3 Spécification de l'IHM concrète

L'IHM concrète est obtenue par la construction de la structure métier de l'application, la liaison entre les composants d'IHM, avec les composants métier est établie à ce niveau.

Pour l'évaluation de la fonction *f,* nous avons besoin de deux composants métier: *AddFCmp,* et*MulFCmp,* le premier effectue l'addition, et le second réalise la multiplication. Le composite métier *NFCmp*comportera ces deux composants, et se partie contrôle effectuera la coordination entre eux.

Un composant d'IHM, comporte toutes les fenêtres de l'application, sa partie *contrôle* effectue la gestion, et la coordination des flux de données et d'évènements, entre les différentes fenêtres, ainsi qu'avec les composants métier. Quand à la partie *contrôle*d'une fenêtre, elle est chargée de gérer les flux entre les différents composants de cette dernière, et communique aussi avec la partie contrôle du composant d'IHM.

Pour notre exemple, le composant d'IHM *IHMCmp* ne comporte qu'une seule fenêtre *FormVCmp,* cette dernière regroupe les neuf composants primitifs. Toutes les instances de

LabelVCmp sont statiques, car ils jouent le rôle d'étiquette seulement, quant aux instances d'*EditVCmp*, et de *ButtonVCmp* elles possèdent un comportement.

Les trois premières instances d'*EditVCmp*permettent à l'utilisateur de saisir trois valeurs, le composant comportemental de la fenêtre, appelé *VisualCmpController*,effectueen ce moment un changement dans l'attribut *text* des trois *EditVCmp*, ces trois valeurs doivent être transmises à la partie métier, qui va effectuer le calcul, donc un port *out* est associé à chacun des composants.

Le composant *ButtonVCmp*est demandeur de service, lorsque l'utilisateur clique dessus, une demande de calcul de la fonction *f* est effectuée, la partie contrôle de la fenêtre, appelé *FormVCmp* reçoit la demande, ainsi que les trois valeurs, et transmet cette demande à la partie contrôle de l'IHM, qui a son tour demande à la partie métier le résultat de calcul.

La figure ci-dessous (*cf.* Figure 5.5), montre l'architecture globale de notre exemple.

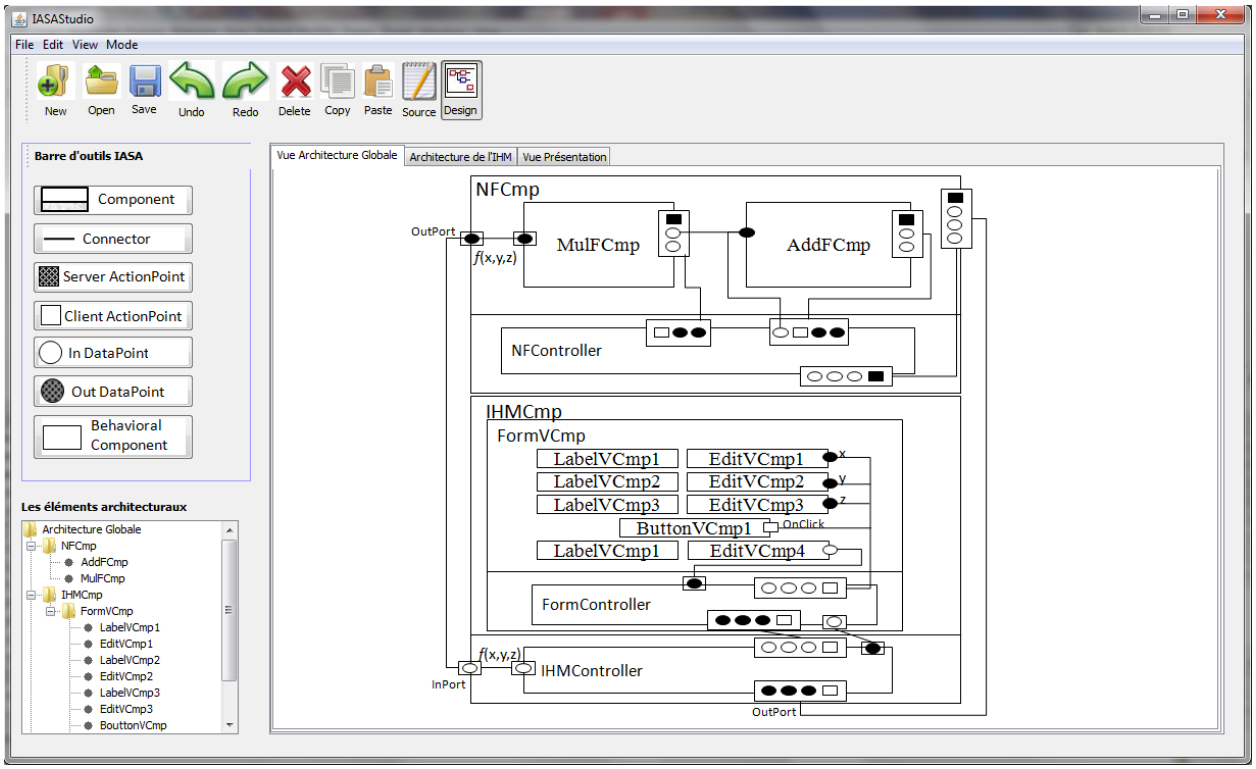


Figure.5.6 : Spécification de l'architecture globale du système pour le calcul de *f*

5.3.4 Spécification de l'IHM finale « *vue présentation* »

La vue présentation est la véritable IHM,elle provient nécessairement de la vue *architecture*, et est fortement dépendante de la technologie sous jacente. Le processus de création d'une vue présentation à partir d'une architecture, est appelée projection. L'IHM finale est

projetée dans un langage de programmation, nous avons choisi *JAVA* comme langage cible, et la boite à outils *swing* pour les composants graphiques.

L'IHM finale doit être fonctionnelle, donc le fait de cliquer sur un bouton, doit lancer le traitement correspondant, elle doit donc encapsuler le code métier.

La projection consiste à parcourir tout les composants d'IHM, et associer chacun à un composant visuel, de père en fils. Le composant *FormVCmp* est associé au composant graphique *JFrame*. Le *LabelVCmp* est associé au *JLabel,* ainsi de suite. Pour l'instant, nous nous sommes contentés des composants graphiques 'widgets', les plus utilisés.

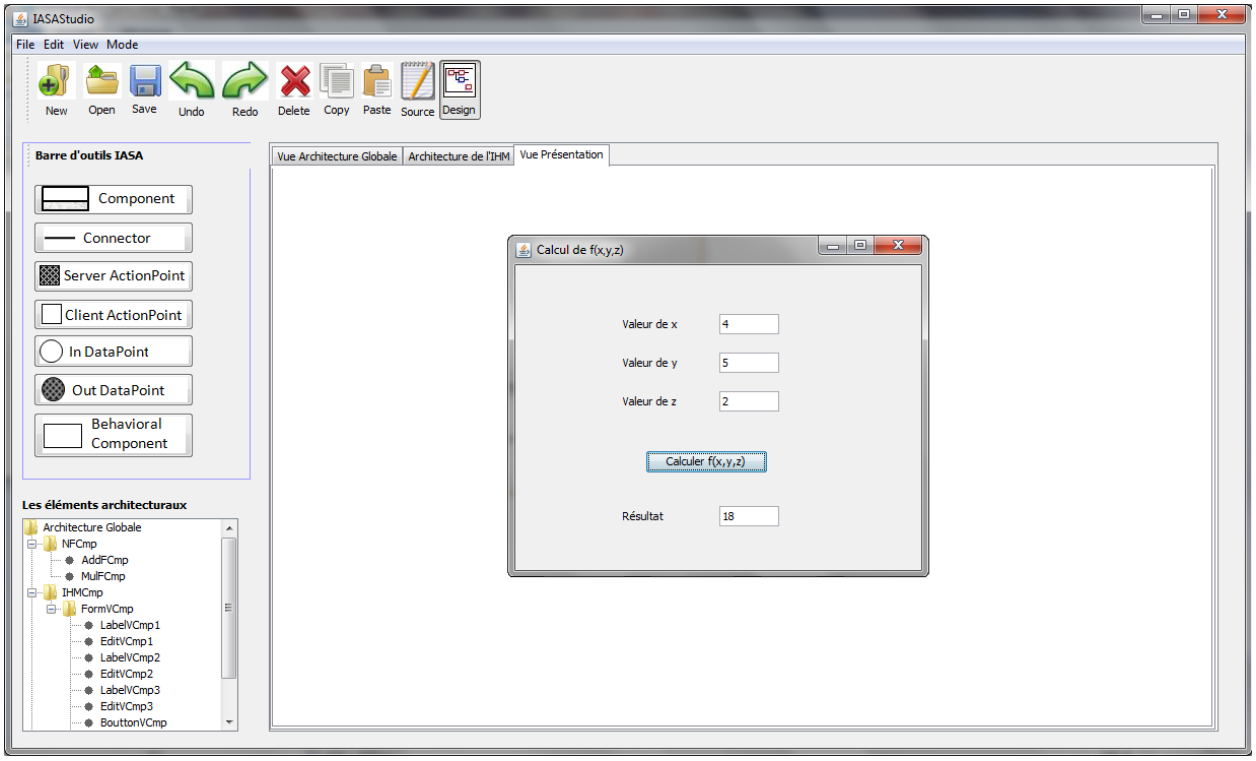


Figure.5.7 : projection de l'architecture globale en IHM finale et fonctionnelle

5.4 Choix techniques d'implémentation

Nous apportons dans ce qui suit une vue sur l'implémentation de l'outil IASAStudio, ainsi qu'a son modèle d'implémentation.

5.4.1 Le langage utilisé

Nous avons utilisé JAVA comme langage de programmation de l'outil. Quand on met en avant la portabilité et la sûreté du langage, Java se révèle unenvironnement de choix qui se révèle très efficace. Celangage a également l'avantage d'être maintenant très répandu, et de disposer de très bons outils de développement. Par ailleurs, le développement en Java suscite l'intérêtd'un nombre croissant de chercheurs.

5.4.2 Le modèle d'implémentation

Le diagramme de classes que nous présentons ici, permet de montrer les liens qui existent, entre les principales classes d'IASAStudio.

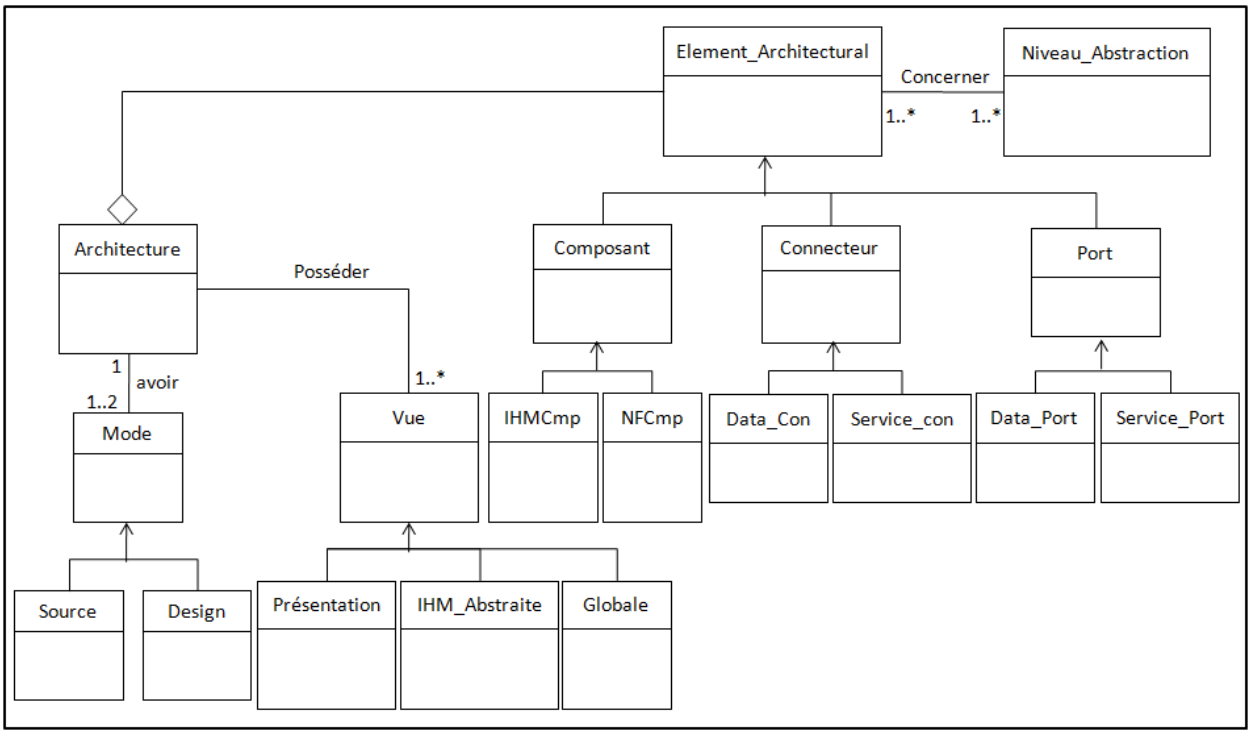
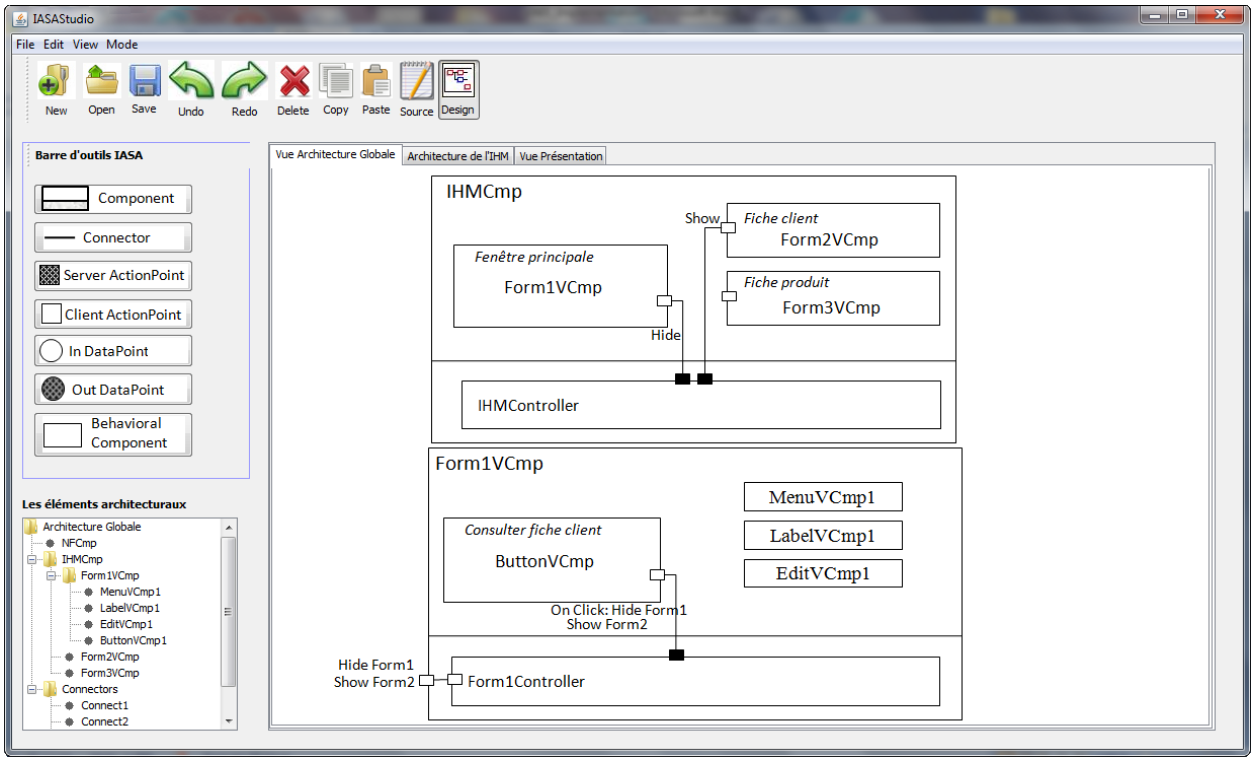


Figure.5.8 : Les principales classes d'IASAStudio

Nous utilisons le concept *Element_Architectural*,pour décrire tout élément qui rentre dans la définition de l'architecture. Nous ajoutons la notion de niveau d'abstraction, qui est très importante dans une architecture. Il est intéressant de conserver cette information, pour une identification plus facile des éléments architecturaux, du niveau inférieur, correspondants à un élément architectural d'un niveau donné. Cette correspondance est nécessaire, pour propager notamment les impacts de l'évolution d'un élément d'un niveau, vers les éléments de son niveau

inférieur. Nous présentons ci-dessous, un exemple d'un même composant visuel, *FormVCmp1*, selon deux niveaux d'abstraction.



Le composant *Form1VCmp,* qui représente une fenêtre est présent ici deux fois, à deux niveaux d'abstraction différents. *IHMCmp* comporte les différentes fenêtres de l'application, qui sont décrites à un niveau d'abstraction élevé, par la suite, le composant *Form1VCmp* est plus détaillé.

Grâce à la notion de niveau d'abstraction présente dans notre implémentation, il est possible de spécifier une architecture, à plusieurs niveaux, comme il est possible de la construire de manière itérative, et incrémentale, ceci est très utile dans le cas des logiciels complexes.

5.5 Evaluation du modèle de composants

Le développement d'IASA Studio permet de concrétiser l'ensemble des propositions autour de l'approche IASA. Il fournit un atelier pour la construction incrémentale de descriptions d'architecture logicielle. Un travail important dans cette mise en œuvre a été effectué, pour

limiter la quantité de code à écrire. IASA Studio bénéficie ainsi de différentes bibliothèques, permettant de générer une partie du code.

Pour la prise en charge des interfaces homme machine, nous avons réorganisé l'outil IASA Studio en trois vues. Cette nouvelle structure permet de prendre en charge, plus tard, d'autres aspects de la définition d'architecture, en augmentant seulement le nombre de vues.

Le modèle de composants que nous avons proposé, vise à considérer une IHM comme assemblage de composants d'IHM, et ensuite, à lier ces derniers avec les composants métier. Pour l'évaluation de l'efficacité de notre modèle, nous nous sommes contentés pour l'instant, de l'exemple présenté dans ce chapitre. L'IHM finale obtenue est fonctionnelle, elle est muni de la vue *architecture logicielle,* qui est très utile pour la conception, l'élaboration, la maintenance, et l'évolution d'une IHM. Le modèle de composants présente de multiples avantages :

- Avoir une spécification d'IHM indépendante du support, ceci donne la possibilité de choisir plusieurs langages cibles, pour le développement de l'IHM ;

- Fournir une représentation globale du système, claire, permettant de bien analyser les liens entre différents composants d'IHMs du système, ainsi que les services déclenchés par l'IHM, ceci offre la possibilité, de détecter des anomalies de manière précoce.

- Donner la possibilité au concepteur de mettre l'accent sur les parties de l'IHM, qui présente plus de possibilités d'évoluer avec le temps.

- Faciliter la maintenance de l'IHM.

5.6 Conclusion

Ce chapitre a présenté notre contribution dans l'outil IASAStudio. Nous avons réorganisé cet outil en trois vues, et ajouter un module important, qui consiste en la prise en charge de la spécification des interfaces homme machine. Notre contribution vise à simplifier l'analyse, la conception, la réalisation, ainsi que la maintenance des IHMs. L'idée de base est de considérer

une IHM comme un composant composite, contenant des composants visuels, et interagissant avec la partie métier.

En démarrant du modèle de tâche utilisateur, et en passant par une spécification par composants, nous avons pu arriver à une IHM fonctionnelle, s'exécutant sur un poste de travail, IASAStudio ne supporte actuellement que JAVA comme langage cible, mais il peut être étendu par la suite pour supporter de nouveaux langages de programmation.

Il reste, au niveau des perspectives de développement, un travail important à mener sur l'extension d'IASAStudio, nous voulons que cet outil soit considérer comme gestionnairede conception des IHMs, qui avertit l'architecte des erreurs ou des maladresses de construction d'une architecture logicielle, et sur le plan ergonomique aussi que nous n'avons pas pris en considération dans notre travail.

CONCLUSION

Le travail effectué dans cette thèse s'inscrit dans la discipline de *l'architecture logicielle,*qui est une discipline récente du génie logiciel. L'architecture Logicielle s'intéresse à la conception de systèmes informatiques, par assemblage de composants. La conception par assemblage est rendu possible, grâce au modèles de composants, sur lesquels se basent cette discipline.

Dans le cadre de cette thèse, nous nous sommes intéressés à la problématique de la spécification des interfaces homme machine, selon une approche *architecture logicielle*. Nous avons étudié différentes facettes de cetteproblématique en proposant :

Une étude détaillée sur la terminologie des architectures logicielles,

Dans cette étude, nous avons présenté dans ce chapitre le contexte global dans lequel s'inscrit cette thèse : le domaine des architectures logicielles à base de composants. Nous avons décrit en détail, les éléments de base d'une architecture qui sont les composants, et les connecteurs. Pour mieux comprendre ce que recouvre l'architecture logicielle, nous avons synthétisé les définitions notables proposées dans la littérature. Nous avons également consacré une section,pour la présentation des principales communautés utilisant l'architecture logicielle.

Un état de l'art sur la spécification des interfaces homme machine

L'état de l'art que nous avons effectué dans cette partie, concerne le domaine desIHMs, nous avons cherché toutes les méthodes etoutils actuels, utilisés pour la spécification, ou le développement des interfaces graphiques, et avons conclu que les méthodes actuels, conduisent deux difficultés principales : les IHMs deviennent moins portables, et seront peu réutilisables. Nous avons remarqué qu'

Un état de l'art sur la prise en compte des interface homme machine, dans une spécification *architecture logicielle*

Dans cette partie, avons effectué une étude approfondie, sur la prise en compte des interfaces homme machine, dans les approches à composants les plus répondues. Cette étude nous a permis de constater la grande diversité dans les caractéristiques, et les objectifs de ces approches. En effet, même si la partie structurelle du composant, semble maintenant relativement stabilisée avec l'utilisation de la notion de port, d'interface, d'opération ou de méthode, la sémantique du composant varie encore beaucoup selon les modèles.De ces différences d'objectifs, et de caractéristiques découlent de nombreux critères d'évaluation possibles d'un modèle. Nous avons choisi nos propres critères d'évaluation, qui satisfassent une bonne spécification d'IHM. Ces critères couvrent la structure d'une IHM, son comportement, et ses liens avec la partie métier de l'application.

La problématique des IHMs en architecture logicielle, a été soulevée suite à une observation à travers laquelle, il parait clairement qu'une grande partie des travaux de recherches, en architecture logicielle, s'intéressent à la conception de la partie métier d'une application. Une analyse des modèles actuels nous a permis de déterminer les caractéristiques fondamentales, d'un modèle de composants, qui supporterait efficacement la spécification des IHMs.

Bilan

IASAest une approche orientée aspect d'Architecture logicielle. Elle vise à généraliser l'approche *architecture logicielle*, à la conception de logiciels de tailles diverses. En plus de permettre le raisonnement avec des composants, représentant des sous systèmes d'une application (gros composant), l'approche IASA permet de supporter un raisonnement d'assemblage de composants, ou ces derniers sont très fins, tels qu'un operateur arithmétique ou logique. C'est dans le contexte de l'approche IASA, que nous avons proposé notre modèle de composants. En réalité, les concepts de base de l'approche IASA ont été d'une grande utilité pour nous.

Le modèle que nous avons proposé, respecte l'organisation générale d'un composant IASA, il possède une vue externe, et une vue interne. Un composant IASA peut soit être un composant métier, et donc posséder une partie opérative et une partie contrôle, soit être un composant d'IHM, et posséder en plus des composants comportementaux pour capter les différents évènements, et déclencher des services.

Un composant métier est lié à un composant d'IHM grâce aux connecteurs. La composition au niveau d'un composant d'IHM est hiérarchique, ainsi, un composant d'IHM peut contenir d'autres composants visuels.

La description architecturale de l'interface homme machine se fera à l'aide de 3ADL. Le langage de description d'architecture logicielle, défini dans le contexte de l'approche IASA, et étendu dans notre travail pour prendre en charge les IHMs. Le langage 3ADL nous a permit de décrire l'architecture de l'IHM de façon abstraite, indépendamment de toute technologie et de tout langage de programmation.

Le développement d'IASA Studio, nous a permit de concrétiser l'ensemble des propositions autour de l'approche IASA, en choisissant la boite à outil swing de Java comme cible, nous avons pu générer le code source d'une IHM, avec sa description architecturale.

Si les modèles de conception actuels sont centrés sur l'architecture logicielle, l'utilisation de cette dernière dans la spécification des IHMs, nous a permis d'identifier quelques avantages :

- Avoir une spécification d'IHM indépendante du support, ceci donne la possibilité de choisir plusieurs langages cibles, pour le développement de l'IHM ;
- Fournir une représentation globale du système, claire, permettant de bien analyser les liens entre différents composants d'IHMs du système, ainsi que les services déclenchés par l'IHM, ceci offre la possibilité, de détecter des anomalies de manière précoce.
- Donner la possibilité au concepteur de mettre l'accent sur les parties de l'IHM, qui présente plus de possibilités d'évoluer avec le temps.
- Faciliter la maintenance de l'IHM.

Ce travail a permis de réconcilier le monde de l'IHM, et du métier, enproposant une vision commune, de ces deux niveaux. Mais, n'est pas clos pour autant, des approfondissements restent à faire. Voyons cequ'il en est.

Perspectives

Le travail réalisé peut se prolonger vers plusieurs perspectives de recherche.

Preuve de correction du code généré : Une description d'architecture logicielle d'une IHM, vérifie tout un ensemble de propriétés de cohérence au niveau de l'assemblage de composants. Ces propriétés sont vérifiables au niveau de la spécification, mais aucune garantie n'est proposée au niveau de l'implantation, à l'aide d'une plate-forme à composants. Une de nos

perspectives, consiste à travailler sur les propriétés architecturales au niveau d'IASA Studio, afin de garantir que l'IHM application reste conforme à la description de son architecture. La vérification de ces propriétés peut consister à vérifier que les interactions au niveau de l'IHM, respectent les interactions précisées dans la description d'architecture logicielle.

Implémenter un gestionnaire de conception d'IHM : qui avertit l'architecte des erreurs, ou des maladresses de construction d'une architecturelogicielle. De plus, il pourrait prendre en charge un coté, que nous n'avons pas pris en considération dans notre travail, qu'est l'ergonomie des IHMs. C'est un travail très important, pour assister l'architecte dans son travail de conception, et transformer IASA Studio en un véritable assistantpour l'architecte dans son travail de conception.

REFERENCES

1. Software Engineering INSTITUTE, “How do you define software architecture?” http://www.sei.cmu.educ/architecture/definitions.html.
2. Bass, L., Clements, P., etKazman, R., “Software Architecture in Practice”, Addison-Wesley Professional, *2003*.
3. Perry, D.E, Wolf, A.L., “Foundations for the study of software architecture”, SIGSOFT Software Engineering Notes, V.17, *1992*, 40-52.
4. Chardigny, S., “Extraction d’une architecture logicielle à base de composants depuis un système orienté objet, Une approche par exploration”, Thèse de Doctorat,2009, UFR Sciences & Techniques, Université de Nantes.
5. Sadou, N., “Evolution Structurelle dans les Architectures Logicielles à base de Composants”, Thèse de Doctorat, UFR Sciences & Techniques, 2007, Université de Nantes.
6. Szyperski, C., “Component Software*”,* ISBN: 0-201-17888-5, Addison-Wesley, 1998.
7. Sevilla, D., Garcia, J.M., and Gomez, A., “CORBA Lightweight Components: A Model for Distributed Component-Based Heterogeneous Computation”, Lecture Notes in Computer Science, 2001, V.2150, 2001, 845-854.
8. Microsoft, “COM: Component Object Model Technologies”,http://www.microsoft.com/com/default.mspx
9. Oussalah, M., “Ingénierie des composants : concepts, techniques et outils”,Editions Vuibert, *2005*.
10. Chefrour, D., et André, F., “Aceel : modèle de composants auto-adaptatifs”, In Systèmes à composants adaptables et extensibles, Grenoble, France, 2002.
11. Belloir, N., “Composition conceptuelle basée sur la relation Tout-Partie”, Thèse de Doctorat, Université de Pau et des Pays de l’Adour, 2004.
12. Shaw, M., et Garlan, D., “Software architecture : perspectives on an emerging discipline”, Prentice-Hall, USA, *1996*.

13. Garlan, D., Monroe, R., et Wile, D., “Acme: An architecture description interchange language”, Proceedings of CASCON’97, Toronto, Canada, 1997.
14. Mehta, N.R., Medvidovic, N., et Phadke, S., “Towards a taxonomy of software connectors”, In ICSE ’00 : Proceedings of the 22nd international conference on Software engineering, pages 178–187, New York, NY, USA, *2000*, ACM.
15. Rosenblum, D.S., Wolf, A.L, “A design framework for internet-scale event observation and notification”, *SIGSOFT Software Engineering Notes*, 22, 344–360, *1997*.
16. Medvidovic, N., Taylor, R.N., “A classification and comparison framework for software architecture description languages”, *IEEE Transaction Software Engineering*, 26, 70-93, *2000*.
17. Allen, R., etGarlan, D., “A formal basis for architectural connection”, ACM Transactions on Software Engineering and Methodology, 6(3):213–249, 1997.
18. Luckham, D. C., Kenney, J. L., Augustin, L. M., Vera, J., Bryan, D. et Mann, W., “Specification and analysis of system architecture using rapide”, IEEE Transactions on Software Engineering, 21(4):336–355, (1995).
19. Zelesnik, G., “The UniCon Language Reference Manual”, School of Computer Science Carnegie Mellon Universtity, Pittsburgh , Mai 1996.
20. Garlan, D., “An introduction to the Aesop System”, School of Computer Science, Carnegie Mellon University, Juillet 1995.
21. Garlan, D., et Perry, D.E.,”Introduction to the special issue on software architecture”, IEEE Transactions on Software Engineering, 21(4) :269–274, *1995*.
22. Favre, J.M., Cevantes, H., Duclos, F., Sanlaville, R., etEstublier, J.,”Issues in reengineering the architecture of component-based software”, In Proceedings of the WCRE 2001, Discussion forum on Software Architecture Recovery and Modeling, 2001.
23. Coutaz, J., ”Interfaces Homme-Ordinateur : conception et réalisation”,Paris, France : Dunod informatique, 1990, 455 p. ISBN : 2040196358.
24. Norman D., “A Cognitive engineering”. In User Centered System Design : new perspectives on human-computer interaction, New-Jersey : Lawrence Erlbaum associates, 1986, pp. 31-61.

25. Navarre, D., “Contribution à l'ingénierie en Interaction Homme-Machine : Une technique de description formelle et un environnement pour une modélisation et une exploitation synergiques des tâches et du système”, thèse de doctorat, l’Université Toulouse, 2001.
26. Falzon P., “Ergonomics, knowledge development and the design of enabling environments”, Humanizing Work and Work Environment HWWE’2005 Conference, December 10-12 Guwahati, India, pp.1-8, 2005.
27. Vander Veer G., Lenting B., Bergevoet B.,” GTA: Groupware Task Analysis-Modelling Complexity”, In *Actapsychologica*, 1996, pp. 297-322.
28. Scapin, D. L., Golbreich, C., “Towards a Method for Task Description: MAD”. In*ProceedingsWorkwith Display Unit*, 1989.
29. Samaan K., “Prise en compte du modèle d'interaction dans le processus de construction et d'adaptation d'applications interactives”, thèse de doctorat, Ecole Centrale de Lyon, 2006.
30. Hartson H. R., Siochi A. C., Hix D., “The UAN: A user-oriented representation for direct manipulation interface designs”. ACM Transactions on Information Systems, Vol. 8, N° 3, 1990, pp. 181–203.
31. Paternò, F., “Model Based Design and Evaluation of Interactive Application”, Springer Verlag, 1999.
32. Scapin D. L., Bastien J. M. C., “Analyse des tâches et aide ergonomique àla conception : l'approche MAD”, Hermès, Paris, 2001, ISBN 2-7462-0239-5.
33. Tabary D., “Contribution à TOOD, une méthode à base de modèles pour laspécification et la conception des systèmes interactifs”. Thèse de doctorat, Université deValenciennes et du Hainaut-Cambrésis, Valenciennes, 2001.
34. Balbo, S., Ozkan, N. et Paris, C., “Choosing the Right Task-modeling Notation: A Taxonomy”. The Handbook of Task Analysis for Human-Computer Interaction, D. Diaper and N. A. Stanton, 2004, pp. 445-466
35. Limbourg, Q., Vanderdonckt, J., “Comparing Task Models for User Interface Design (Chapter 6)”, The Handbook of Task Analysis for Human-Computer Interaction, D. Diaper and N. Stanton, 2003, pp.
36. Barthet, M.F.,”Logiciels Interactifs et Ergonomie, Modèles et méthodes de conception”. Dunod Informatique, 1988.

37. Martijn, V.W., “task-based user interface design”, thèse de doctorat, Dutch Graduate School for Information and Knowledge Systems, 2001.
38. Card, S.K., Moran, T. P., Newell, A., “The psychology of human-computer interaction”. Lawrence Erlbaum Associates; 1983.
39. Arnaud, B., “Vers une démarche industrielle pour le développement d’Interfaces Homme-Machine : De l’analyse de l’activité à la génération du code”, thèse de doctorat, Université de Rouen, 1999.
40. Card S.K., Moran T.P., Newell A., “The Keystroke-Level Model for User Performance Time with Interactive Systems”. Communications of the ACM, Vol 23 No7,1980, pp. 396-410.
41. Kieras D. E., “Guide to GOMS model usability evaluation using NGOMSL”, Handbook of Human-Computer Interaction. M. H. a. T. Landauer. Amsterdam, North-Holland, 1996.
42. Gray W. D., John B. E., Atwood M. E., “Project Ernestine: Validating a GOMS Analysis for Predicting and Explaining Real-World Task Performance”, Human-Computer Interaction, Vol. 8, N° 3, 1993, pp.237-309.
43. Gray P., England D., McGowan S., “XUAN: enhancing the UAN to capture temporal relationships among actions”, Research Report, IS-94-06, GIST, University of Glasgow, UK. 1994
44. Dix, A. J., “Formal Methods for Interactive Systems”, Academic Press, 1991.
45. Paternò, F., Mancini, C., Meniconi, S., “ConcurTaskTrees: A Diagrammatic Notation for Specifying Task Models”, (1997), Proceedings of the IFIP TC13 Interantional Conference on HumanComputer Interaction.
46. Dragicevic, P., “Un modèle d'interaction en entrée pour des systèmes interactifs multi-dispositifs hautement configurables“, thèse de doctorat de l'université de Nantes, l'École Nationale Supérieure des Techniques Industrielles et des Mines de Nantes, 2004.
47. Pfaff, G., Hagen. P., “Seeheim Workshop on User Interface Management Systems”, Springer-Verlag, Berlin, 1985.
48. The UIMS tool developers workshop, “A metamodel for the runtime architecture of an interactive system”, ACM SIGCHI Bulletin, 24(1):32-37, 1992.
49. Elmarzouqi, N., Garcia, E., Lapayre, J.C., “Continuum de Collaboration Augmentée dans un nouveau modèle de TCAO“. In SETIT'07, 4th IEEE int. conf. on Sciences of

Electronic, Technologies of Information and Telecommunications**,**Hammamet, Tunisia, pages 94--106, March 2007.

50. Schmucker, K., ”MacApp: an application framework”, Byte Magazine, 11(8):189-193, (1986).

51. Krasner, G. E., Pope, S. T., “A cookbook for using the model-view-controller user interface paradigm in Smalltalk-80”. Journal of Object OrientedProgramming, 1(3):26-49, (1988).

52. Goldberg, A., “Smalltalk-80: The Interactive Programming Environment”, Addison-Wesley, 1984.

53. Model-view-controller (MVC) Architecture, http://www.jdl.co.uk/briefings/index.html#mvc

54. Jacquet,C., “Présentation opportuniste et multimodale d’informations dans le cadre de l’intelligence ambiante“ , l’Université de Paris-Sud XI Orsay, 2006.

55. Coutaz, J., “Interface Homme-Ordinateur : Conception et Réalisation“ ; Thèse de doctorat d'état, Université Joseph Fourier de Grenoble, 1988.

56. Depaulis, F., “Vers un environnement générique d’aide au développement d’applications interactives de simulations de métamorphoses“, thèse de doctorat, 2002, ENSMA, université de Poitier.

57. Nigay, L.,“Conception et modélisation logicielles des systèmes interactifs“, thèse de l’Université Joseph Fourier, Grenoble, 1994.

58. Graphic Java 2 AWT, Volume 1,Editeur : Prentice Hall; Edition **:** 3 (21 septembre 1998), ISBN-10: 0130796662.

59. Eckstein, R., loy, M.,“Java Swing. O’Reilly& Associates », Inc., 981 Chestnut Street, Newton, MA 02164, USA, 2e édition, 2002. ISBN 0-596-00408-7.

60. http://www.htmlkit.com

61. OMG, “Xmlmetadatainterchange (xmi) v2.0 specification“, document formal/03-05-02. Object Management Group Web Site : http ://www.omg.org/docs/formal/03-05-02.pdf, 2003.

62. Dubinko, M., Leigh, L., Klotz, Jr., Merrick, R., Raman, T. V., “XForms1.0., W3C Recommendation, World Wide Web Consortium, October 2003. (http://www.w3.org/TR/xforms/)

63. Pemberton, S. et al., XHTML 1.0: The Extensible HyperText Markup Language: A Reformulation of HTML 4 in XML 1.0. W3C Recommendation, World Wide Web Consortium, Janvier 2000. (http://www.w3.org/TR/xhtml1/)

64. Abrams, M., Helms, J., "User Interface Markup Language (UIML) Specification", version 3.1. Technical report, Harmonia, 2004.

65. Merrick, R. A., Wood, B., Krebs, W., "Abstract User Interface Markup Language", In Advances on User Interface Description Languages, Workshop of Advanced Visual Interfaces (AVI) 2004, Expertise Center for Digital Media, 2004.

66. Coutaz, J., Nigay, L., "architecture logicielle conceptuelle des systèmes interactifs", in KOLSKI 2001 IHM pour les SI, vol1, p. 207-246.

67. Depaulis, F., Jambon, F., Girard, P., Guittet, L.,"Le modèle d'architecture logicielle H4 : Principes, usages, outils et retours d'expérience dans les applications de conception technique", Revue d'Interaction Homme-Machine, vol. 7, pp. 93-129, 2006.

68. Vestal, S., "A cursory overview and comparison of four architecture description languages". Technical report, Honeywell Technology Center. (1993).

69. Le Goaer, O., "Styles d'évolution dans les architectures logicielles",Thèse de doctorat,Laboratoire d'Informatique de Nantes Atlantique (LINA), université de Nantes, 2009.

70. Magee, J., Dulay, N., Eisenbach, S., Kramer, J., "Specifying distributed software architectures". Dans Proceeding of the 5th European Software Engineering Conference, ESEC '95, (1995).

71. Magee, J., Kramer, J., "Dynamic structure in software architectures", Dans Proceedings of the Fourth ACM SIGSOFT Symposium on the Foundations of Software Engineering, pages 3–14, (1996).

72. MOO MENA, F.J., "modélisation des architectures logicielles dynamiques : application à la gestion de la qualité de service des applications à base de services Web", Thèse de doctorat, Laboratoire d'Analyse et d'Architecture des Systèmes, l'Institut National Polytechnique de Toulouse, 2007.

73. Hall, A., Chapman, R., "Correctness by construction: Developing a commercial secure system", dans IEEE Software, vol. 19, n° 1, 2002, p. 18–25.

74. Raz, Y., "The Dynamic Two Phase Commitment (D2PC) protocol ",Database Theory ICDT '95, Lecture Notes in Computer Science, Volume 893/1995, pp. 162-176, Springer, ISBN 978-3-540-58907-5, (1995).

75. Allen, R., Douence,R., Garlan, D., “Specifying Dynamism in Software Architectures“, Proceedings of the First Workshop on the Foundations of Component-Based Systems, Zurich, Switzerland, 1997.

76. Medvidovic, N., Taylor, R., Whitehead, E., “Formal Modeling of Software Architectures at Multiple Levels of Abstraction”, Department of Information and Computer Science, University of California, Irvine, Avril 1996.

77. Medvidovic, N., Taylor, R., Khare, R., Guntersdorfer, M., “An Architecture-Centered Approach to Software Environnement Integration”, Department of Information and Computer Science, University of California, Irvine, Mars 2000.

78. Projet ACCORD : Etat de l’art sur les Langages de Description d’Architecture(ADLs). Livrable 1.1-2, Juin 2002.

79. Fielding, R.T., “Architectural styles and the design of network-based software architectures”, Thèse de doctorat, 2000.

80. Bennouar, D., “une approche intégrée pour l’architecture logicielle“, thèse de doctorat, école supérieur d’informatique (ESI).

81. Bennouar, D., Khammaci, T., Henni, A.,”A new approach for component’s port modeling in software architecture”, The Journal of Systems and Software 83, (2010) 1430–1442.

82. Bennouar, D., Saadi, A., ”SEAL, Un langage pour la spécification de composants exécutables à un haut niveau d’abstraction”, Rapport Interne, SA0306, Laboratoire LDRSI, USDB, Blida, Nov2006.

83. Aldrich, J., Chambers, C., Notkin, D., “ArchJava : connecting software architecture to implementation”, In Proceedings of the 24th International Conference on Software Engineering (ICSE-02), pages 187–197, New York, mais 19–25, 2002. ACM Press.

84. Kalibera, T., Tuma, P., “Distributed component system based on architecture description: The sofa experience“, In On the Move to Meaningful Internet Systems, 2002 - DOA/CoopIS/ODBASE 2002 Confederated International Conferences DOA, CoopIS and ODBASE 2002, pages 981–994, London, UK, octobre 2002. Springer-Verlag. ISBN : 3-540-00106-9.

85. Romeo, F., “Administration de composants logiciels pour systèmes sans fil“, thèse de doctorat, l’université de Pau et des pays de l’Adour, Novembre 2007.

86. Bachmann, F., Bass, L., Buhman, C., Comella-Dorda, S., Long, F., Seacord, R.J., Wallnau, K., “Volume II : Technical Concepts of Component-Based Software Engineering”. Technical Report CMU/SEI-2000-TR-008, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, 2000. 6, 11, 19.

87. Box, D., “Essential COM”, Addison-Wesley, Janvier 1998.

88. Sun Microsystems, JavaBeans, http://java.sun.com/javase/technologies/desktop/javabeans/

89. Sun Microsystems, Inc. JavaBeans Specification, v1.01, 1997.

90. Sun Microsystems. Enterprise javabeansspecification version 2.0. Documenten ligne, Aout 2001. http ://java.sun.com/products/ejb/docs.html.

91. Roman, E., Ambler, S., Jewell, T.,“Mastering Enterprise JavaBeans”, Wiley Computer Publishing, second edition, 2002.

92. Cauvet, C.,Ramadour, P., “Ingénierie des composants : concepts, techniques et outils, chapitre : Les composants métiers dans l’ingénierie des systèmes d’information“, pages 197–228. Vuibert, Paris, 2005.

93. AUTOSAR Development Partnership, AUTOSAR - Technical Overview, 2008

94. Hariri, A., Lepreux, S., Tabary, D., Kolski, C., “Principes et étude de cas d’adaptation d’IHM dans les SI en fonction du contexte d’interaction de l’utilisateur“,Ingénierie des Systèmes d’Information (ISI), Networking and Information Systems, 14 (3), pp. 141-162, 2009.

95. Bandelloni, R., et Paternò, F., “Migratory user interfaces able to adapt to various interaction platforms“, Int. J. Human-Computer Studies, Vol. 60, n° 5-6, 2004, p. 621-639.