République Algérienne Démocratique et Populaire
Ministère de l'Enseignement Supérieur et de la Recherche Scientifique

Université Saad Dahleb, Blida
USDB.

Faculté des sciences.
Département informatique.

Mémoire pour l'obtention
D'un diplôme d'ingénieur d'état en informatique.
Option : Intelligence artificielle

Sujet :

Réalisation d'un IDE sous Eclipse pour l'architecture logicielle

Présenté par : Salma Allal
Errahmani Bilel

Promoteur : Djamel Bennouar
Encadreur : Nom encadreur

Organisme d'accueil : USDB.

Soutenue le: date soutenance, devant le jury composé de :

Mme Ouahrani	**Président**
Mr Hadj Yahia	**Examinateur**
Mr Hammouda	**Examinateur**

Année promotion 2006-2007

Remerciement

Remerciements

Avant tous, nous remercions **ALLAH** *que grâce à* **LUI** *que nous avons obtenu et réalisé ce travail*

*Nous tenons à remercier notre promoteur M*r*. DJAMEL BENNOUAR pour nous offrir l'occasion d'avoir ce sujet et pour son aide, sa patience, sa disponibilité, et sa compréhensibilité.*

Nous remecions aussi nos familles respective pour leurs soutien et leurs patience en cours de réalisation de ce travail

Nous remercions les membres du jury pour nous avoir fait l'honneur de juger notre travail.

Nous remercions tous les enseignants du département informatique.

Nous remercions, de tout coeur, tous ceux qui ont contribué de prés ou de loin à la réalisation de ce travail.

Dédicace

A mes très chers parents

A mes frères HANNACHI, MOHAMED, KAMEL, ABD ENNOUR et MAHFOUD

A mes sœurs SALIMA et FATIMA EZZAHRAA

A ma nièce AYA

A mon neveu SID ALI

A Mon binôme ALLAL

A HADJI MOHAMED, YOUSSEF BOUMSID et KEMMANE ABDELHAK

A KHIDA FAROUK, ABDALLAH MUSTAPHA, BOUKHARI WALID et MESSOUDI HAMZA

A mes amis MOUNIR, SIFO, MAHDI, SAMIR, ELAARBI et KADI

A HASAIRY MOHAMED et RAHMANI AMINE

Errahmani Bilel

Dédicace

A mes très chers parents

A mes frères RABAH, KAMEL, MOURAD, SLIMENE et AMINE

A mes sœurs :FATIHA ,HOURIA et FATMA

A ma nièce AMIRA

A mes neveu ABDO, OUSSAA, AIMANE, IMAD EDDINE

A Mon binôme BILEL

A "FAROUK, MUSTAPHA, WALID, HAMZA, REDUANE, MOHAMED et KALIM"

A mes amis MOUNIR, MAHDI, SAMIR, SIFO, ELAARBI et KADI et SAFIA ,SAMIA, HOURIA, HASSIBA ,FOUZIA.

Salma Allal

Réalisation d'un environnement de développement intégré pour l'architecture logicielle sous forme d'un plugin Eclipse

Le grand succès du modèle objet et sa prise en charge efficace par un nombre important de langage de programmation a grandement ouvert la voie vers la mise en place de modèles et d'outils qui devrait permettre d'appliquer à la conception de logiciels des démarches similaires à celles des concepteurs de système électronique. Ces modèles se dite les modèles composant.

Notre travail consiste à réaliser un environnement de développement intégré pour un modèle dite l'approche intégré vers l'architecture des logiciels.

Cet environnement devrait permettre de dessiner des architectures basées sur ce modèle, et généré un code spécifique pour l'architecture dessiné.

Realization of an integrated development environment for software architecture as an Eclipse plug-in

The big success of object model and its effective using by an important number of programming languages has opened the way to create models and tools that allow the application of steps similar to that used in electronics systems concept to software concept. Those models called component models.

Our work consists in realization of an integrated development environment for a model called integrated approach to software architecture.

This environment should allow drawing architecture draw on this model, and generate a specific code for the drown architecture.

إنجاز وسط مدمج للتطوير لأجل هندسة البرامج على شكل وصلة ل"إكليبس"

إن النجاح الكبير للنموذج الشيئي و استعماله من طرف عدد كبير من لغات البرمجة، فتح الطريق لتطوير نماذج و أدوات التي من المفترض أن تسمح بتطبيق خطوات مشابهة لتلك المستعملة من طرف مصممي الأنظمة الإلكترونية على تصميم البرامج. هذه النماذج تدعى بنماذج المركبات.

عملنا ينص على إنجاز وسط مدمج للتطوير لأجل نموذج يدعى النمط المدمج نحو هندسة البرامج.

هذا الوسط يجب أن يسمح بتخطيط هندسيات مسندة على هذا النموذج، و استخراج اصطلاح مخصص للمخطط.

Introduction Générale

INTRODUCTION GENERALE

L'architecture logicielle décrit d'une manière symbolique et schématique les différents composants d'un ou de plusieurs programmes informatiques, leurs interrelations et leurs interactions. Contrairement aux spécifications systèmes résultantes de l'analyse fonctionnelle, l'architecture logicielle, produite lors de la phase de conception, ne décrit pas ce que doit réaliser un programme mais plutôt comment il doit être conçu de manière à répondre aux spécifications. L'analyse d'écrit le « quoi faire » alors que l'architecture décrit le « comment le faire ».

On peut distinguer deux modèles d'architecture :

- Le modèle objet.
- Le modèle composant.

Le modèle orientée objet considère le logiciel comme une collection d'objets dissociés et identifiés, définis par des propriétés. Une propriété est soit un attribut (i.e. une donnée caractérisant l'état de l'objet), soit une entité élémentaire comportementale de l'objet. La fonctionnalité du logiciel émerge alors de l'interaction entre les différents objets qu'ils constituent. L'une des particularités de cette approche est qu'elle rapproche les données et leurs traitements associés au sein d'un unique objet[cours-UML] . Comme nous venons de le dire, un objet est caractérisé par plusieurs notions:

✓ L'identité : L'objet possède une identité, qui permet de le distinguer des autres objets, indépendamment de son état. On construit généralement cette identité grâce à un identifiant découlant naturellement du problème (par exemple un produit pourra être repéré par un code, une voiture par un numéro de série, etc.)

✓ Les attributs : Il s'agit des données caractérisant l'objet. Ce sont des variables stockant des informations sur l'état de l'objet.

✓ Les méthodes : Les méthodes d'un objet caractérisent son comportement, c'est-à-dire l'ensemble des actions (appelées opérations) que l'objet est à même de réaliser. Ces opérations permettent de faire réagir l'objet aux sollicitations extérieures(ou d'agir sur les autres objets).De plus, les opérations sont

étroitement liées aux attributs, car leurs actions peuvent dépendre des valeurs des attributs, ou bien les modifier.

La difficulté de cette modélisation consiste à créer une représentation abstraite, sous forme d'objets, d'entités ayant une existence matérielle (chien, voiture, ampoule, personne,...) ou bien virtuelle (client, temps,...).

La Conception Orientée Objet (COO) est la méthode qui conduit à des architectures logicielles fondées sur les objets du système, plutôt que sur la fonction qu'il est censé réaliser.

Le modèle composant considère le logiciel comme une collection de composants interconnectés. Le modèle composant est caractérisé par plusieurs notions :

✓ Le composant : Un composant est avant tout une unité de composition qui sera utilisé dans une conception par simple assemblage de composant. C'est une entité développé indépendamment du logiciel dans lequel il sera utilisé. C'est une entité responsable de la réalisation d'une (ou plusieurs) fonctionnalité(s) bien précise(s) dans une architecture à un certain niveau d'abstraction ou implémentation. C'est une entité fournissant des fonctionnalités de calcul et de stockage. Il interagit avec les autres composants pour réaliser un ou plusieurs objectifs d'une architecture. Un composant peut être très simple, comme un objet d'une classe C++ ou complexe comme un serveur HTTP ou un SGBD. Les composants complexes peuvent être le résultat d'assemblage de deux ou plusieurs composants simples(ou complexes).

✓ Le port : La caractéristique fondamentale d'un composant est la notion de port. Un composant interagit avec le monde extérieur par ses ports. C'est à travers les ports qu'un composant fournis ses services, exprime ses besoins et fixe les conditions à remplir pour qu'il puisse être exploité correctement.

✓ L'assemblage : La technique de conception par assemblage consiste à connecter plusieurs composants entre eux pour obtenir un système opérationnel, Où chaque composant garantie une fonctionnalité de ce système.

✓ Le connecteur : Un connecteur est une liaison entre deux(ou plusieurs) composants. Il est le responsable des différentes interactions entre composants.

La Conception selon le modèle composant est la méthode qui conduit à des architectures logicielles fondées sur les fonctionnalités qu'un système doit réaliser, plutôt que sur les objets qu'il contient.

Le grand succès du modèle objet et sa prise en charge efficace par un nombre important de langage de programmation (C++, JAVA) a grandement ouvert la voie vers la mise en place des IDE équipé d'un éditeur d'interface graphique (comme C++ BUILDER, DELPHI...).

L'IDE (Integrated Development Environment) est une interface qui permet de développer, compiler et exécuter un programme dans un langage donné. La plupart des langages de programmation (Java, Langage C, etc...) sont associés à un outil permettant de saisir du code, compiler, débugger et exécuter des programmes.

Cet outil combine les fonctionnalités d'un éditeur de texte, d'un compilateur et d'un débugger. Il rend plus pratique la programmation, et aussi il peut disposer de fonctions de complétion automatique de textes pour accélérer la saisie. Comme pour le modèle Objet, Il existe des IDE pour le modèle composant (comme Visual Studio de Microsoft qui utilise l'approche .Net).

Notre travail consiste à réaliser un IDE pour l'architecture logicielle selon le modèle composant (nous utilisons l'approche IASA (INTEGRATED APPROCHE TO SOFTWARE ARCHITECURE) voir chapitre II). L'IDE est réalisé dans le contexte de la plateforme ECLIPSE sous forme d'un PLUGIN.

Eclipse IDE est un environnement de développement intégré (le terme *Eclipse* désigne également le projet correspondant, lancé par IBM) extensible, universel et polyvalent, permettant potentiellement de créer des projets de développement mettant en œuvre n'importe quel langage de programmation. Eclipse IDE est principalement écrit en Java (à l'aide de la bibliothèque graphique SWT, d'IBM), et ce langage, grâce à des bibliothèques spécifiques, est également utilisé pour écrire des extensions (PLUGIN).

En informatique, **un plugin** est un programme qui interagit avec un logiciel principal, appelé programme hôte, pour lui apporter de nouvelles fonctionnalités. Le terme plugin provient de la métaphore de la prise électrique standardisée et désigne une extension prévue des fonctionnalités, en comparaison des ajouts non prévus

initialement apportés à l'aide de patches. La plupart du temps, ces programmes sont caractérisés de la façon suivante :

- Ils ne peuvent fonctionner seuls car ils sont uniquement destinés à apporter une fonctionnalité à un ou plusieurs logiciels
- Ils sont mis au point par des personnes n'ayant pas nécessairement de relation avec les auteurs du logiciel principal.

PROBLEMATIQUE

Il est question dans ce sujet de réaliser dans le contexte de la plateforme ECLIPSE, un IDE destiné à la conception de logiciel selon une approche basée sur les concepts de composant et de connecteur. Le modèle de composant et connecteurs utilisé est plus fin que celui d'UML2.0. Le modèle UML2.0 ne permet pas de considérer les éléments d'interface de manière indépendante. Une interface UML2.0 est considérée dans sa totalité dans toute opération de connexion de ports de composant. Le modèle UML2.0 ne permet pas une grande flexibilité dans la spécification d'architecture logicielle et ne peut pas prendre en charges la spécification informelle d'architecture logicielle.

Le modèle sur lequel se basera l'IDE à développer reconnaît les notions de ports et interfaces d'UML2.0 et en plus permet de manipuler les éléments constitutifs de chaque interface que nous appelons : point de connexion. Ces derniers sont de deux grandes catégories : les points véhiculant des actions et les points véhiculant des données.

Les connecteurs peuvent aussi être considérés de manière plus fine que dans UML. C'est ainsi que dans le modèle sur lequel se basera l'IDE il y' a la notion de connecteurs élémentaire et la notion de connecteurs. De plus, les connexions se feront soit par le tracé direct de connecteurs entre ports soit en utilisant des composant de communication.

L'IDE à réaliser permettra de dessiner une architecture logicielle (spécification de composant et de connecteurs entre ports de composant). L'IDE devra permettre un tracé libre et très flexible de connexion, très similaire à la spécification informelle

d'architecture logicielle, donc très proche du modèle mental que se fait un architecte de son logiciel..

L'IDE doit permettre l'enrichissement de la spécification structurelle (composant et connecteur) par la spécification d'un timing associée à l'architecture dessinée et d'un plan de déploiement des composants. Ce sont ces deux aspects qui donneront une rigueur à la spécification informelle. C'est à partir du timing et du plan de déploiement qu'il deviendra par la suite possible de construire à partir d'une spécification informelle une spécification rigoureuse appelée spécification standardisée. La spécification standardisée représente le point de départ pour la génération de code associée à l'architecture indiquée

Après la spécification de l'architecture dans des styles sans grandes contraintes, l'IDE devra en finalité génère un code qui représentera l'architecture dessinée.

METHODOLOGIE DE CONCEPTION ET DE REALISATION

Constat local

- L'objectif est un logiciel local.
- Destiné à être réalisé par un(e) ou deux étudiants sans aucune expérience.
- Non destiné obligatoirement à être terminé par un autre groupe.
- Sans contrainte budgétaire.
- Avec contrainte de la nécessité de mise en place dans les plus brefs délais d'un prototype fonctionnel.
- Du niveau des étudiants en termes de maîtrise des outils de programmation.
- De la nécessité d'une période d'apprentissage des outils.
- D'une période nécessaire à la compréhension du problème.
- D'une période nécessaire à la détermination progressive des vrais besoins.

Les méthodes traditionnelles

Les démarches traditionnelles, basées sur la fameuse séquence « spécification > conception >réalisation > validation », concentrent la plupart des décisions en début de projet.

L'objectif de cette approche est louable : le client veut des garanties sur ce qu'il obtiendra en fin de projet, et le chef de projet souhaite disposer des informations nécessaires à l'organisation de son équipe.

Malheureusement, les équipes qui évoluent dans un environnement changeant ou complexe savent à quel point il est difficile de s'en tenir aux décisions initiales. Le client réalise que ses besoins ont changé, ou bien l'équipe découvre en phase d'implémentation des erreurs de spécification ou de conception qui compromettent les plans de développement. Le changement s'impose donc tôt ou tard, mais voilà : cette organisation suppose l'absence de changement, et celui-ci se révèle bien vite très coûteux - suffisamment parfois pour compromettre la rentabilité du projet.

Mais puisque le changement est une composante incontournable de tout projet de développement logiciel, pourquoi ne pas l'accepter ? N'existe-t-il pas un moyen pour que les équipes de développement n'opposent plus de rigidité excessive aux demandes de leur maîtrise d'ouvrage ?

La méthode proposée

La méthodologie à suivre est basée sur la programmation intensive, itérative et progressive (incrémentale) centré sur les tests.

La méthodologie incite à commencer par les aspects les plus simples, les réaliser et les tester (Exemple : portage sur linux qui en réalité est une tache un peu cruciale) et passer ensuite à la réalisation d'un autre aspect. Ce dernier aspect pourra mettre en cause les aspects précédents. Dans ce contexte un réajustement des étapes précédentes est nécessaire. Il faut revenir en arrière pour refaire la conception /.Réalisation.

Ce processus pratique permettra d'une part de maîtriser l'outil de programmation et d'avoir à chaque étape une version fonctionnelle d'une partie du logicielle.

Cette méthodologie faite partie d'une famille émergente de processus dit processus dits "agiles", qui se démarquent des démarches traditionnelles en mettant l'accent sur le travail d'équipe et la réactivité. La méthode XP fait partie de cette

famille et la méthodologie que nous suivons s'apparente sur beaucoup d'aspect à l'XtremeProgramming. Notre méthodologie se focalise sur la construction proprement dite du logiciel, en aval des phases préparatoires d'études d'opportunité ou de faisabilité.

Dans notre méthode

- Le client (maîtrise d'ouvrage) pilote lui-même le projet, et ce de très près grâce à des cycles itératifs extrêmement courts (1 ou 2 semaines). Le client dans notre cas est le promoteur principal du sujet, à savoir Mr Djamel BENNOUAR, CC au Département Informatique de l'USDB.

- L'équipe formé de deux étudiant en phase de préparation de leur mémoire de fin d'étude, livre, très tôt dans le projet une première version du logiciel, et les livraisons de nouvelles versions s'enchaînent ensuite à un rythme soutenu pour obtenir un feedback maximal sur l'avancement des développements.

- L'équipe constituée des deux étudiantes, s'organise elle-même pour atteindre ses objectifs, en favorisant une collaboration maximale entre ses membres.

- L'équipe doit mettre en place un ensemble de jeux d'essai ou doit mettre en place des tests automatiques pour toutes les fonctionnalités qu'elle développe, ce qui devrait garantit au produit un niveau de robustesse très élevé.

- Les développeurs améliorent sans cesse la structure interne du logiciel pour que les évolutions y restent faciles et rapides.

Le grand gagnant de cette démarche est d'abord le client du projet. Plutôt que de voir son intervention cantonnée à la phase initiale de recueil du besoin, il intègre véritablement le projet pour en devenir le pilote. A chaque itération, il choisit lui-même les fonctionnalités à implémenter, collabore avec l'équipe pour définir ses besoins dans le détail, et reçoit une nouvelle version du logiciel qui intègre les évolutions en question.

Cette démarche présente de nombreux avantages en termes de conduite de projet :

- Le client jouit d'une très grande visibilité sur l'avancement de développement.

- Le client utilise le logiciel lui-même comme support de réflexion pour le choix des fonctionnalités à implémenter, il peut en particulier intégrer très tôt les retours des utilisateurs pour orienter les développements en conséquence.

- La première mise en production du logiciel intervient très tôt dans le projet, ce qui avance d'autant le moment à partir duquel le client peut en tirer des bénéfices.

- L'ordre d'implémentation des fonctionnalités n'est pas guidé par des contraintes techniques, mais par les demandes du client. Celui-ci peut donc focaliser les efforts de l'équipe sur les fonctionnalités les plus importantes dès le début du projet, et ainsi optimiser l'utilisation d'un éventuel budget.

Méthode

1. Analyse globale : Détermination de l'ensemble ECU des besoins (documenter par les use case de UML) le nombre des besoins
2. Pour chaque besoin cu non traité (cas d'utilisation) d'ECU
   - 2.1. Conception de cu -→ ccu
   - 2.2. Réalisation de cu → rcu
   - 2.3. Test rcu et comparaison avec cu
   - 2.4. Si test non satisfaisant
     - 2.4.1. réajuster le besoin cu
     - 2.4.2. aller à 2.1
   - 2.5. Sinon marquer cu comme achevé
   - 2.6. Etudier impact de cu sur ECU. Pour chaque besoin cuii impacté
     - 2.6.1. Réajuster cui

Marque cui comme non achevé (à réétudier) Ajouter d'éventuels nouveaux cun à ECU (émane d'une réorganisation ou d'un ajout simple).

CHAPITRE I

LE MODELE COMPOSANT SELON L'APPROCHE IASA

CHAPITRE I: LE MODELE COMPOSANT SELON L'APPROCHE IASA (INTEGRATED APPROCHE TO SOFTWARE ARCHITECTURE)

I.1 Introduction

Dans ce chapitre, nous présentons en détail les éléments fondamentaux du modèle de composants IASA. Nous commencerons par les éléments modélisant la vue externe (les composants), à savoir le concept de point d'accès et les concepts de ports et de connecteurs. Nous présenterons ensuite le concept d'enveloppe et nous étudierons le modèle de vue interne.

I.2 Le Point D'accès

I.2.1 Présentation générale

Le point d'accès est le plus petit élément manipulable dans l'architecture. Il représente l'élément de base pour la définition d'un port. Toute information, quelque soit sa nature arrive ou part d'un composant à travers un point d'accès. Il permet de localiser les diverses ressources fournies ou requises à travers un port. Il permet de renseigner sur les éléments intervenant dans la réalisation d'un comportement observable sur un port.

Un point d'accès peut être manipulé indépendamment des autres points d'accès. Il est ainsi possible de tirer une connexion d'un point d'accès d'un port de composant vers un autre point d'accès d'un autre port, sans que les autres points d'accès des deux ports n'interviennent dans cette connexion (Figure 1). De cette manière il deviendrait possible d'utiliser un paramètre d'une méthode indépendamment de la méthode dans laquelle il est défini.

Un point d'accès possède un nom qui l'identifie de manière unique dans une architecture. Nous utilisons une notation similaire au nom de domaine Internet, dans laquelle le nom du point d'accès est une feuille dans un arbre inversé ou les noeuds de niveau supérieurs représentent les instances de composants et la racine, le type du composite dans lequel il apparaît. A titre d'exemple, dans la figure 1 nous avons un point d'accès dont le nom est **a.p0.d.App**. Lorsque le composite, représentant

l'application **App**, sera instancié dans une application **App** avec le nom d'instance **app00**, alors le point d'accès aura le nom **a.p0.d.app00.App.**

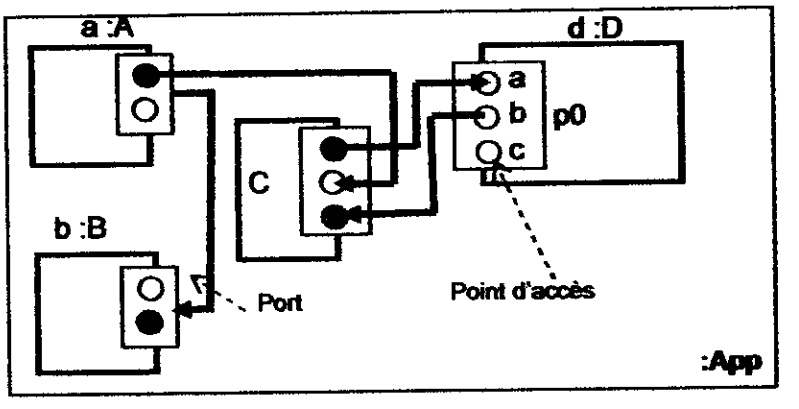

Figure 1: Connexions utilisant les points d'accès

I.2.2 Les types fondamentaux relatifs aux points d'accès :

Les points d'accès sont tous représentés par le type de base *AnyPoint* (Figure 2). Le type *AnyPoint* possède un nom d'instance et deux attributs décrivant le mode d'interaction du point d'accès (*synchrone, asynchrone*) et le temps de validité de l'information véhiculée par le point d'accès (*PresenceMode*).

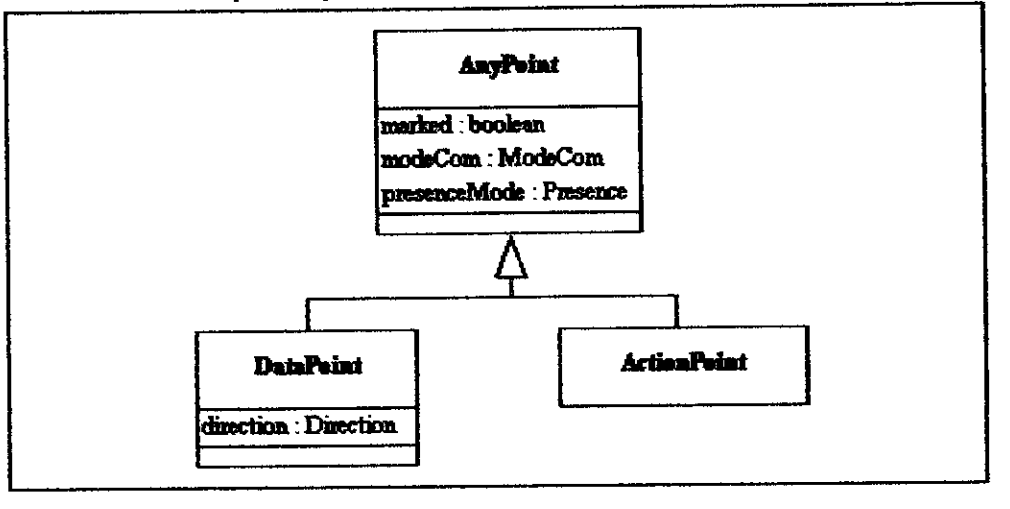

Figure 2: Diagramme de classes du point d'accès

Un point d'accès peut être ou non marqué (*marked, unmarked*). Un point d'accès marqué est un point d'accès correctement connecté. Cette caractéristique est très utile dans le processus de validation d'une architecture et est exploitée dans un processus de conception du général vers le particulier avec validation progressive

de l'architecture durant les différentes phases de conception, sans nécessité que les composants utilisés soient effectivement réalisés.

Dans l'état actuel, le point d'accès est destiné à supporter deux concepts : le transfert de donnée et le transfert de flux. Le transfert de flux correspond souvent à l'invocation synchrone ou asynchrone d'un service. Chacun de ces deux concepts est supporté par un point d'accès spécifique (Figure 2) :

- ✓ Le point d'accès dédié aux transferts de données est représenté par le type *DataPoint*
- ✓ Le point d'accès dédié au transfert de flux de contrôle (invocation d'une action, reprise du flux de contrôle) est représenté par le type *ActionPoint*.

I.2.3 Les points d'accès données : le type *DataPoint*

Un *DataPoint* est utilisé pour spécifier un transfert explicite de données. L'architecte peut utiliser un ensemble de *DataPoint* prédéfini ou définir un *DataPoint* qui lui est spécifique. Les *DataPoint* prédéfinis englobent les types de données primitifs que nous retrouvons dans les divers langages de programmation (entier, réel, caractère, booléen) ainsi que des types spécifiques à notre approche, tels que les types de données renseignant sur l'état structurel d'un composant composite. Dans le contexte actuel où le langage Java est utilisé comme cible dans les diverses opérations de validation de modèles, les types primitifs associés aux *DataPoint* correspondent aux types primitifs Java.

La définition de nouveau *DataPoint* spécifique doit suivre un style de nommage et une méthodologie de définition bien précise. Le nom du nouveau type commence par le nom du type associé au point d'accès et se terminer par le nom *DataPoint*.

La méthodologie de construction d'un nouveau point d'accès donnée consiste principalement à définir dans le port la zone qui contiendra la donnée et pourvoir le point d'accès de fonctionnalités usuelles tels que les accesseurs (méthodes *get* et *set*), la copie de données de point d'accès vers un point d'accès (méthode *copy*) et

l'instanciation d'un point d'accès à partir d'un autre point d'accès ou du type de donnée auquel il correspond.

Un *DataPoint* peut être associé à une ou plusieurs entrées ou sorties d'une action. Les point d'entrées et sorties d'actions sont représentés par le type *ActionPin*.

Un point d'accès de type *DataPoint* est doté d'attributs indiquant le sens des opérations de transfert de données (Figure 2). Trois valeurs sont possibles pour spécifier le sens des données : *in, out* et *inout*. Les points d'accès peuvent être explicitement spécifiés avec des valeurs d'initialisation.

I.2.4 Les points d'accès aux services : le type *actionPoint*

Un point d'accès de type *ActionPoint* indique qu'un service peut être initié à partir de ce point. Dans l'état actuel de la définition du modèle, nous considérons qu'un *ActionPoint* correspond uniquement à un seul service et un service permet de réaliser plusieurs actions.

Vis-à-vis d'un service, un point d'accès ne peut être que fournisseur ou client. Un point d'accès à travers lequel un service est fourni est représenté par le type spécifique *ServerActionPoint* (Figure 3). Lorsqu'un point d'accès spécifie un besoin de service, il est alors représenté par le type spécifique *ClientActionPoint* (Figure 3).

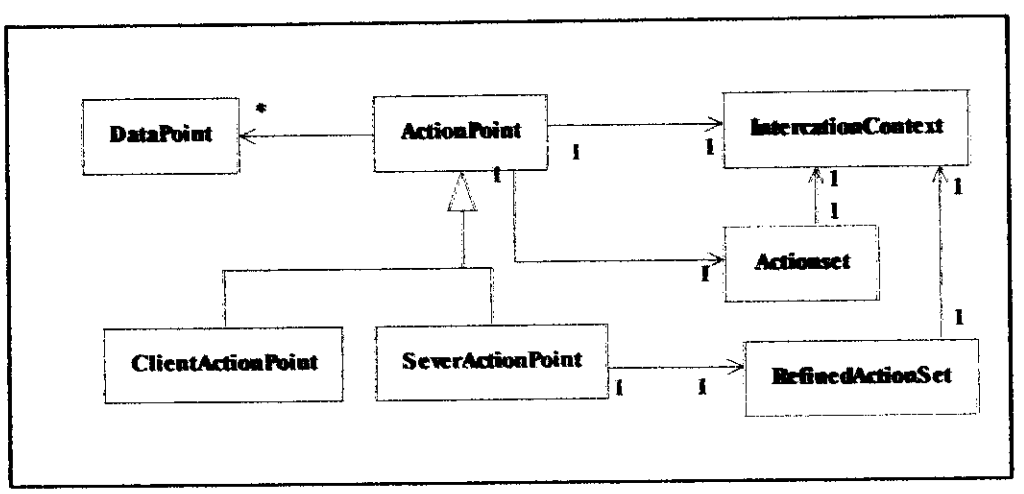

Figure 3: Diagramme de classe du point d'accès ActionPoint

I.2.5 Les points d'accès spécifiques

Ce sont des points d'accès orientés vers le support de la spécification explicite des divers aspects usuels, non métier, que nous trouvons dans tout logiciel. Dans notre cas nous distinguons les types suivants :

✓ Les points d'accès dédient au contrôle : L'objectif du contrôle est de permettre la spécification d'opérations permettant :

- La spécification de la disponibilité ou non d'une ressource(*EnableDataPoint*).
- La spécification du démarrage, arrêt, pause, reprise et redémarrage de service(*ControlledServerActionPoint*).

✓ Les points d'accès d'état : permettent d'exporter vers le monde externe les divers états d'un composant. Ils permettent aussi de définir une topologie bien précise exploitant ces états.

I.2.6 Les points d'accès exception

La gestion des erreurs est un des aspects que nous isolons des fonctions métiers d'un composant. Nous utilisons la technique des exceptions pour la signalisation et la prise en charge du traitement des erreurs. Les points d'accès d'exception sont représentés par le type *ExceptionDataPoint*.

I.3 Les Ports :

Un port est un regroupement de points d'accès, généralement, étroitement associés dans le contexte d'un objectif commun tel que la réalisation d'un service particulier. Tous les ports sont représentés par le type *AnyPort* (Figure 4). Un port possède un attribut renseignant sur le nom de l'instance. Le port représente un espace de nom pour les points d'accès. Chaque point d'accès est identifié de manière unique dans le contexte d'un port. Le composant représente un espace de nom pour un port. Ce dernier est identifié de manière unique dans un composant.

I.3.1 Les catégories de ports

La structure externe d'un composant fait ressortir des aspects qu'il est nécessaire d'isoler de l'aspect métier des composants. Ces aspects concernent la gestion des erreurs par exception, l'état des composants, les *logs* et les contrôles globaux sur les composants. Pour une prise en charge explicite de ces aspects, des ports dédiés à de tels aspects ont été prédéfinis.

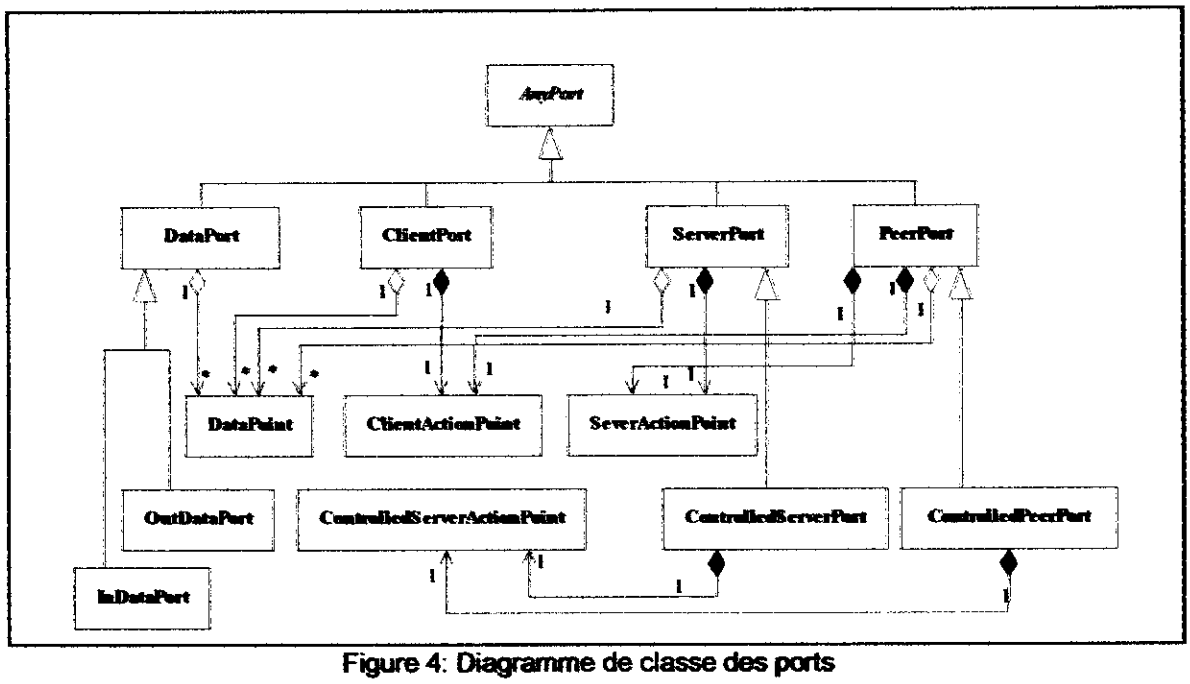

Figure 4: Diagramme de classe des ports

Les ports que nous utilisons pour atteindre les objectifs que nous venons de citer sont organisés en quatre catégories :

I.3.1.1 Les ports ordinaires

Nous avons défini 4 types de ports ordinaires qui répondent à une grande variété d'applications :

✓ Un port de données (*DataPort*) : peut contenir plusieurs point d'accès de données.

✓ Un port client (*ClientPort)* : contient un et un seul point d'accès *ClientActionPoint* et optionnellement un ou plusieurs point d'accès de données (*DataPoint*).

✓ Un port de service (*ServerPort*) : ne peut contenir qu'un point d'accès de service (*ServerActionPoint*). Les autres points d'accès optionnels ne peuvent être que des points d'accès de données.

✓ Un port de type *PeerPort* : doit contenir un point d'accès *ServerActionPoint*, un point d'accès *ClientActionPoint* et optionnellement plusieurs port de données (*DataPoint*).

I.3.1.2 Les ports contrôlés

Un port contrôlé supporte la spécification des diverses opérations de contrôle définies dans le contexte des actions de contrôle (*Enable, Disable, Start, Stop, Pause, Resume, et Restart*). Trois type des ports contrôlés sont pour l'instant identifiés: les ports de données contrôlés (*ControlledDataPort*), les ports de service contrôlés (*ControlledServerPort*) et les ports d'égal à égal contrôlés (*ControlledPeerPort*) . Un *controlledServerPort* et *un ControlledPeerPort* doivent obligatoirement contenir un *ControlledServerActionPoint*.

I.3.1.3 Les ports spécifiques, orientes aspects non métier :

Ces ports sont destinés à supporter les aspects tels que les exceptions, les états, les logs et le contrôle global sur le composant (Figure 5). Dans l'état actuel des travaux nous avons identifiés les 4 ports suivants :

- ✓ Le port principal du composant.
- ✓ Le port d'état du composant.
- ✓ Le port de gestion des états exceptionnel.
- ✓ Le port destiné aux logs.

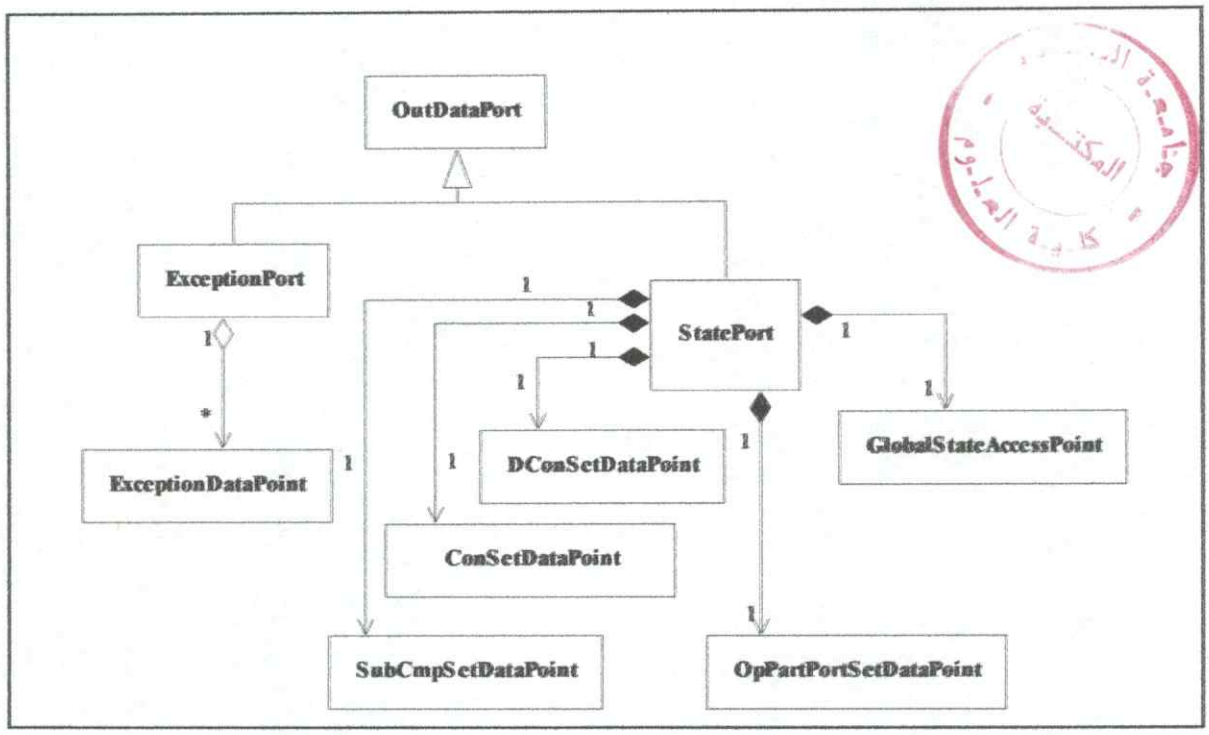

Figure 5: Diagramme de classes des ports d'exception et d'états

I.3.1.4 Les ports prédéfinis

Ce sont des ports orientés vers le support de connecteurs très répandus tels qu'une interaction dans le contexte d'un protocole standard de communication ou une interaction avec un environnement d'exploitation (système d'exploitation, serveur d'application). La grande différence entre ces ports et les autres ports résident dans les faits suivants :

- ✓ La vue concrète du port s'attache directement à une extrémité du connecteur standard correspondant et ne nécessite aucun service supplémentaire d'adaptation.
- ✓ Les ports prédéfinis peuvent contraindre le déploiement d'un composant à un nombre restreint de cas de déploiement.

I.4 Les Composants

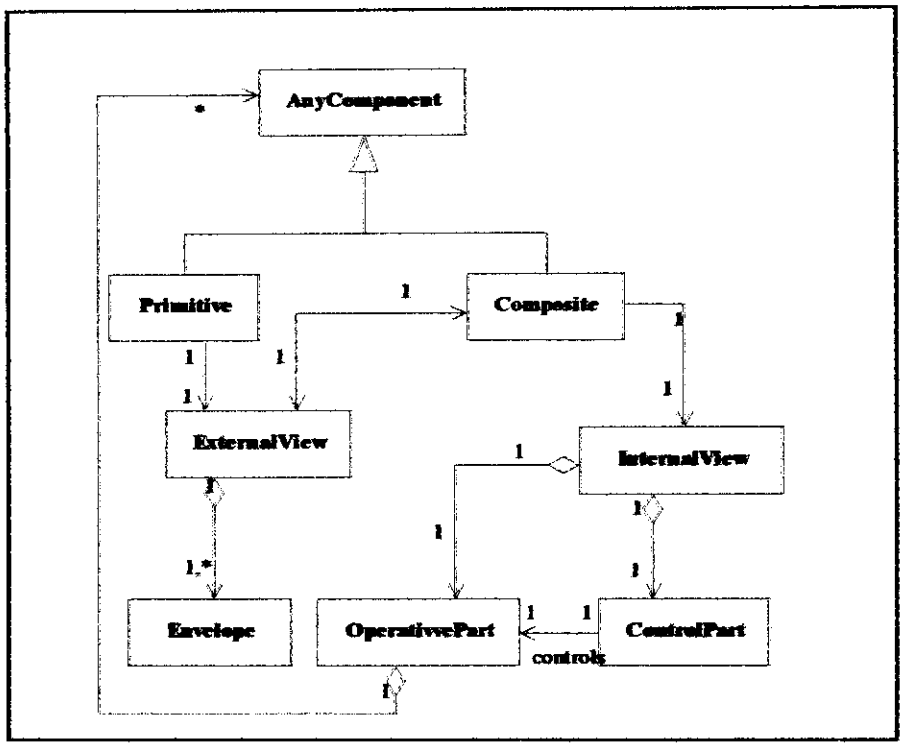

Figure 6: Diagramme de classe du modèle de composant

Le modèle de composant (Figure 6) distingue entre deux grandes catégories de composants: Les composants primitifs(*PrimitiveComponent*) et les composants composites(*CompositeComponent*).

Dans l'approche intégrée, l'objectif de l'élaboration d'une application consiste à réaliser un composant composite. Ainsi, à l'exception des connecteurs, tout est composant, de l'opérateur à une application toute entière. Le modèle de composant définit une organisation de la vue externe à laquelle doit adhérer n'importe quel composant (primitifs, composites, anciennes applications) et une organisation de la vue interne, applicable exclusivement aux composites[ISPS07i] . La vue externe est représentée par le concept d'enveloppe ou sont localisés les ports modélisant le composant. La vue interne est organisée en deux grandes partie : la partie opérative et la partie de contrôle.

I.4.1 La vue externe et le concept d'enveloppe

La vue externe de tout composant est formée d'une enveloppe munie d'un certain nombre de ports. Le concept d'enveloppe a été introduit pour permettre d'atteindre les objectifs suivants :

- ✓ L'isolation totale la vue interne d'un composant du monde externe.
- ✓ Offrir le support nécessaire à la réalisation de topologies .
- ✓ Offrir le support nécessaire au déploiement des instances d'un composant.
- ✓ Offrir le support nécessaire à la réalisation des opérations de validation.
- ✓ La transformation de n'importe quel composant, en un composant prêt à être exploité dans les opérations d'assemblage.
- ✓ Permet de supporter la spécification d'une logique dans laquelle un composite, et plus particulièrement sa tâche principale, peut posséder un cas de déploiement totalement différent de ses composantes.

Nous avons pour l'instant identifié deux sortes d'enveloppes : Les enveloppes marquées et les enveloppes de déploiement (Figure 7).

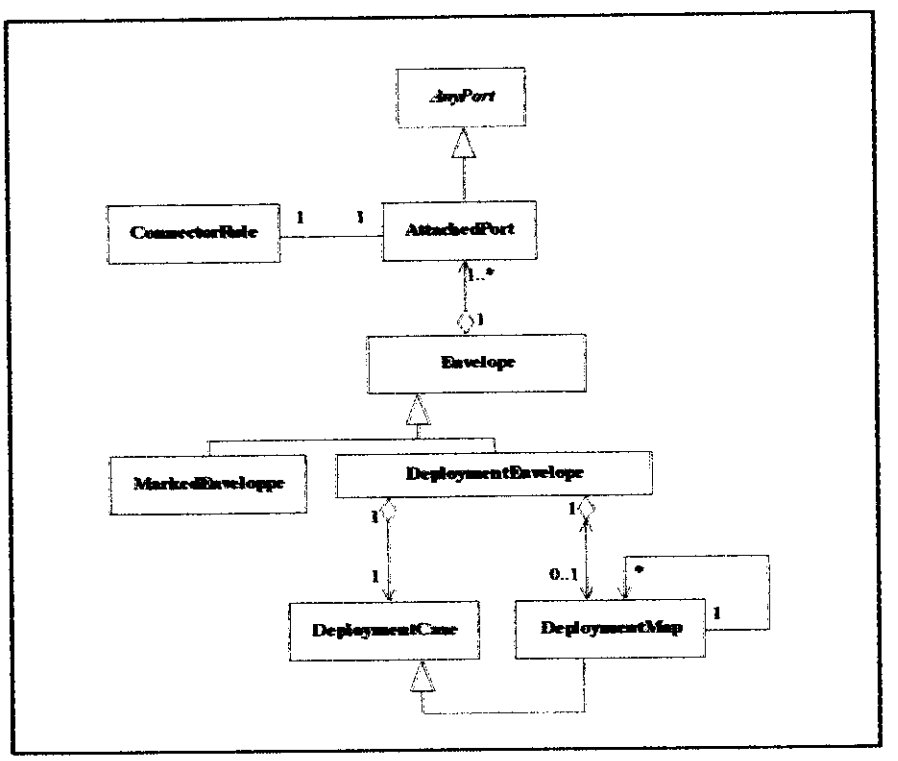

Figure 7: Diagramme de classes de l'enveloppe

I.4.1.1 Les enveloppes marquées

Les enveloppes marquées sont utilisées dans le processus de vérification de l'état global (stabilité) d'un composant. Dans de telles enveloppes, il est possible de marquer totalement ou partiellement les points d'accès. Le marquage d'un point

d'accès signifie que ce dernier est correctement connecté. Une enveloppe totalement marquée est une enveloppe dont tous les points d'accès sont marqués. Dans une enveloppe marquée partiellement vers l'extérieur seules les services requis (*ClientDataPoint*) et les *DataPoint* dans le sens est *in* ou *inout* sont marqués. Dans une enveloppe marquée partiellement vers l'intérieur, seules les services fournis (*ServerActionPoint*) et les *DataPoint* dans le sens est *out* ou *inout* sont marqués.

I.4.1.2 Les enveloppes de déploiement

Une enveloppe de déploiement est dotée d'un cas de déploiement. les cas de déploiement effectif de toutes les instances dans les divers niveaux de la hiérarchie de composition. Grâce à ce concept, il est très possible d'associer à des instances d'un même type de composant, des cas de déploiement totalement différents. L'application d'une enveloppe de déploiement se fait soit à l'instanciation d'un composant dans le processus de construction d'un composite soit au type de composant. Dans ce dernier cas, l'application de l'enveloppe a pour but de générer l'application[ISPS07i] .

I.4.2 Organisation de la vue interne

La vue interne est organisée en deux parties. Une partie opérative et une partie contrôle.

I.4.2.1 La partie opérative

Elle contient les composants représentants par leurs interconnexions la logique générale du composite à un moment bien précis. Elle est représentée par le type *OperativePart*.

Les instances de composants et connecteurs sont soit statiques soit dynamiques.

En plus des instances de composant, la partie opérative peut comporter la définition de type de composant internes. L'instanciation de ces types internes n'est possible qu'au niveau de la partie opérative dans, laquelle ils ont été définis.

I.4.2.2 La partie contrôle

La partie contrôle, représentée par le type *ControlPart* (Figure 8), 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.

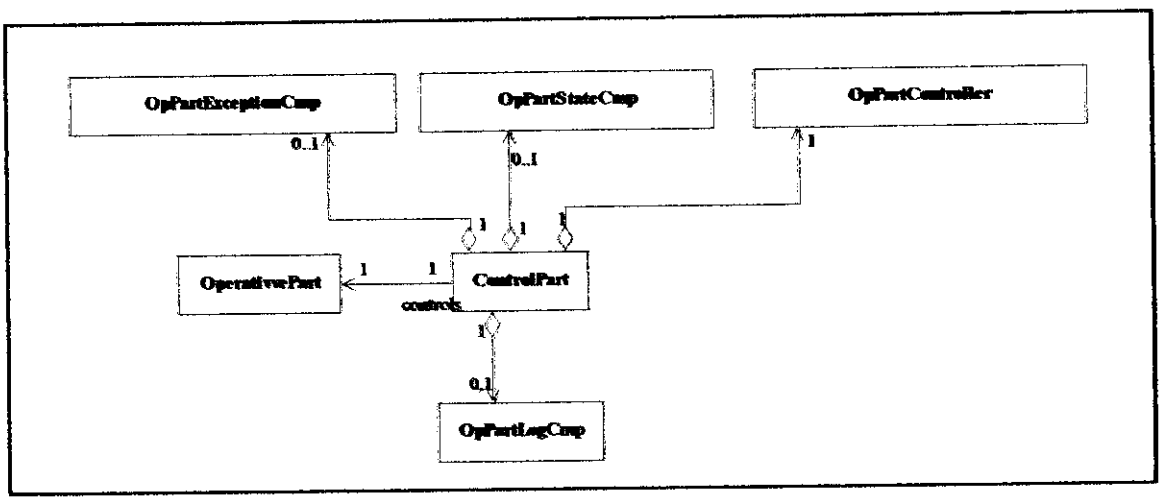

Figure 8: Diagramme de classe de la partie contrôle

Les opérations de la partie contrôle sont assurés par quatre composants spécifiques (Figure8) , L'*OpPartController*, l'*OpPartStateCmp*, l'*OpPartExceptionCmp* et l'*OpPartLogCmp*. La partie contrôle ne peut contenir qu'une seule instance des types de composants qui lui sont spécifiques. L'instance de l'*OpPartController* est obligatoire alors que les instances des trois autres types sont optionnelles.

Les actions qu'applique le contrôleur (*OpPartController)* à la partie opérative font partie d'un contexte d'action bien précis (*ControlActionContext*). Ce contexte adresse entre autres les aspects suivants :

- ✓ La sélection/chargement d'un comportement
- ✓ Le contrôle global, partiel ou individuel des composants de la partie opérative (activation/désactivation)
- ✓ Gestion du transfert du flux de contrôle entre les composants de la partie opérative (lancement séquentiel ou parallèle de service).

- ✓ L'évolution dynamique de la partie opérative (création/ destruction de composants et de connecteurs).

Enfin le composant *OpPartController* reporte au composant d'états l'état structurel initial et tout changement structurel durant l'exécution.

I.4.3 Les composants usuels

Parmi ces composants nous trouvons les divers opérateurs, les composants d'E/S, les composants visuels, les composant d'accès aux bases de données, les clients de protocole standards et les composants de spécification de partage. Dans ce qui suit nous introduisons brièvement un certains nombre de composant usuels.

I.4.3.1 Les operateurs

Ils permettent d'exprimer dans les divers niveaux d'abstraction, les opérations usuelles que nous retrouvons dans les langages de programmation. A titre d'exemple, l'opérateur *plus* est représenté par le type *PlusOpCmp* qui est la racine de toutes les opérations utilisant le signe plus. Le composant *PlusOpCmp* possède un *DataPoint* en *out* et deux ou plusieurs *DataPoint* en *in* (Figure 9).

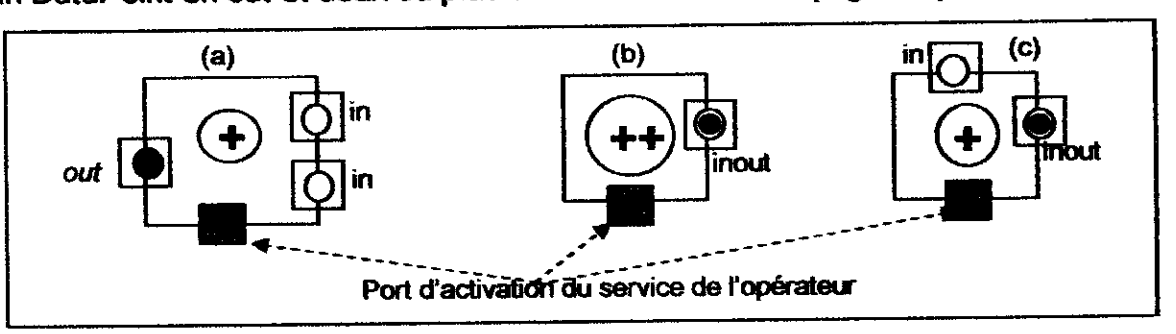

Figure 9: Les opérateurs Plus, PlusPlus et Incrementation

La figure9 montre les figures de deux autres composants opérateurs. L'opérateur d'incrémentation d'un nombre entier (*PlusPlusIntOpCmp* ,Figure 9.b) et le composant d'ajout d'une quantité à un nombre ou d'ajout d'un élément à une collection (*IncrementOpCmp*, Figure 9.c).

I.4.3.2 Les composants d'initialisation

Ils Permettent la spécification de valeurs initiales et constantes pour des *DataPoint* dont le sens est *in*. Quand une instance d'un composant d'initialisation est

ajoutée à une architecture, l'architecte doit spécifier une valeur dans l'initialiseur de chaque point d'accès.

I.4.3.3 Les bouchons

Les bouchons (*LoopBackCmp*) ont été introduits pour vérifier la stabilité d'une architecture durant les étapes intermédiaires d'un processus de conception. Ce composant est doté d'un ou plusieurs ports de type *ServerPort*, fournissant des services vides ou des services de tests, réalisant toutes les actions spécifiées dans le port client correspondant.

I.4.3.4 **Les composant de terre:** (*GroundCmp*)

Ils permettent de spécifier le non intérêt d'une donnée qui sortirait d'un *DataPoint*. C'est une sorte de trou noir. Un *GroundCmp* est composé d'un ou plusieurs *InDataPort*.

I.4.3.5 Les selecteurs

Nous distinguons deux types de sélecteurs : le sélecteur des entrées (*InSelectorCmp*) et le sélecteur des sorties (*OutSelectorCmp*) (Figure 10).

Un sélecteur des entrées est composé de plusieurs entrées, une sortie et un port de sélection des entrées. Les entrées et sorties sont soit toutes des *DataPort* soit toutes des *ActionPorts*. Le port de sélection est un *IntegerDataPort* ayant le sens *in* C'est à travers le port de sélection qu'un port d'entrée bien précis sera connecté à la sortie.

Le sélecteur des sorties réalise l'opération inverse d'un sélecteur d'entrée, en reportant l'entrée vers la sortie sélectionnée.

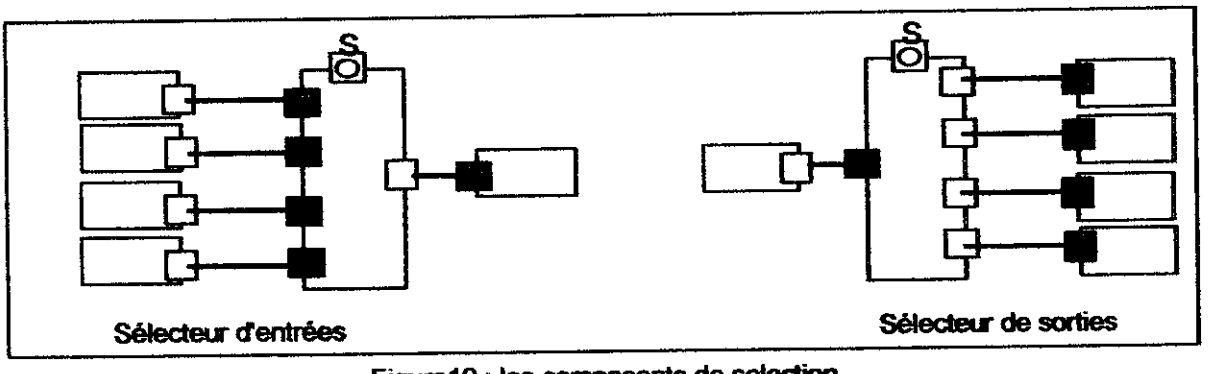

Figure10 : les composants de selection

I.4.3.6 Les composants de spécification du partage

Ces composants n'ont pas d'existence réelle là ou ils sont instanciés. Ils jouent le rôle de représentant d'autres instances de composant. Ces composants sont dotés d'un nom local et du nom absolu de l'instance qu'ils représentent. Le nom local est une sorte d'alias et le composant lui même peut être pensé comme étant un lien symbolique vers une instance de composant.

Deux objectifs principaux sont recherchés à travers ces composants :

- ✓ la réutilisation de composant qui n'existe dans une application qu'à travers une seule instance tels qu'un composant représentant un système de fichiers, un composant représentant un serveur de base de données et un composant représentant un serveur HTTP.
- ✓ Le deuxième objectif est la clarté de la spécification d'une architecture.

I.4.3.7 Les composants visuels

I.4.3.7.1 Le modèle de représentation des composants visuels

Actuellement la majeure partie des applications utilisent les composants visuels dans le contexte de la réalisation de leurs interfaces homme machines. Dans l'approche intégrée, les composants visuels sont représentés par un modèle général (Figure 11 et 12).qu'il est nécessaire d'adapter d'une part pour représenter globalement chaque composant visuel et d'autre part pour représenter le composant visuel dans un environnement d'instanciation particulier.

La vue externe du composant, selon le modèle de représentation des composants visuel, possède trois groupes de ports, chacun étant représenté sur une face (figure 11).

- ✓ Les ports qui seront activés suite à l'apparition d'un événement au niveau d'un composant visuel (ports de la face du haut).
- ✓ Les ports à travers lesquels seront déclenchés les services associés à la logique de prise en charge d'un événement (ports de la face gauche).
- ✓ Les ports permettant d'accéder aux spécificités du composant visuel (accès aux attributs, simulation de déclenchement d'événement).

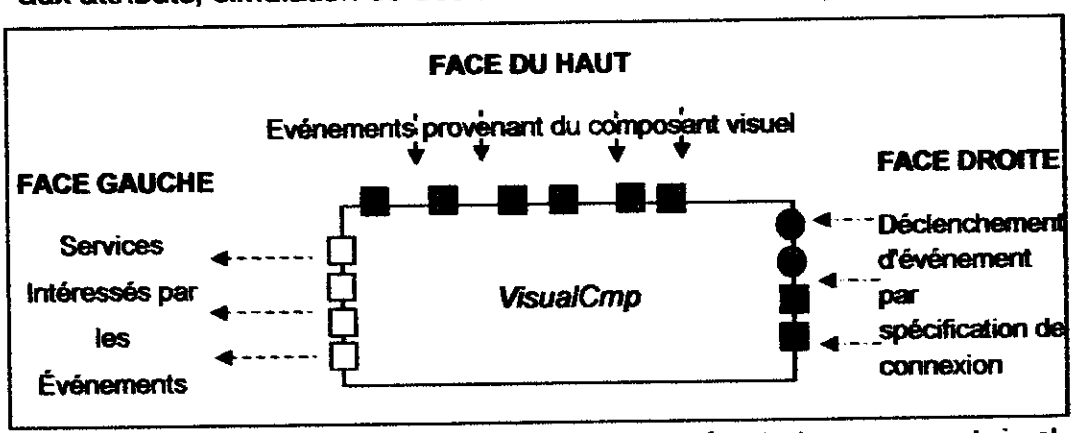

Figure 11: Modèle de la vue externe d'un composant représentant un composant visuel

La vue interne du modèle est organisée de la même manière que n'importe quel composite. Elle est constituée d'une partie opérative et une partie contrôle (Figure 12). La partie opérative contient deux parties :

- ✓ Une partie qui interagir directement avec le composant visuel associé.
- ✓ Une partie représentant les instances de composants rentrant dans la définition structurelle du composant visuel.

Deux objectifs principaux sont assignés à la vue interne modélisant un composant visuel:

- ✓ La spécification de la logique de prise en charge d'un événement.
- ✓ Le déclenchement d'événement par programme.

La logique de distribution des événements aux intéressés, écrite dans le

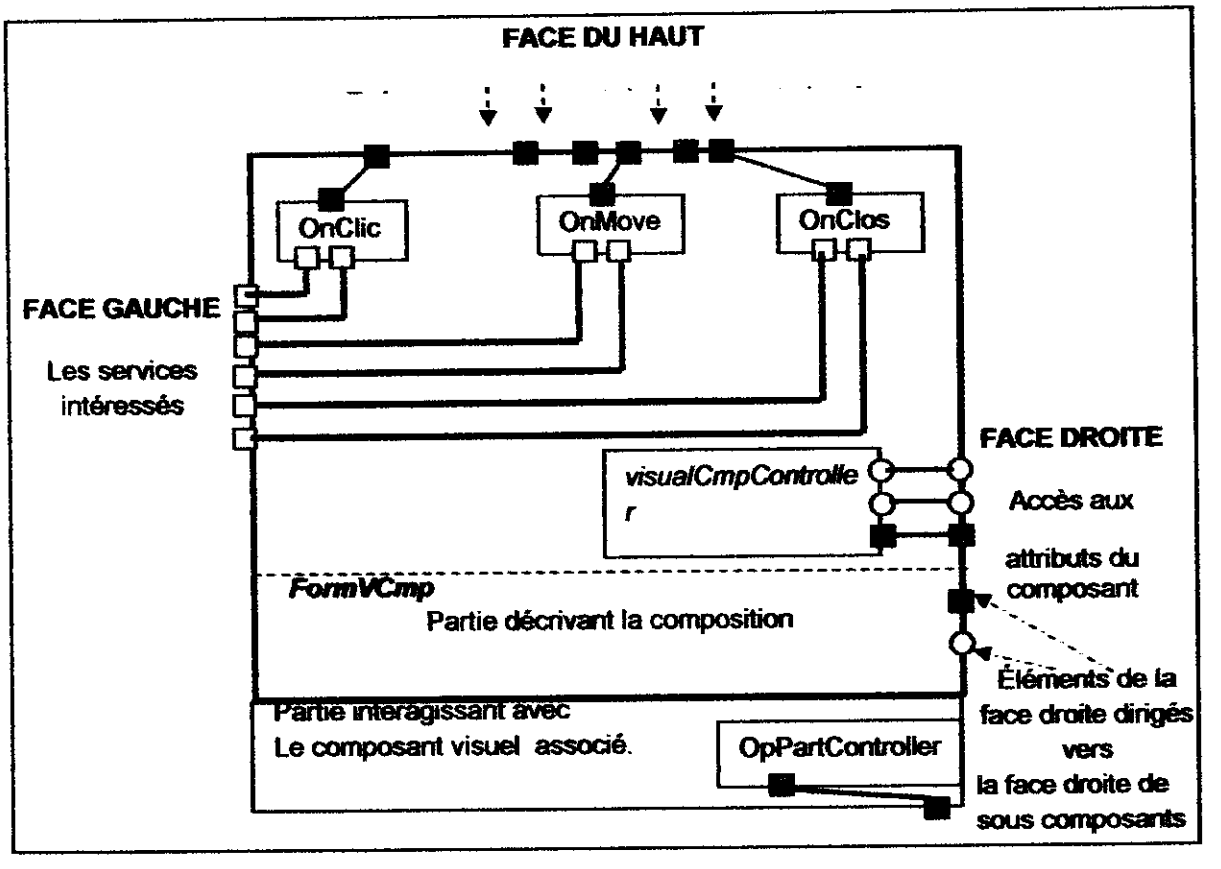

Figure12: Vue interne du modèle de composant associé aux copoosants visuels

I.4.3.7.2 Les types associés aux composants visuels

Chaque composant visuel est associé à un type de composant le représentant dans une spécification architecturale. Chaque type de composant possède:

- ✓ Un nombre fixe de *ServerPort* (port de la face haute et de la face droite du modèle) représentant chacun un événement pouvant se produire sur le composant visuel.
- ✓ Un nombre fixe de ports (*DataPort* et *ServerPort*) associé aux attributs du composant visuel et à ses méthodes.
- ✓ Un nombre variable de port client sur la face gauche. Cette variation est du à l'environnement d'instanciation du composant visuel.

Par défaut, le nombre de ports client (face gauche) est égal au nombre d'événements. Chaque port client correspond à un port de service (face haute).

I.4.3.7.3 Association de types internes aux composants visuels

Lors de l'instanciation d'un composant visuel, le schéma d'activation des services associés à un événement est souvent différent du schéma par défaut que nous venons juste de présenter. Ainsi dans une même fenêtre plusieurs composants visuels d'un même type (i.e. un boutton) peuvent être instanciés avec des schémas d'activation de services totalement différents. Souvent, chaque schéma d'activation est très spécifique, local et non réutilisable dans d'autres situations. Pour répondre à de telles situations, le modèle utilise la notion de type interne.

Le type interne est construit juste pour prendre en charge une situation d'instanciation spécifique d'un composant visuel. La définition d'un type interne associé au composant visuel se traduit par les actions suivantes :

- ✓ Ajout de port client au niveau externe et au niveau des composants de comportement et établissement des connecteurs de délégation.
- ✓ Redéfinition, si nécessaire du comportement des composants comportementaux.
- ✓ Ajout de composant visuels dans le cas ou le composant visuel en cours est un composite.

I.4.3.7.4 Les composants visuels dans une spécification d'architecture

Lors de la spécification d'une architecture utilisant les composants visuels, les événements provoqués sur ces derniers proviennent souvent de sources communes (clavier et souris). Dans de tels cas il n'est pas nécessaire de reporter dans une spécification d'architecture les ports d'arrivée d'événements (port de la face haute).

Lors de la spécification d'une connexion générant un événement sur la face gauche d'un composant associée à un composant visuel, le composant *visualCmpController* de la partie opérative ne relayera l'événement au composant visuel associé que si l'événement est effectivement connecté à un service intéressé (face gauche).

I.4.4 Les composants de la partie contrôle

En plus du composant contrôleur que nous avons présentés précédemment, la partie contrôle contient trois composants spécifiques : Le composant d'état, le composant de gestion des exceptions et le composant de logs.

I.4.4.1 Le composant d'état : (*OpPartStateCmp*)

Le composant d'états est conçu pour être utilisé uniquement dans la partie contrôle. Il est utilisé pour collecter les divers états de la partie opérative. Les diverses listes sont exportées vers le monde extérieur à travers le port d'états (*StatePort*) dont est doté chaque composant composite.

La connexion entre le contrôleur et le composant d'état est utilisée par le contrôleur pour envoyer au composant d'états l'état structurel de la partie opérative. Ces opérations de transferts ont lieu à chaque opération impliquant un changement de la structure (ajout / suppression / activation / désactivation de composant, de ports et de connecteurs).

I.4.4.2 Le composant de gestion des exceptions : (*oppartexceptioncmp*)

Ce composant joue le rôle principal de mise en zone de garde des divers composants, sources d'exception. Si tous les composants sont mis en zone gardée, alors le composite ne sera pas une source d'erreurs pour ses utilisateurs.

I.4.4.3 Le composant log : *(oppartlogcmp)*

Les objectifs suivants ont été à l'origine de la nécessité de doter la partie contrôle de ce composant spécifique.

- ✓ Faire ressortir de manière explicite l'aspect log dans les applications.
- ✓ Définir un lieu commun ou seront spécifiés les divers formats de log.
- ✓ Permettre de transmettre les logs vers diverses destinations telles qu'un fichier, une base de données ou une quelconque application intéressée par l'analyse des logs.

Comme les autres composants, le composant de logs peut être instancié en actif ou en passif. L'instanciation en mode actif permet à ce composant de se

connecter automatiquement à tout port de log de n'importe quel composant de la partie opérative.

I.4.5 Etat globaux et qualificateurs de composant

Nous avons définis trois états globaux renseignant sur le composant durant le cycle de conception : l'état stable, l'état instable et l'état opérationnel. Les qualificateurs de composant renseignent sur le type de manipulation qu'il est possible de réaliser sur un composant.

I.4.5.1 Stabilité de composant

La stabilité renseigne sur la validité d'une conception soit en statique ou en dynamique (évolution structurelle). Nous avons défini deux stabilités : stabilité simple et stabilité totale.

Un composant est dit stable (stabilité simple) si toutes les ressources fournies paraissent complètement définies lorsque nous appliquons une enveloppe marquée vers l'extérieur sur ce composant. Le contrôle de stabilité ne prend pas en considération les points d'accès dont la direction est *out*. Ainsi dans un composant stable il est très possible que des points d'accès dont la direction est *out* ne soient pas connectés.

I.4.5.2 Composant opérationnel

Un composant stable peut être opérationnel ou non opérationnel. Un composant opérationnel est un composant stable doté d'un plan de déploiement. Un composant opérationnel est un composant prêt à être généré et déployé.

I.4.5.3 Qualificateurs de composant

Les opérations de modification structurelles et comportementales menées sur les composants se résument à la création et suppression d'instances de composants, de ports et de connecteurs, à la modification du comportement de composants comportementaux, et à la spécification de propriétés non fonctionnelles aux instances (Actuellement, nous nous limitons à la propriété déploiement des instances de composants). Ces diverses opérations peuvent avoir de lourdes

conséquences et mener à l'incohérence d'une architecture si elles ne sont pas régulées. Pour limiter l'impact de ces opérations, nous avons défini deux grandes catégories de composants :

✓ La première catégorie englobe les composants pouvant être échangés entre projets. Les composants de ces catégories sont qualifiés par l'un des qualificateurs suivants : *composant final* et *composant fermé*.

❖ Un composant final est un composant opérationnel, dont la vue interne n'est pas accessible.

❖ Un composant fermé est un composant final, doté d'une seule enveloppe de déploiement. A l'exception des opérations d'instanciation avec l'enveloppe fournie, aucune opération n'est possible sur un composant final.

✓ La deuxième catégorie est représentée par les composants de projet. Ces composants sont à tout moment dans l'un des états suivants : *stable, opérationnel, instable*.

I.4.6 Déploiement des composants

Le modèle de composant que nous avons défini permet une haute flexibilité dans la spécification des propriétés de déploiement des composants. Ces propriétés sont utilisées pour produire le code, et éventuellement les descripteurs de déploiement effectif nécessaires[ISPS07i] . La spécification du déploiement consiste à définir pour une instance de composant un cas de déploiement et un plan de déploiement (Figure13).

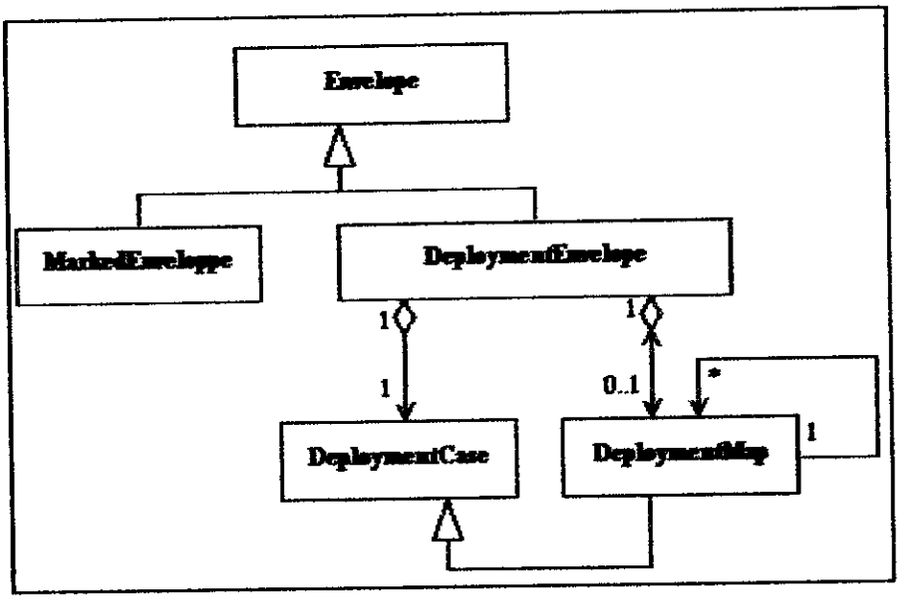

Figure 13: Diagramme de classe du cas et plan de déploiement

I.4.6.1 Les cas de déploiement

Le cas de déploiement spécifie l'état du composant à l'exécution et l'environnement dans lequel le composant sera exécuté.

Nous avons identifié 6 cas de déploiement direct : *Process, MainThread, Thread, Applet, Servlet, Ejb*[1]. Il est clair que cette liste est extensible pour la prise en charge de cas de déploiement divers.

Chaque cas de déploiement doit être spécifié avec son environnement. .

I.4.6.2 Plan de déploiement

La spécification du déploiement est réalisée en deux grandes étapes. Dans la première étape, l'architecte définit les diverses enveloppes pour un type de composant. Dans une deuxième étape, et pour chaque instance d'un composite il applique une enveloppe qui aurait été défini au préalable et de ce fait choisi pour l'instance un cas de déploiement. Ainsi, dans le même composite, il est fort probable que deux instances d'un même type de composant aient des cas de déploiement complètement différents.

[1] Le choix de ces cas est du à l'utilisation du langage JAVA qui dispose d'une grande variété de technologies d'implémentation

Si un composant est pourvu d'au moins un plan de déploiement, ce composant devient *opérationnel* et les divers codes et descripteurs de déploiement effectifs de l'application peuvent être générés.

I.4.6.3Le concept de distance entre composant

Lorsque deux instances de composants sont connectées, le connecteur qui les relie ne sera pas le même pour tous les cas de déploiement des deux instances. Dépendant du cas de déploiement associé à chaque instance, une distance sépare les composants connectés. Selon cette distance le connecteur aura une structure particulière [ISPS07i] .

Selon les cas de déploiement associés aux instances connecté, la distance devient plus ou moins importante. Nous avons défini quatre distances entre composants.

Directe : La distance est dite directe lorsque les deux instances opèrent dans une même tâche.

Locale : La distance est dite locale si les ports connectés appartiennent à des composant qui évoluent dans deux taches différentes d'un même processus.

Moyenne : Elle est moyenne lorsque les deux composants appartiennent à des processus différents, qui évoluent dans un même environnement (même système d'exploitation, même serveur d'application).

Etendue : Les composants évoluent dans des environnements distincts (systèmes d'exploitation sur machines différentes, serveurs d'application sur machines différentes).

Lorsque la distance est directe, les deux composants opèrent dans une même tâche (cas de déploiement *MainThread, Thread*). Le connecteur est alors dit direct et est réalisé par des techniques locales à la tâche. Ces techniques ne permettent pas d'aller au delà de la tâche. Les techniques les plus usuelles sont l'appel de

procédure et les variables partagées. Dans une même tâche, les deux composants ne peuvent pas évoluer en parallèle. Si une ressource (un port) devait être accédée par les divers composants, l'environnement de déploiement par sa nature résous le problème d'accès. Cependant, puisque il est très possible que les composants utilisés, pourraient avoir été développés par des sources différentes, il n'est pas évident que les ports interconnectés soient totalement compatibles du moins au niveau des noms. Une couche d'adaptation devient nécessaire pour représenter un connecteur direct.

Si les composants sont séparés par une distance locale, les deux composants opèrent dans deux taches différentes (cas de déploiement *Thread*) dans un même processus. Le connecteur est dit local. L'accès à une ressource partagée (représenté par un port) doit être régulé. Dans ce cas le connecteur local ne sera pas uniquement un simple appel de méthode ou un simple accès à une variable. Une logique de synchronisation des accès doit être implémentée par le connecteur. Une telle logique ne doit pas faire partie du port du composant, car elle deviendrait encombrante et inutile dans le cas ou les deux autres instances des deux composants son connectés par un connecteur directe. Un connecteur local préserve la forme originale de l'information présentée sur les ports connectés.

Lorsque la distance est moyenne, les composants opèrent dans un environnement commun. Pour interagir, ils doivent utiliser des mécanismes de communication fournis par l'environnement commun. A titre d'exemple, si nous disposons d'une architecture en *pipe et filtre*, et que les composants évoluent dans le système *UNIX*, les mécanismes usuels pour supporter une telle architecture pourrait être les *pipe Unix* ou les *sockets Unix*. Parfois les mécanismes de communication de l'environnement transforme l'information avant sa transmission et la remet en état pour la destination.

Lorsque les composants sont trop distants, il devient nécessaire d'utiliser des mécanismes de communication fournis par des environnements différents. Des environnements distincts peuvent offrir des mécanismes de communication incompatible.

Dans ce cas le connecteur devra jouer en plus du rôle de transport, un rôle de passerelle entre port utilisant des protocoles de communication incompatibles. Malgré que ce cas puisse paraître rare, il pourrait devenir fréquent avec la poussée des modèles de composant objets tels que les *EJB*, les *CCM* et les composants *.Net*. Les environnements basés sur ces composants utilisent des mécanismes totalement incompatible (RMI pour le premier, bus CORBA pour le second et Web services pour le troisième). Lorsque la distance est étendue certains mécanismes de communication dans le contexte d'un environnement ne sont plus valables. C'est le cas par exemple des *pipes Unix* qui ne permettent pas de sortir d'une machine.

I.5 LES CONNECTEURS

I.5.1 Les connecteurs de l'approche intégrée

Lors de la définition des connexions entre les ports de composants, l'architecte spécifie parfois une interaction et parfois indique que la connexion doit être réalisée par une technologie bien connue tels qu'un protocole de communication ou un appel distant de procédure dans le contexte d'une infrastructure de communication très répandue (Bus CORBA, RMI).

Lorsqu'il définit une connexion comme étant supportée par une technologie bien précise, l'architecte spécifie parfois une interaction en utilisant des actions que peut supporter la technologie choisie. A titre d'exemple si la technologie choisie est un protocole de communication, l'architecte ne peut utiliser un vocabulaire étranger à ce protocole.

Il est possible dans une architecture de retrouver les mêmes connecteurs instanciés plusieurs fois (Figure 14). Dans ces diverses instances, le connecteur possède la même vue abstraite. Cependant, le cas de déploiement des instances de composants qu'il relie peuvent être complètement différents. Il est clair que le cas de déploiement peut avoir une conséquence directe sur la technologie d'interconnexion à utiliser. Il est donc possible que le connecteur *c1* soit un appel de procédure et que le connecteur *c2* soit un *RPC*.

Chaque instance de connecteur possède un nom qui l'identifie de manière unique dans une application. Le nom du connecteur est défini dans un espace de nom représenté par le composite dans lequel le connecteur est instancié.

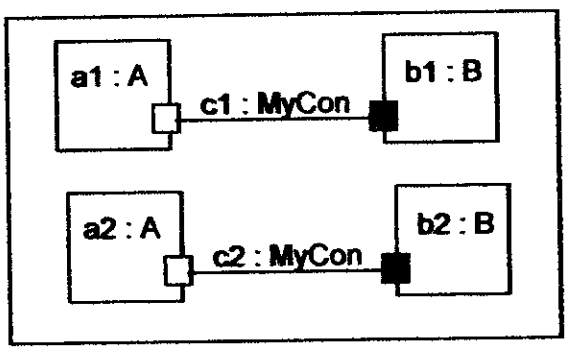

Figure14: Instanciation de connecteurs

Pour supporter ces diverses situations, le modèle de connecteur que nous avons défini est basé sur deux grandes vues (Figure 15) : Une vue abstraite et une vue concrète. La correspondance entre connecteur abstrait et concret est assurée au niveau connecteur. Afin de pouvoir spécifier librement diverses topologies, le modèle a défini deux catégories de connecteurs (Figure 16) : 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.

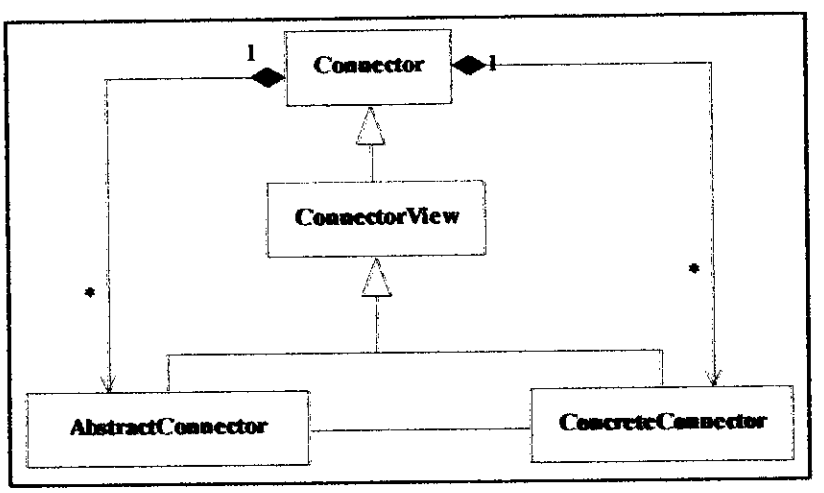

Figure15: Les deux vues d'un connecteur

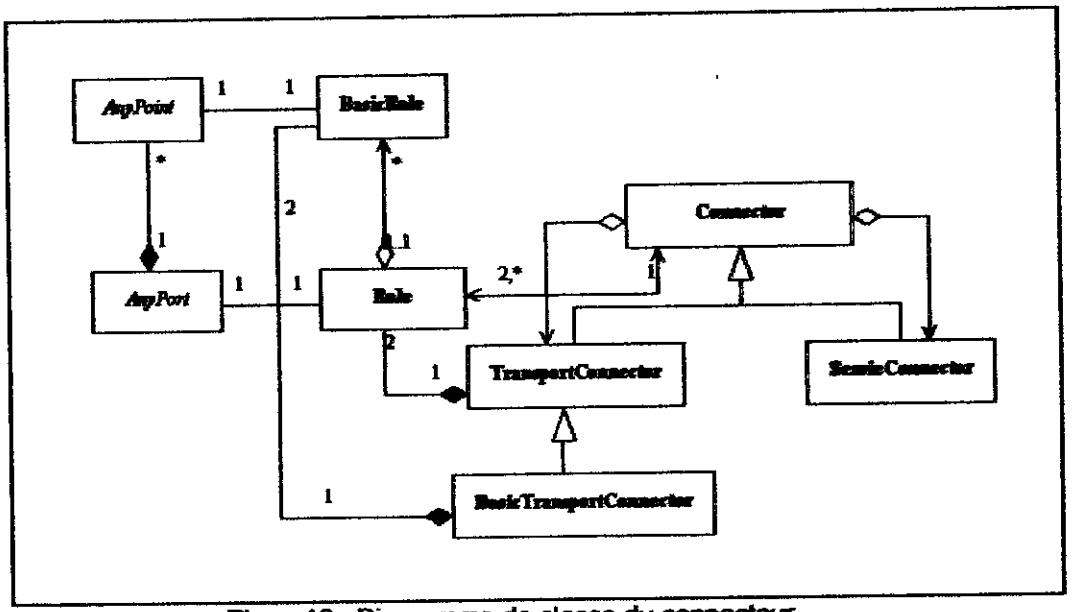

Figure16 : Diagramme de classe du connecteur

Nous distinguons entre deux catégories de connecteurs point à point : Les connecteurs simples liant deux ports et les connecteurs élémentaires liants deux point d'accès. Un connecteur élémentaire permet la spécification des connexions de manière très flexible et sans contrainte particulière. C'est ainsi qu'il devient possible d'envoyer un point d'accès vers plusieurs autres points d'accès de ports différents (figure 17) à l'aide d'un composant de service dédié à la répétition ou la distribution.

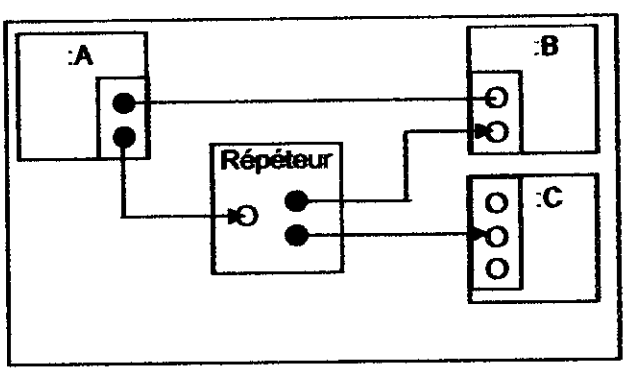

Figure17: Connecteurs élémentaires

La distinction entre connecteurs simples et connecteurs élémentaires ouvre la voie vers la création de connecteurs à base de connecteurs élémentaires (composition, décomposition) et aussi vers la réalisation d'opération dynamique sur la structure d'un connecteur.

I.5.2 La vue abstraite

Les connecteurs abstraits *(AbstractConnector)*(Figure18) sont utilisés d'une part pour permette une validation d'architecture dans les diverses phases d'un processus de conception top down, notamment dans les niveaux abstraits, et d'autre part pour spécifier des connexions qui ne contraignent pas le déploiement des composant à des cas de déploiement particuliers.

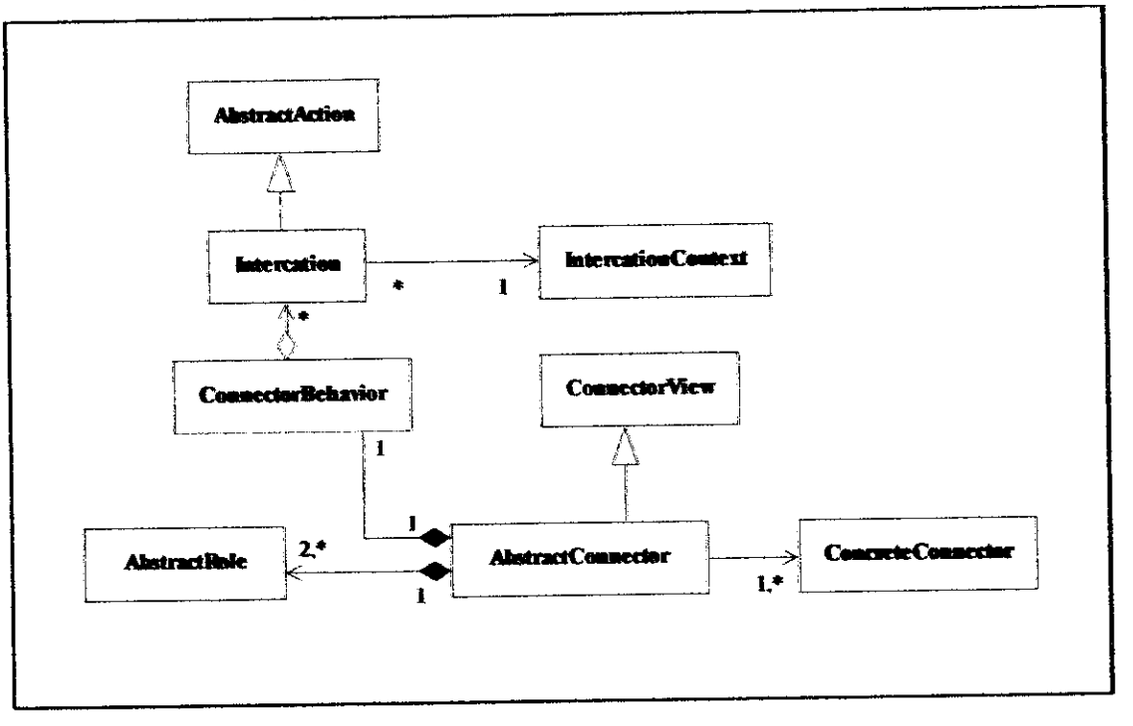

Figure 18: Diagramme de classe du connecteur abstrait

I.5.3 La vue concrète

La vue concrète d'un connecteur (*ConcreteConnector*) est soit spécifiée de manière explicite, soit générée automatiquement à partir des cas de déploiement associés aux composants interconnectés. La spécification explicite concerne les connecteurs standard ou prédéfinis. Dans l'approche de modèle IASA les connecteurs de services sont tous prédéfinis.

I.5.4 Les ports spécifiques aux connecteurs

Les connecteurs de transport ou de service peuvent être dotés de ports spécifiques. Ces ports permettent :

- ✓ L'exportation des états du connecteur (trace).

✓ La gestion des erreurs exceptionnelles suite à un problème au niveau du connecteur (coupure de la connexion, dépassement de temps) ou le non respect des spécifications (interaction non conforme au protocole d'interaction défini dans le comportement du connecteur)

✓ L'accès par le connecteur aux services d'une infrastructure de communication (i.e. résolution de nom).

✓ La réalisation d'opération de contrôle (Initialisation, arrêt, démarrage et isolation du connecteur des ports, le changement de ses attributs, Le changement de sa structure (i.e. ajout, suppression de rôles)).

✓ Le changement de comportement.

I.5.5 Génération de connecteurs de transport

I.5.5.1 La résolution du problème d'adaptation d'interface

L'adaptation de port est prise en charge au niveau du connecteur concret. C'est un aspect de niveau implémentation. Lorsque deux ports sont interconnectés, il n'est pas évident que le processus de normalisation transformant la structure du port en structure d'interface (supporté par les langages standards tels UML, les ADL tels ArchJava ou directement le langage Java) aboutissent à des interfaces ayant une correspondance totale en termes de noms de méthodes et ordre de spécification des paramètres. Pour rendre opérationnelle la connexion entre de tels ports, et plus exactement entre deux interfaces, une adaptation s'avère nécessaire.

La stratégie d'adaptation que le modèle a adoptée se base sur d'une part le patron de conception (*design pattern*) adaptateur décrit dans [ISPS07i] et d'autre part sur les principes suivants :

- Le connecteur est doté d'une interface commune. Cette caractéristique permet la réutilisation du connecteur dans le cas d'un changement de port du à un changement de composant.

- Chaque rôle du connecteur est doté d'un adaptateur d'interface soit du port vers le connecteur soit du connecteur vers le port.

Le patron de conception adaptateur permet de relier deux classes conçues indépendamment. La technique d'implémentation de l'adaptateur que nous avons suivi consiste à introduire une classe qui implémente l'interface requise. C'est au niveau d'une méthode implémentée de l'interface qu'il y'aura l'appel effectif à la méthode non conforme en terme de nom et signature avec la méthode implémentée.

Dans notre approche de réalisation de connecteurs, nous avons défini une interface du connecteur qui comporte tous les services correspondants au contexte d'action du connecteur. Les noms de méthodes de l'interface des connecteurs sont définis indépendamment des ports interconnectés. L'adaptation se fait au niveau des deux rôles d'un connecteur simple. Au niveau port client, l'adaptation transforme un appel de service requis en un appel d'un service correspondant du connecteur. Au niveau rôle connecté au port de service l'adaptation consiste à transformer une méthode du connecteur en un appel du service effectif de l'interface fournie par le port. La classe associée au rôle coté client implémente l'interface requise et la classe associée au rôle attaché au port de service implémente l'interface du connecteur.

I.5.5.2 La résolution du problème de distance

Lorsque la distance devient moyenne ou étendue, les connecteurs doivent être renforcés par une couche permettant la mise en œuvre d'une technologie d'interconnexion fournie par le système d'exploitation [ISPS07i] . La technologie la plus usuelle repose sur les flux à base de *socket IP*. Parmi ces technologies, nous trouvons les protocoles standards de l'Internet (FTP, HTTP), les RPC, les RMI, le Bus CORBA et les Web services.

L'objectif de la résolution du problème de distance, consiste à déporter un rôle de connecteur de transport vers le port distant. Ainsi après réalisation du connecteur permettant de résoudre le problème de distance, le rôle local se trouve attaché au port distant.

I.5.5.3 Espace de noms pour les connexions distantes

Dans le contexte de connecteur distant, les composants évoluent dans des espaces d'exécution différents. Les références utilisées dans le cours d'une

exécution peuvent différer lors d'une prochaine exécution ou si le composant était éliminé puis reconstruit dynamiquement. De plus lorsque deux composants s'exécutent dans des espaces différents (processus différents), il n'est pas possible pour un composant dans un processus de connaître directement la référence du composant dans un autre processus.

La détermination de la référence des composants dans d'autres espaces d'exécution nécessite l'utilisation d'un mécanisme qui puisse, à partir d'une référence indépendante du déploiement réelle de l'application, nous renseigner sur la référence exacte du composant durant son exécution. La référence indépendante doit appartenir à un espace de nom garantissant son unicité.

La référence indépendante du déploiement que nous utilisons est le nom d'une instance d'un élément de modélisation. Le nom d'instance appartient à un espace de nom hiérarchique représenté par la hiérarchie de composition d'un composant qui a une structure arborescente. C'est cette structure arborescente de la hiérarchie de composition qui nous permet de définir un espace de nom garantissant l'unicité d'un nom d'une instance.

Pour l'identification d'une instance quelconque (point d'accès, port, connecteur, connecteur élémentaire, composant), nous utilisons une technique similaire à l'identification de machines et programmes dans le réseau Internet. Le nom d'une application (qui est aussi un nom de type car une application même complexe peut être vue comme un type de composant instanciable), représente la racine d'un espace de nom pour toutes les instances de composants, port, connecteurs et point d'accès. L'espace de nom d'un point d'accès est l'instance de port, celui du port est l'instance de composant. Le domaine de nom d'une instance de composant est l'instance du composite.

Lors de l'exécution de plusieurs instances d'une même application, il est nécessaire d'identifier chaque instance. Nous utilisons comme première solution le numéro de processus pour distinguer les instances dans un même environnement d'exécution. La distinction entre instances sur des machines différente est réalisée par l'ajout du nom de domaine Internet au nom d'une instance de l'application.

Cette technique de construction de nom d'instance garantit l'unicité même dans le contexte d'un système fortement distribué qu'il soit dans le contexte du réseau Internet ou au niveau d'un réseau intranet.

I.5.5.4 Les connecteurs de services

Les connecteurs de service sont des connecteurs prédéfinis offrant les services élémentaires que nous pouvons trouver dans une architecture et même dans les réseaux informatique et architecture de processeurs et ordinateurs.

Dans l'approche actuelle, du fait de leur complexité et les problèmes qu'ils doivent prendre en charge, les connecteurs de service sont conçus et réalisés directement dans un langage de programmation [ISPS07i] . Nous considérons dans l'approche IASA que la réalisation de connecteurs de services n'est pas une activité d'un architecte d'application mais d'un architecte de connecteurs. L'architecte d'application peut mettre en place des connecteurs complexes par composition de connecteurs de transport et de connecteurs de services. Dans ce qui suit nous présentons quelques connecteurs de services usuels.

I.5.5.4.1 Les répéteurs 1 vers n (ou distributeurs 1 vers n)

Un répéteur (Figure 19) permet de réaliser une connexion d'un port client vers plusieurs ports de services, présentant les mêmes interfaces mais pouvant réaliser des opérations complètement différentes.

Ce type de composant est utilisé pour

- Lancer selon une politique prédéfinie (séquence ou parallèle) les services présents sur les ports de plusieurs composants (i.e. plusieurs instances d'un même composant). Les services offerts par les divers composants connectés au répéteur doivent présenter la même interface et pourraient implémenter des logiques totalement différentes.
- Envoyer des données à plusieurs destinations.

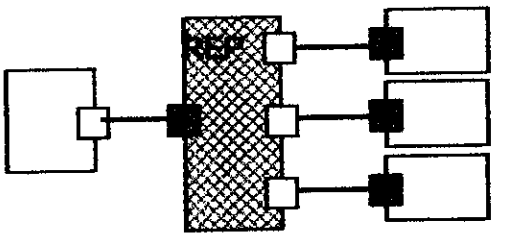

Figure 19: Schéma du répéteur

La politique de lancement peut être contrainte par le plan de déploiement des composants et du connecteur. Ainsi si le connecteur se trouve dans la même tache que les services à lancer, il ne peut pas appliquer à ces services une politique de lancement parallèle. Le port d'entrée du composant et les ports de sortie sont totalement compatibles. Le port d'entrée est soit un *DataPort* ou un ServerPort et les ports de sorties sont tous soit des *DataPort* si le port d'entrée est un *DataPort*, soit tous des *ClientPort* si le port d'entrée est un *ServerPort*.

Les lancements de services représentent soit une logique ordinaire dans laquelle un service est lancé et le client attend d'éventuelles réponses, soit une logique de distribution d'événements. Dans ce dernier cas, une connexion signifie un intérêt pour l'événement.

Le cascadage de répéteur permet d'obtenir des logiques de lancement de services très complexes (Figure 20.)

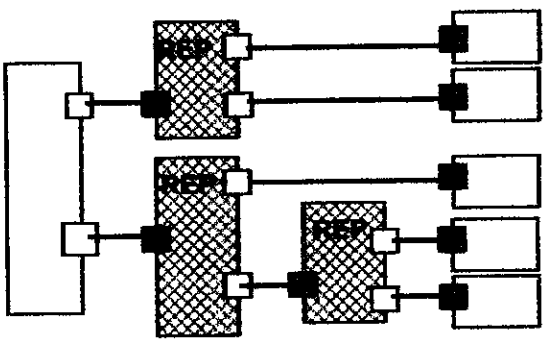

Figure20: Cascadage de répéteurs

Le répéteur n vers n peuvent être représenté par un le connecteur de diffusion ou le commutateur que nous présentons par la suite.

I.5.5.4.2 Les concentrateurs

Un concentrateur (Figure 21) permet à plusieurs clients un accès à un même service. Le composant est doté de plusieurs ports d'entrée de type *ServerPort* et d'un seul port de sortie *ClientPort*. Le concentrateur réalise principalement l'opération de synchronisation des accès au service requis par plusieurs clients. Un connecteur d'équilibrage de charge peut être cascadé à la sortie de ce connecteur (Figure 21.b).La synchronisation de l'accès au service est gérée par ce composant et dépend du plan de déploiement des composants attachés au connecteur.

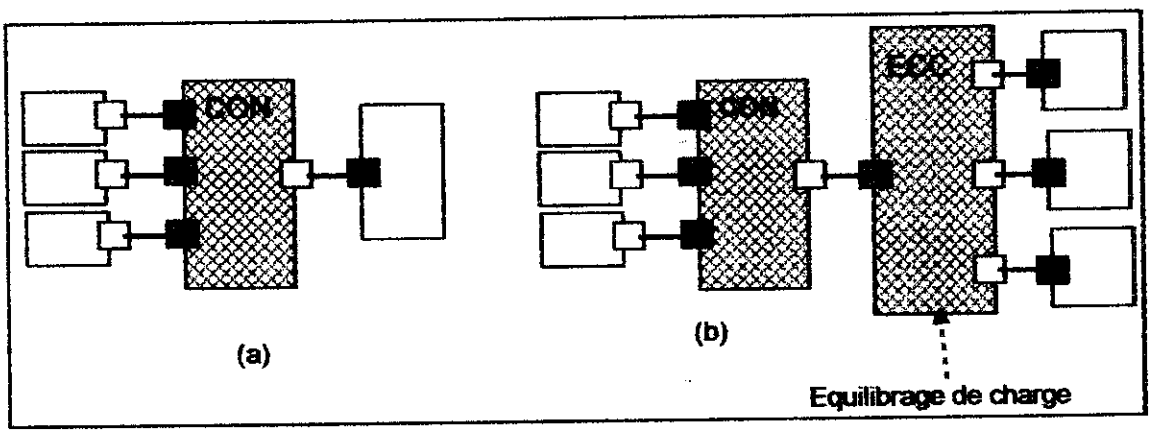

Figure 21: Concentrateur (a) simple, (b) avec équilibrage de charge

I.5.5.4.3 Les observateurs d'interaction

Ce sont des connecteurs dédiés au contrôle dynamique des interactions. Ils reportent les traces d'une interaction et génèrent des exceptions dans le cas ou l'interaction ne se fait pas réellement selon ses spécifications. Ils s'interposent entre un port client et un port serveur (Figure 22)

Un observateur est doté de 3 ports de contrôles : un port de contrôle global du service, un port d'exception et un port de données dédié à la production de trace. Le port de contrôle permet d'activer ou non le connecteur. Lorsque le connecteur est inactif, il devient transparent et le port de service et client entre lesquels le connecteur s'interpose seront directement reliés. En activité une action ne peut aller directement vers sa destination. C'est le connecteur qui la fera parvenir dans le cas ou l'interaction respecte les spécifications.

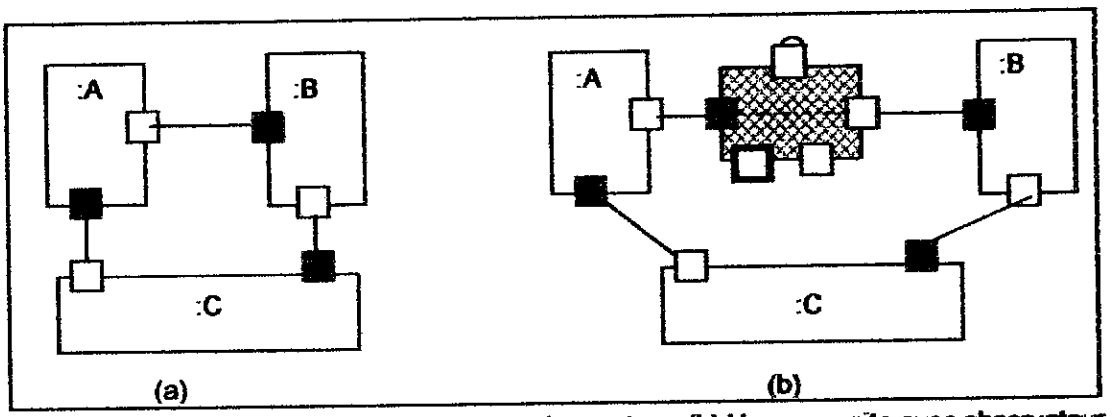

Figure 22: L'observateur : (a) Un composite sans observateur. (b) Un composite avec observateur.

I.5.5.5 Complexité des composants de services

Pour illustrer la complexité des connecteurs de service, et la nécessité de les traiter dans une discipline à part que nous pourrons appeler architecture de connecteurs, nous présentons deux composants qui pourraient avoir une utilisation très intéressante dans la réalisation d'applications fortement distribuées.

Il est possible que des similitudes puissent exister entre certain connecteurs de services et d'autres solutions software, notamment dans les systèmes distribués (i.e. similitude entre tableau noir et diffuseur). Cependant dans notre approche nous nous intéressons plus à l'aspect connectique et la nécessité d'utiliser ces connecteurs dans une approche de mise en place de connecteurs complexes par cascadage.

I.5.5.5.1 Les diffuseurs

Un diffuseur permet de spécifier de manière explicite le concept de partage de mémoire. Les rôles d'un diffuseur sont tous de type *InOutDataRole*. L'accès à tous les rôles (en émission et en réception) du diffuseur est synchronisé par le diffuseur lui même. La réalisation du diffuseur peu être très complexe, car dans notre approche il ne s'agit pas réellement de données partagée se trouvant dans un seul endroit. Une copie de la donnée partagée est en fait maintenue au niveau de tous les ports *inout* interconnectés. Ces données se trouvent réellement positionné au niveau de leurs composants et les ports connectés doivent à tout moment présenter une même valeur. L'approche IASA consiste à doter le connecteur de rôle offrant une interface dont les méthodes, une fois implémentées, doivent impérativement faire appel à une

méthode synchronisée, avant de réaliser effectivement la tâche de transfert ou d'acquisition de donnée. La méthode synchronisée permet de gérer l'accès au diffuseur et de le mettre à la disposition d'un seul port. De cette manière nous garantissant que le diffuseur accède à tous les ports pour mettre à jour la valeur des ports de données *inout*.

I.5.5.5.2 Les commutateurs

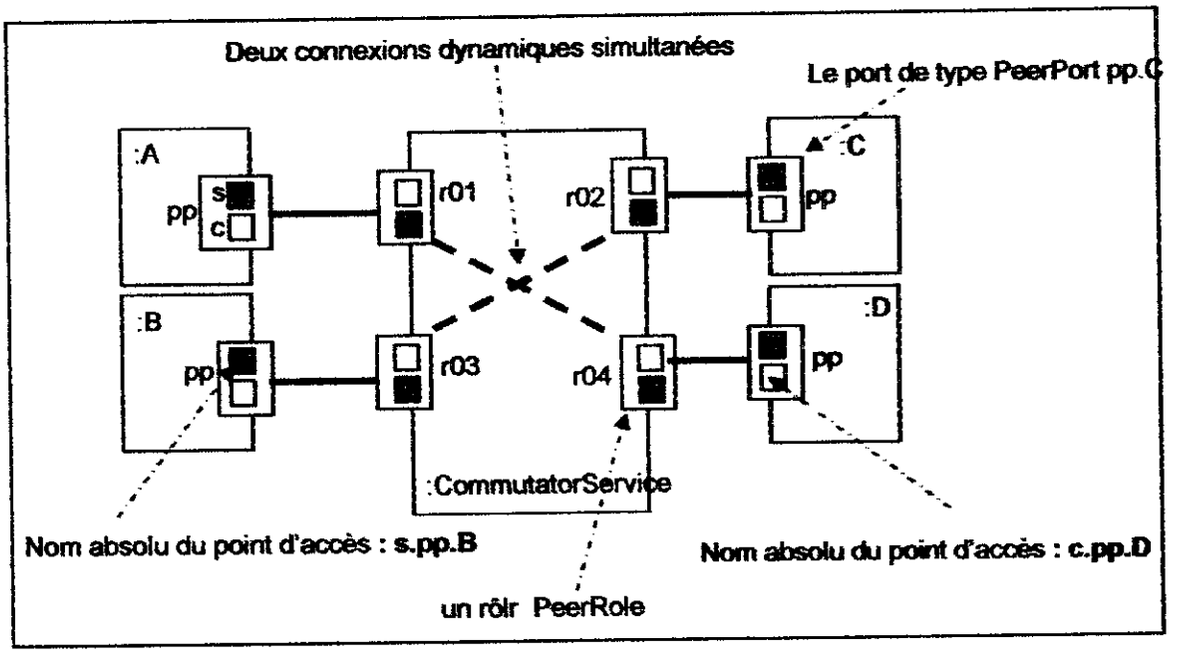

Figure 23: Connecteur de commutation de ports

Les extrémités (rôles) d'un commutateur sont tous de type *PeerRole* comme le montre la figure 23 Dans un commutateur, les connexions sont créées et supprimées dynamiquement. Les connexions à l'intérieur du commutateur sont représentées par des connecteurs directs. L'objectif du commutateur est de permettre dans le contexte d'un système distribué à des composants de communiquer simultanément deux par deux. Si un composant est en communication (son port est occupé), il ne peut être sollicité pour une communication avec un autre composant. Lorsque la communication se termine (ce dernier aspect peut être représenté par la fin de l'appel d'une procédure en mode synchrone) les composants qui étaient en communication peuvent alors être sollicités par d'autres composants. Chaque rôle du commutateur, associé à un point de service (i.e. point **s.pp.A, s.pp.B**), est une

ressource critique dont l'accès doit être synchronisé. L'exemple suivant illustre le fonctionnement du commutateur.

Lorsque le composant **A** (figure 23) veut se connecter au composant **B**, le *PeerPort* du composant **A** active le rôle correspondant sur le commutateur (c'est le point d'accès ***c.pp.A*** qui est utilisé). Ce dernier tente alors d'accéder au rôle de service attaché au point de service du port du composant **B**, c'est-à-dire au point d'accès **s.pp.B** Si le point **s.pp.B** est en activité le système d'accès synchronisé du commutateur ne permettra pas l'accès à ce point que lorsque il ne sera plus occupé. Le point d'accès **c.pp.A** sera mis en attente de la libération du point d'accès **s.pp.B**. Si le point d'accès **s.pp.B** est libre, une connexion est alors établie. Si le port **pp.A** effectue une connexion asynchrone, le port est libéré et la requête de service est mise en file d'attente du rôle destinataire.

I.6 CONCLUSION

Nous venons de voir les différents aspects de modèle IASA qui (comme la plupart des modèles composant) est une approche naturelle plus proche du modèle mentale. Cette approche bien qu'elle soit conceptuellement ancienne et directe n'a commencé à être considéré de manière plus sérieuse et plus rigoureuse qu'après le développement extraordinaire des systèmes informatique et la prolifération des réseaux informatique.

CHAPITRE II

Analyse Et Conception

CHAPITRE II : ANALYSE ET CONCEPTION

II.1 Introduction

Nous commençons ce chapitre par un rappel du problématique ainsi que les objectifs qu'il faut les réalisés. Par la suite nous passons à la présentation de démarche de développement de l'IDE.

Pour réaliser cet outil nous avons procède comme suit:

Dans une première étape nous avons commencé par limiter les fonctionnalités de l'IDE que nous devons implémenter. Autrement dit, nous avons essaye de répondre a ces deux questions: À qui va servir cet IDE? Et Que doit- il faire?

Ensuite nous avons procédé à la construction du modèle objet en utilisant certain diagrammes de la méthode UML.

II.2 Rappel du Problématique et Objectifs

Il est question dans ce sujet de réaliser dans le contexte de la plateforme ECLIPSE, un IDE destiné à la conception de logiciel selon une approche basée sur les concepts de composant et de connecteur. Le modèle de composant et connecteurs utilisé est plus fin que celui d'UML2.0. Le modèle UML2.0 ne permet pas de considérer les éléments d'interface de manière indépendante. Une interface UML2.0 est considérée dans sa totalité dans toute opération de connexion de ports de composant. Le modèle UML2.0 ne permet pas une grande flexibilité dans la spécification d'architecture logicielle et ne peut pas prendre en charges la spécification informelle d'architecture logicielle.

Le modèle sur lequel se basera l'IDE à développer reconnaît les notions de ports et interfaces d'UML2.0 et en plus permet de manipuler les éléments constitutifs de chaque interface que nous appelons : point de connexion. Ces derniers sont de deux grandes catégories : les points véhiculant des actions et les points véhiculant des données.

Les connecteurs peuvent aussi être considérés de manière plus fine que dans UML. C'est ainsi que dans le modèle sur lequel se basera l'IDE il y' a la notion de connecteurs élémentaire et la notion de connecteurs. De plus, les connexions se feront soit par le tracé direct de connecteurs entre ports soit en utilisant des composant de communication.

L'IDE à réaliser permettra de dessiner une architecture logicielle (spécification de composant et de connecteurs entre ports de composant). L'IDE devra permettre un tracé libre et très flexible de connexion, très similaire à la spécification informelle d'architecture logicielle, donc très proche du modèle mental que se fait un architecte de son logiciel..

L'IDE doit permettre l'enrichissement de la spécification structurelle (composant et connecteur) par la spécification d'un timing associée à l'architecture dessinée et d'un plan de déploiement des composants. Ce sont ces deux aspects qui donneront une rigueur à la spécification informelle. C'est à partir du timing et du plan de déploiement qu'il deviendra par la suite possible de construire à partir d'une spécification informelle une spécification rigoureuse appelée spécification standardisée. La spécification standardisée représente le point de départ pour la génération de code associée à l'architecture indiquée

Après la spécification de l'architecture dans des styles sans grandes contraintes, l'IDE devra en finalité génère un code qui représentera l'architecture dessinée.

II.3 La spécification des besoins

La spécification des besoins est une étape essentielle au début du processus de développement. Son but est d'éviter de développer un logiciel non adéquat [Sommerville, 88].

La finalité de cette étape est la description générale des fonctionnalités du système. Par la réponse à ces questions : "quelles sont les fonctions du système ?", "quels sont les utilisateurs du système?", "et qu'attendent-ils du système ?". Cette étape étudiée le comportement du système exprimé sous la forme des cas d'utilisation, le contexte du système, les acteurs et les scénarios.

II.3.1 Les cas d'utilisations

L'analyse détermine le quoi faire, c'est à dire les besoins de l'utilisateur. L'expérience montre que la technique des cas d'utilisation (use cases) se prête bien à la détermination des besoins d'utilisateurs [Muller, 97]. Les cas d'utilisation décrivent sous la forme d'actions et de réactions le comportement du système du point de vue de l'utilisateur.

L'étude des cas d'utilisation débute par la détermination des acteurs (catégories d'utilisateurs) du système.

II.3.1.1 Définition des acteurs

Les acteurs sont la réponse à la question : pour qui est réalisé l'IDE ? Notre IDE est destiné à un utilisateur qui souhaite développer des applications en utilisant l'approche d'assemblage des composants. Dans la plus part des cas c'est un *développeur*. Mais il se peut qu'un utilisateur qui n'a que des notions de base en informatique utilise cette IDE.

II.3.1.2 Description textuelle des cas d'utilisations

Nous avons choisi de présenter les cas d'utilisations de la manière suivante : d'abord nous résumons les cas d'utilisations dans des cas plus généraux, puis nous détaillons chaque cas d'utilisations dans un diagramme indépendant.

Les cas d'utilisations les plus généraux sont :

- la gestion des projets : qui comporte la création, la suppression, l'ouverture, l'importation, l'exportation, la modification et la sauvegarde d'un projet. On peut détailler la création d'un projet comme suit :

  - Saisie de nom de projet.
  - Vérification de la validité du nom de projet : Comme le nom d'un projet sera le nom du package (dans la génération du code), certain conditions s'imposent dans le nommage de projet (les conditions de nommage des packages dans JAVA).
  - Vérification de non-existence de projet : On devrait vérifier aussi qu'il n'existe pas un autre projet du même nom choisi.

- La création de répertoire contenant le projet : Chaque projet a un répertoire de son nom qui va contenir les répertoires et les fichiers de ses composants.

La suppression du projet ne contient qu'une confirmation de suppression et la suppression de répertoire, des sous répertoires et des fichiers de projet.

L'ouverture d'un projet consiste à lire les données de projet, et les représenter graphiquement sous forme d'architecture.

L'importation d'un projet est en deux étapes :

- Recopier le répertoire de projet et tous ce qu'il contient dans le répertoire de travail.
- L'ouverture de projet recopié.

L'exportation d'un projet consiste à le sauvegarder totalement ou partiellement sous forme d'un composant composite dans un emplacement de disque choisie par l'utilisateur. Donc on peut la diviser comme suit :

- La sélection des composants qu'on veut exporter.
- Explorer le disque afin de trouver le chemin où on va exporter le composant.
- Sauvegarder les fichiers du composant composite.

La modification d'un projet <*extend*> tous les autres cas d'utilisations, c'est à dire l'utilisation de n'importe quelle cas d'utilisation se compte comme une modification de projet.

La sauvegarde du projet a deux options :

- Une sauvegarde simple : C'est sauvegarder les modifications survenu sur le projet dans le même répertoire du projet.
- Une sauvegarde sous : C'est sauvegarder tout le projet sous un autre nom dans un autre répertoire.

➢ La gestion des composants : qui comporte la création, la suppression, l'importation, l'exportation et la modification d'un composant, et aussi les différentes opérations graphiques liées a l'affichage du composant dans le dessin d'architecture du projet. La gestion d'un composant contient aussi la gestion des

ports et la gestion des composants ainsi que la gestion des connecteurs dans le cas des composants composites.

On peut détailler la création d'un composant comme suit :

- Saisie de nom de composant.
- Choisie de mode de composant : Simple ou composite.
- Choisie le type de composant : usuel, visuel, contrôle....
- Vérification de la validité du nom de composant.
- Vérification de non-existence de composant dans le projet.
- Positionner le composant : trouver une position adéquate pour le composant dans l'architecture.
- Dessiner le composant dans l'architecture.
- La création de répertoire contenant les fichiers associé au composant.

La suppression du composant contient :

- Une confirmation de suppression.
- Choisie le type de suppression : il s'agit de choisir si la suppression va affecter le composant ou seulement l'instance (dans le cas ou il y aura une seule instance de composant dans l'architecture). Dans le cas de la suppression d'instance, on ne va que supprimer les données de composant relatives à l'architecture (c'est-à-dire juste ou an a utilisé le composant dans la composition), et on efface le dessin de ce composant de l'architecture. Mais dans le cas ou on va supprimer le composant, en plus de la suppression des données relative à l'architecture et l'effacement du dessin, on doit procéder à la suppression de répertoire et des fichiers associés au composant ainsi que l'effacement du dessin de composant de l'espace de dessin d'architecture de projet.

L'importation d'un composant dans un projet est en deux étapes :

- La copie du répertoire associe au composant ainsi que les fichiers qu'il contient dans le répertoire de projet.
- Lire les données de composant afin de l'afficher dans le dessin de l'architecture de projet.

L'exportation d'un composant est de sauvegarder le composant totalement ou partiellement (dans le cas d'un composant composite) dans un autre répertoire que le répertoire de projet.

Tous comme la modification d'un projet, la modification d'un composant est représenter par l'utilisation d'un ou plusieurs cas d'utilisation qu'il *<extend>* (nous présentons ces cas par la suite). La modification d'un composant concerne aussi la modification de son nom, son mode et de son type.

Les opérations graphiques liées à l'affichage d'un composant sont : positionner, redimensionner et déplacer un composant.

- La gestion des ports : Ce cas se divise en : ajouter un port a un composant, supprimer un port d'un composant et modifier un port d'un composant.

Ajouter un port a un composant consiste a :

- Saisie le non de port.
- Vérifie la validité de ce nom.
- Vérifie la non-existence d'un port du même nom dans le composant dont on va ajouter le port.
- Choisie le type de port : selon le type de port, on va choisie le nombre ainsi le type des point d'accès qu'on va ajouter a ce port.
- Choisie l'orientation de port : c'est l'emplacement de ce port dans le composant (Nord, Sud, Est, Ouest).
- Saisie le nombre ainsi que la direction des point data qu'on va ajouter au port : ce cas est optionnel, car en plus des points d'accès prédéfinie selon le type de port, on peut ajouter des point d'accès data selon le besoin.

La suppression du port d'un composant consiste a :

- Confirmer la suppression.
- Supprimer les données du port contenu dans la définition de composant.

- Effacer le dessin du port dans le composant.
- Supprimer les connecteurs (s'ils existent) vers ou partant de ce port.

La modification d'un port est l'utilisation de n'importe quel cas d'utilisation contenu dans la gestion des points d'accès, en plus la modification de son nom et la modification de son orientation.

➢ La gestion des points d'accès : c'est résumé dans l'ajout d'un point d'accès à un port, la suppression et la modification d'un point d'accès.

L'ajout d'un point d'accès à un port est :

- Saisie le nom de point d'accès.
- Vérifie la validité du nom saisie.
- Choisie le type de point d'accès : Service ou Data.
- Choisie la direction de point d'accès : Dans le cas d'un point Service c'est Server ou Client, et dans le cas d'un point Data c'est IN, OUT ou INOUT.

La suppression d'un point d'accès est similaire à celle du port. Donc elle est constituée d'une confirmation, suppression des données, effacement de dessin, et la suppression des connecteurs.

La modification d'un point d'accès concerne la modification de son type et de son direction.

➢ La gestion des connecteurs : la gestion des connecteurs ne contient que deux cas : l'ajout et la suppression d'un connecteur.

L'ajout d'un connecteur se décompose comme suit :

- La sélection des ports (ou les points d'accès) qu'on veut connecter.
- Vérifier la validité de connexion : Vérifier si on peut connecter ces deux ports (ou points) ou non.
- Déterminer le type de connecteur selon les ports (les points) qu'on va connecter.
- Déterminer le chemin (dans le dessin) que le connecteur va suit.
- Tracer le connecteur dans le dessin de l'architecture.

- Ajouter les données de connecteur a celles de projet.

La suppression d'un connecteur est en 3 étapes :

- Confirmation de suppression.
- Effacement de dessin de connecteur.
- Suppression de données de connecteur.

II.3.1.3 Diagrammes des cas d'utilisations

Comme la description textuelle des cas d'utilisation, nous allons faire un diagramme des cas d'utilisations les plus généraux, puis nous les détaillons dans des sous diagrammes.

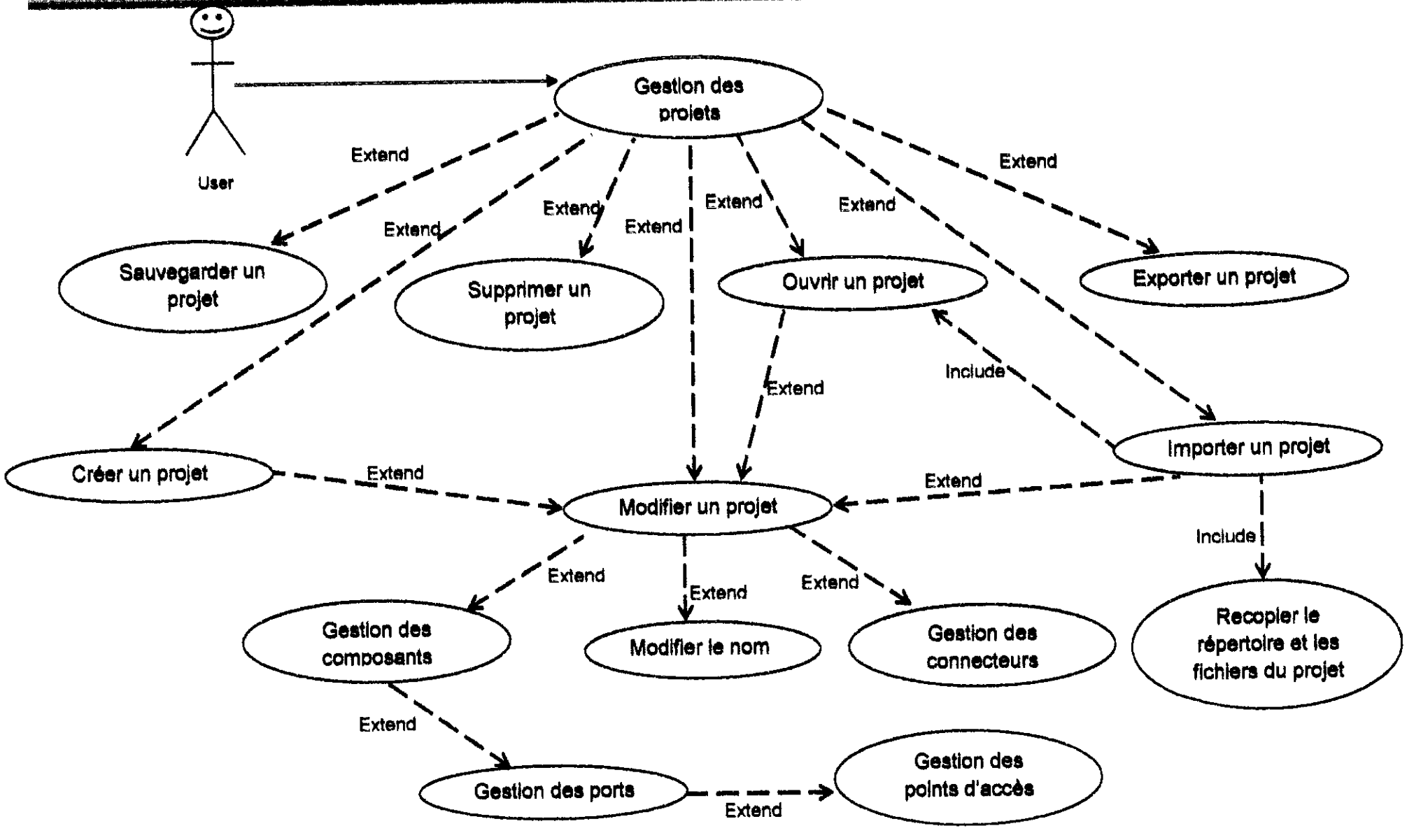

Figure 24 : Diagramme générale des cas d'utilisation

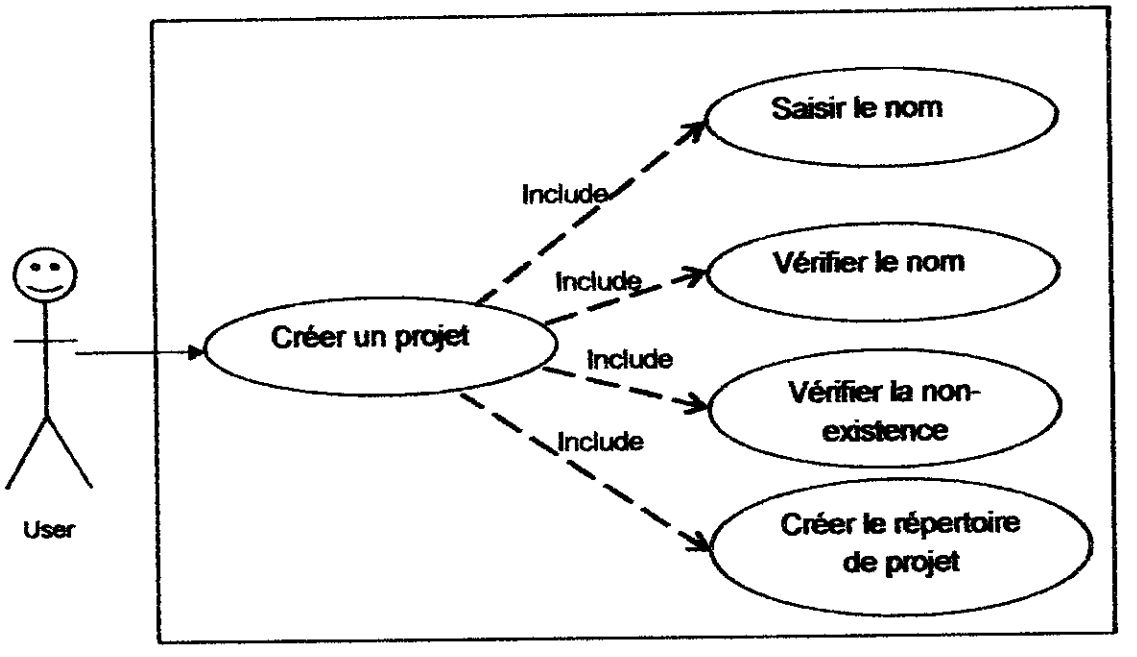

Figure 25 : Diagramme de cas d'utilisation : Créer un projet

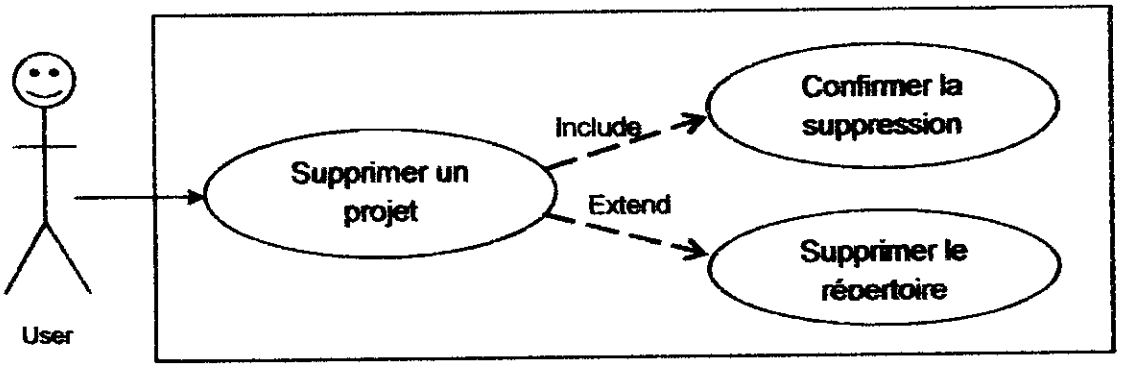

Figure 26 : Diagramme de cas d'utilisation : supprimer un projet

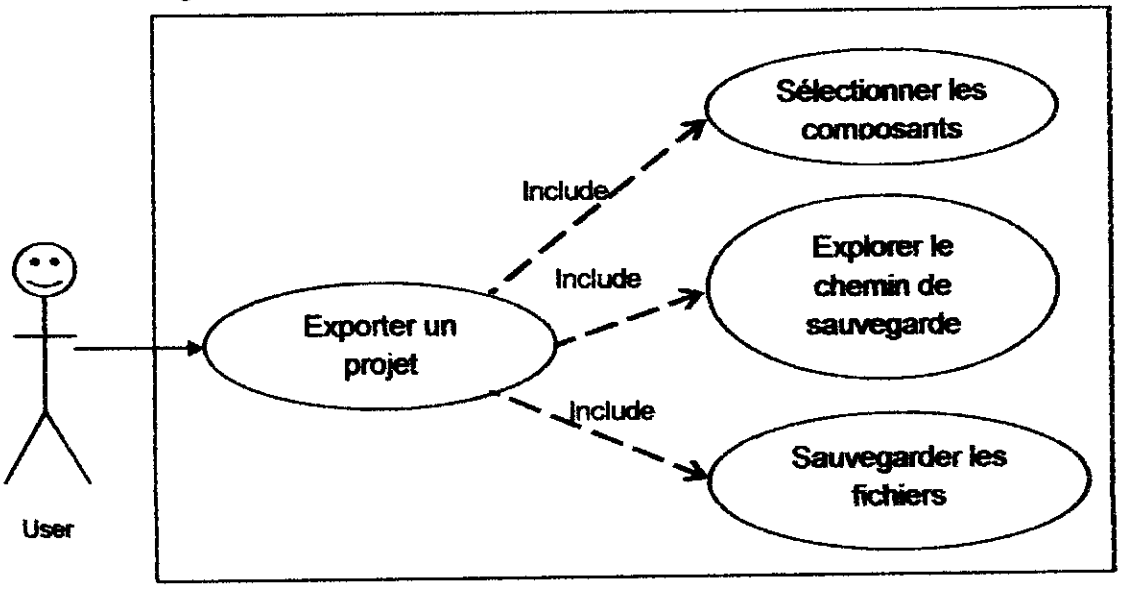

Figure 27: Diagramme de cas d'utilisation : exporter un projet

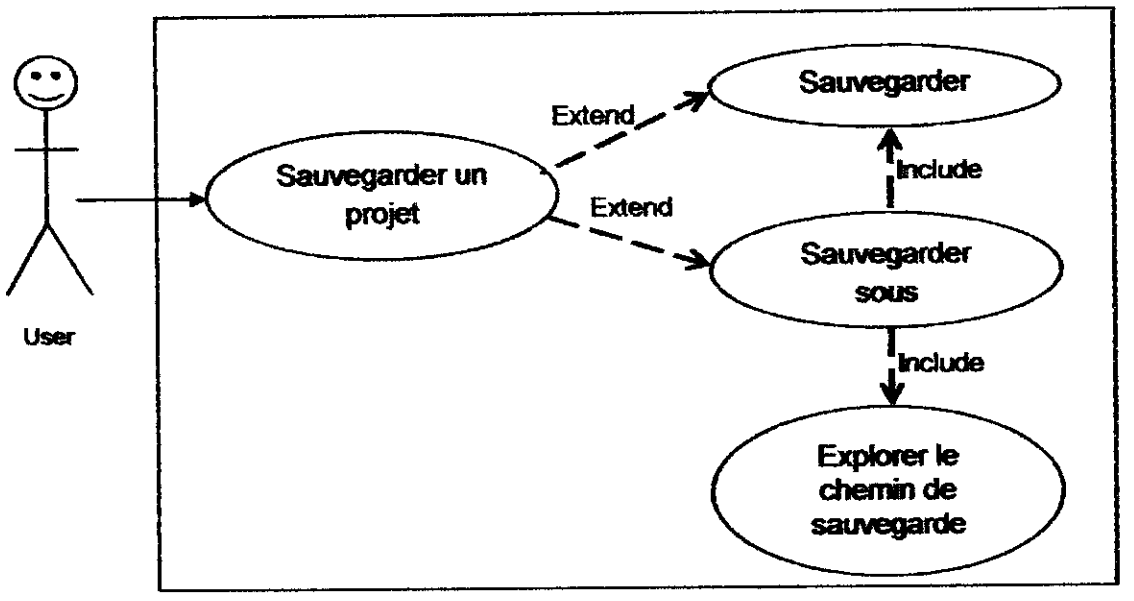

Figure 28: Diagramme de cas d'utilisation : sauvegarder un projet

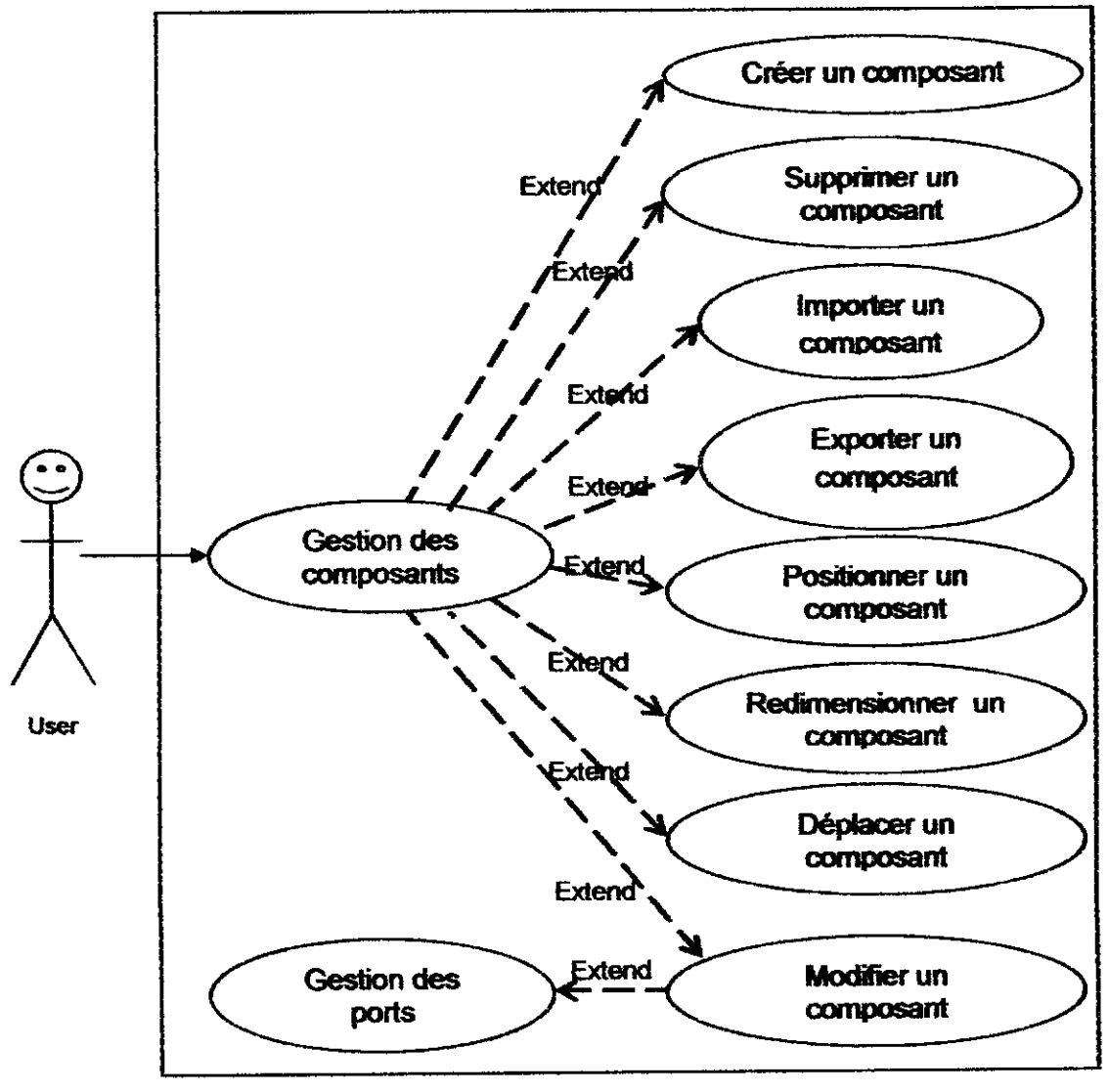

Figure 29: Diagramme de cas d'utilisation : Gestion des composants

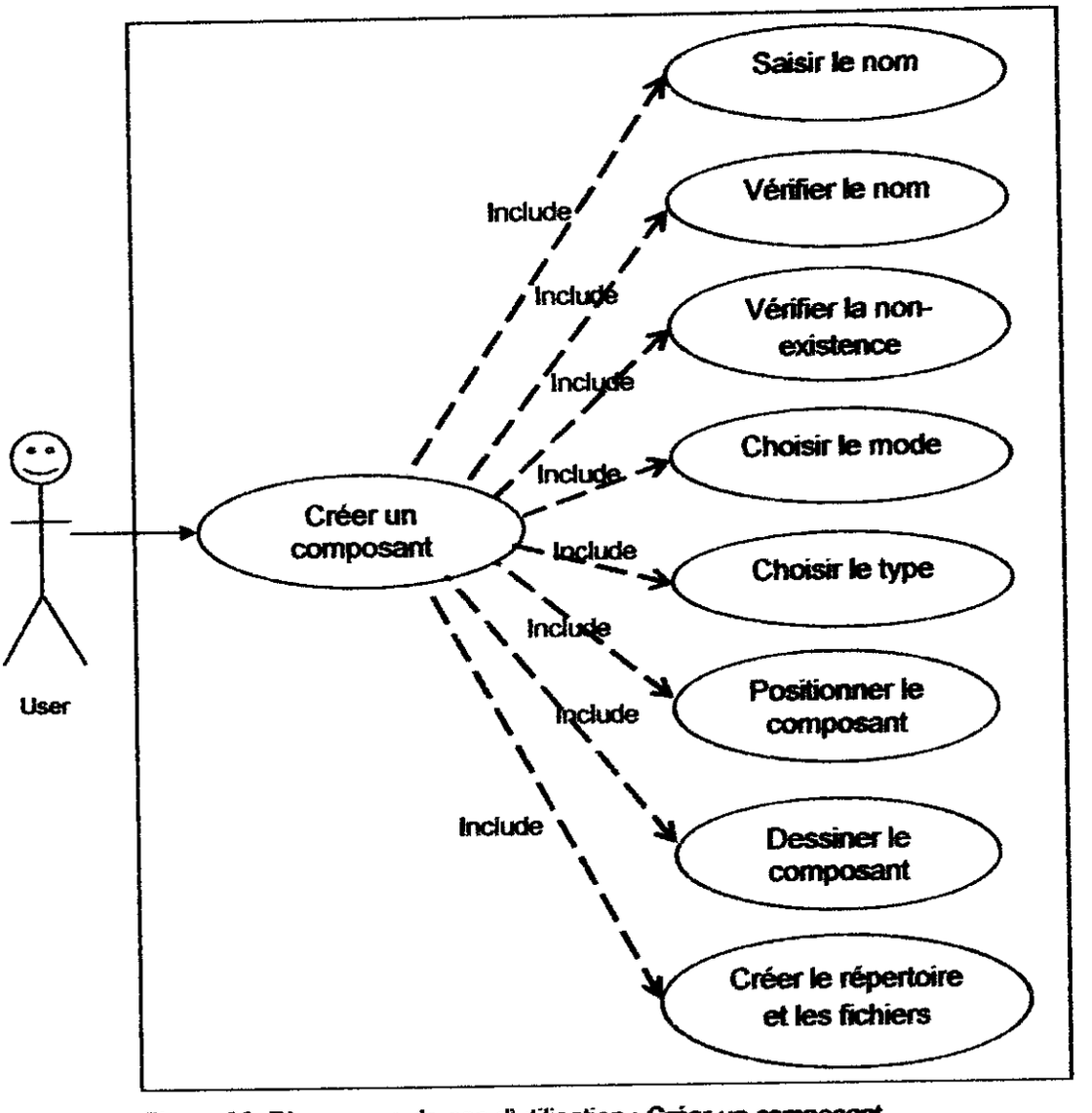

Figure 30: Diagramme de cas d'utilisation : Créer un composant

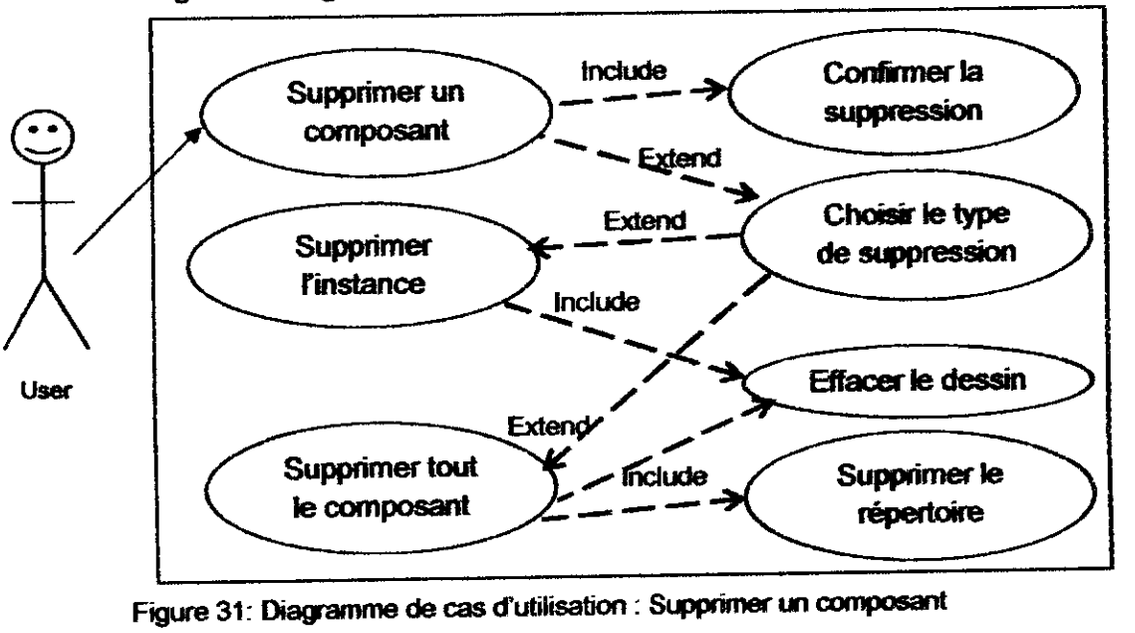

Figure 31: Diagramme de cas d'utilisation : Supprimer un composant

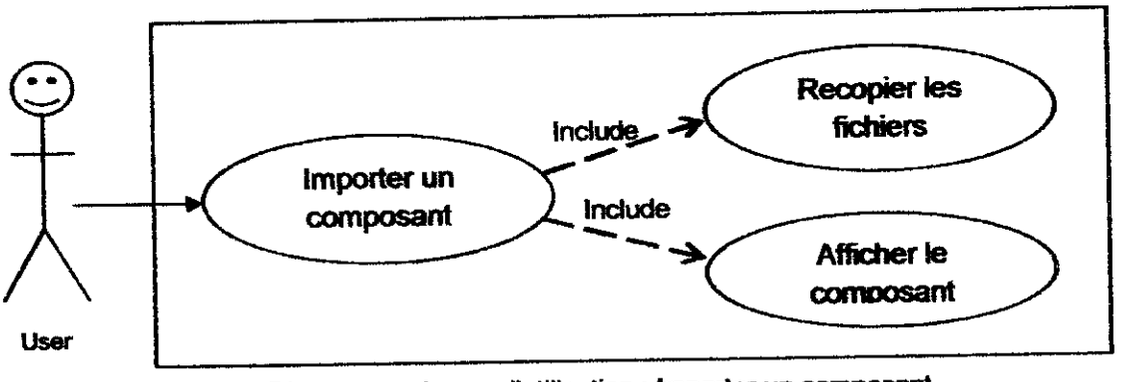

Figure 32: Diagramme de cas d'utilisation : Importer un composant

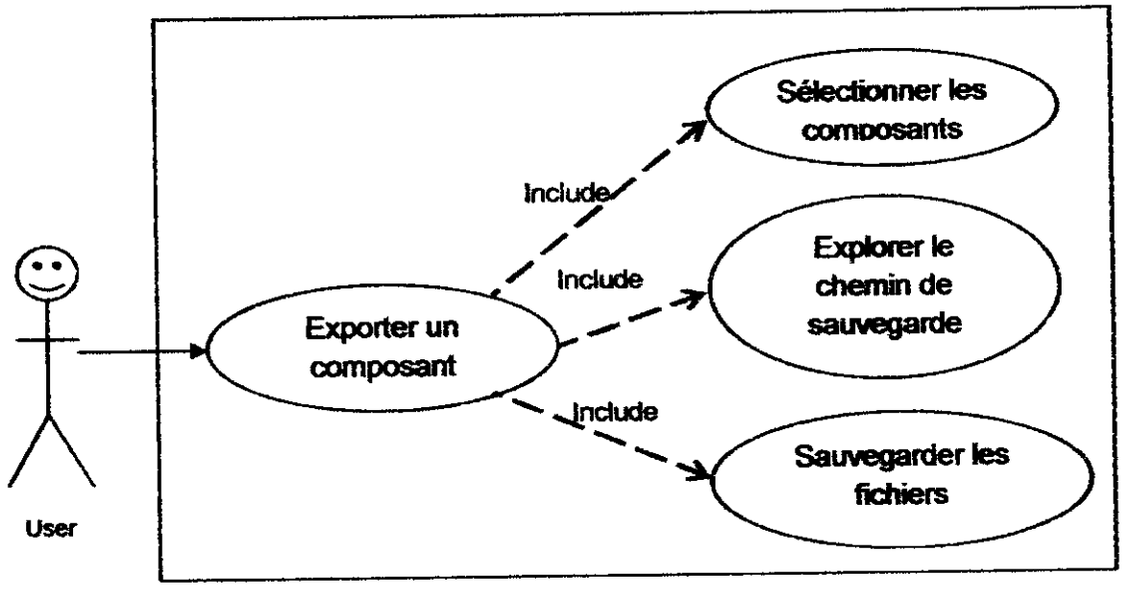

Figure 33: Diagramme de cas d'utilisation : Exporter un composant

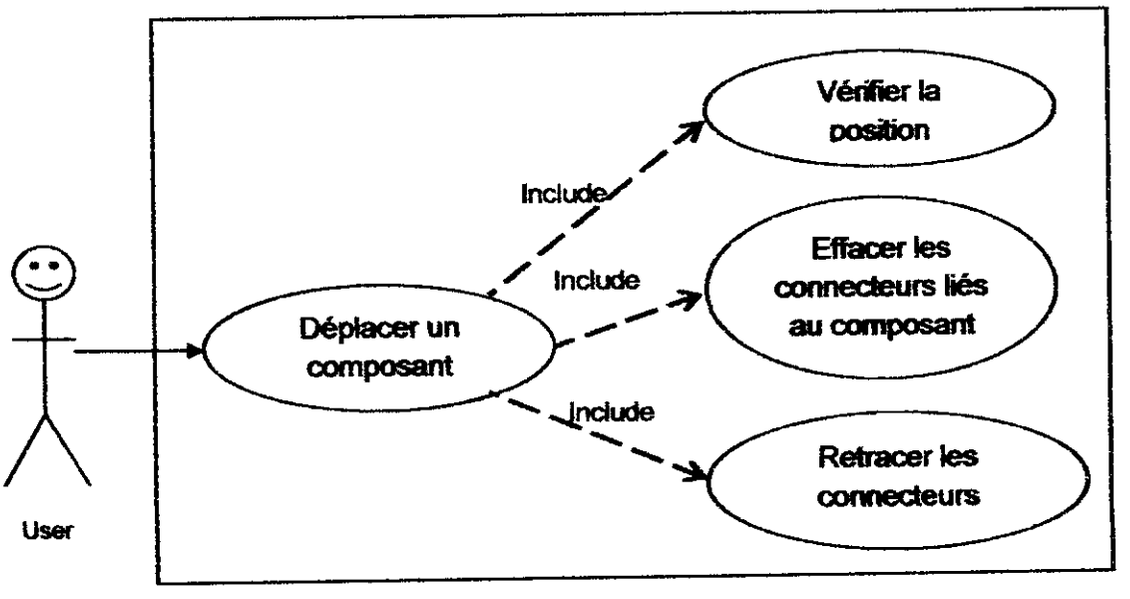

Figure 34: Diagramme de cas d'utilisation : Déplacer un composant

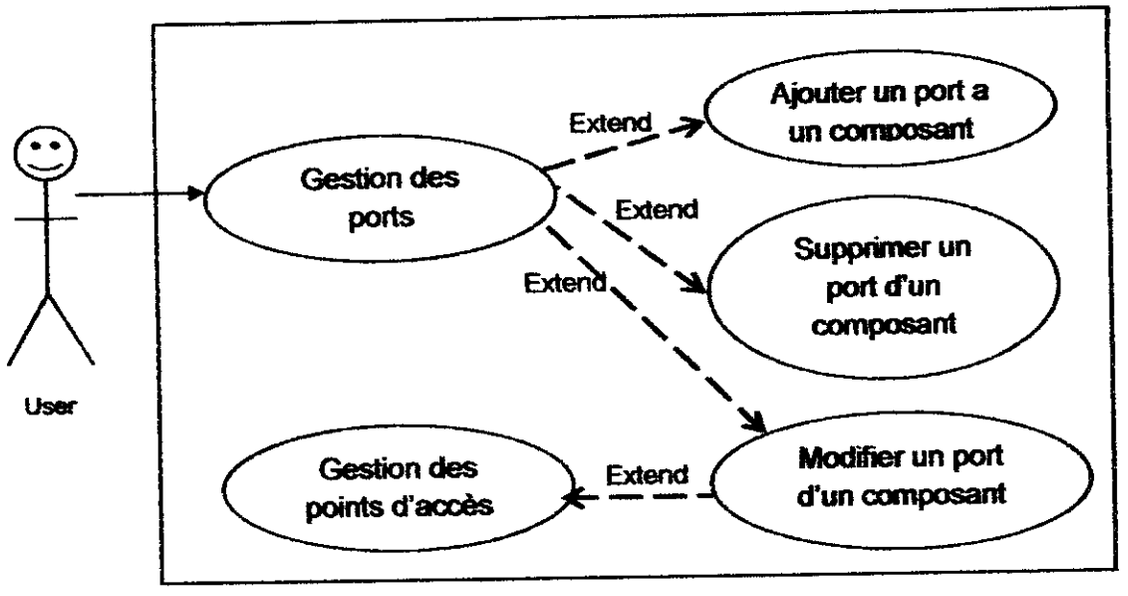

Figure 35: Diagramme de cas d'utilisation : Gestion des ports

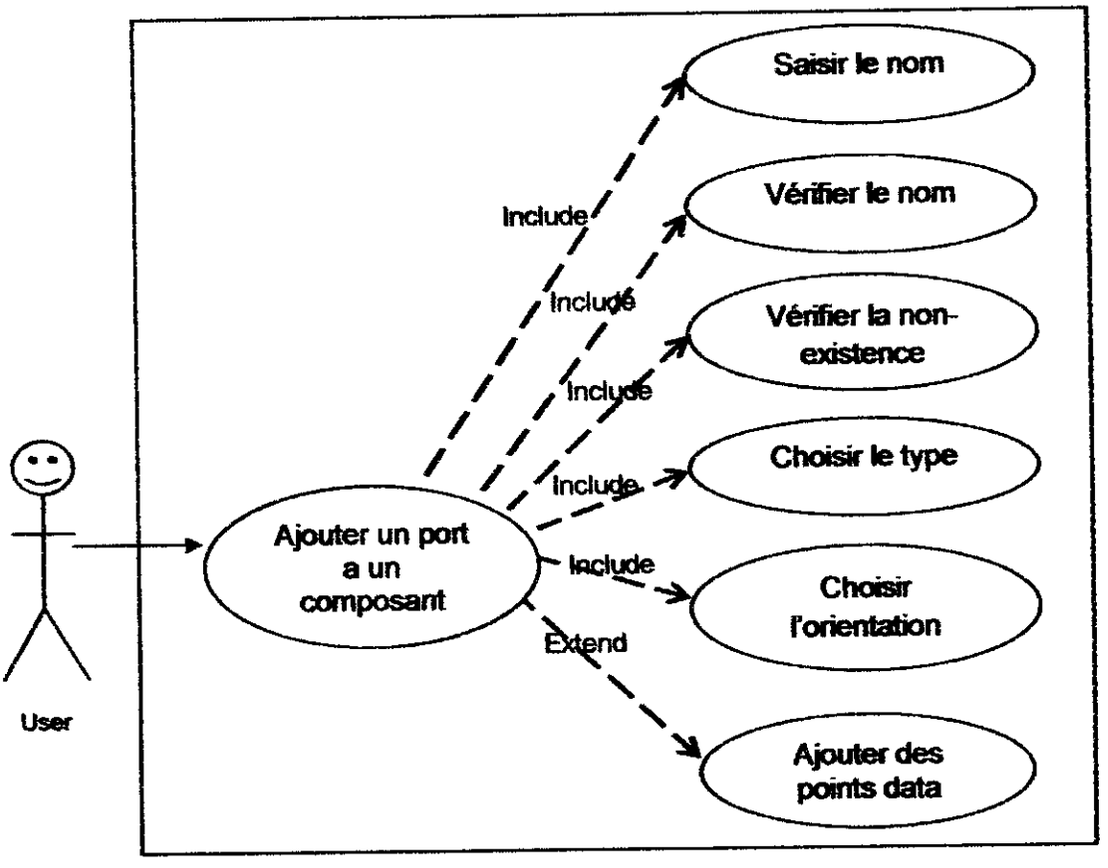

Figure 36: Diagramme de cas d'utilisation : Ajouter un port a un composant

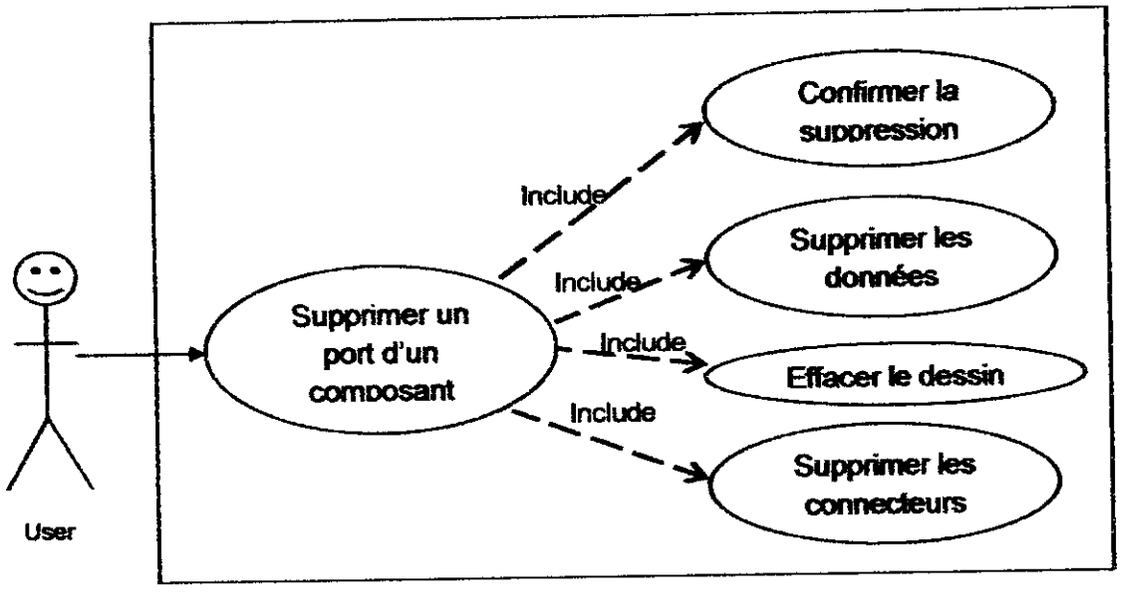

Figure 37: Diagramme de cas d'utilisation : Supprimer un port d'un composant

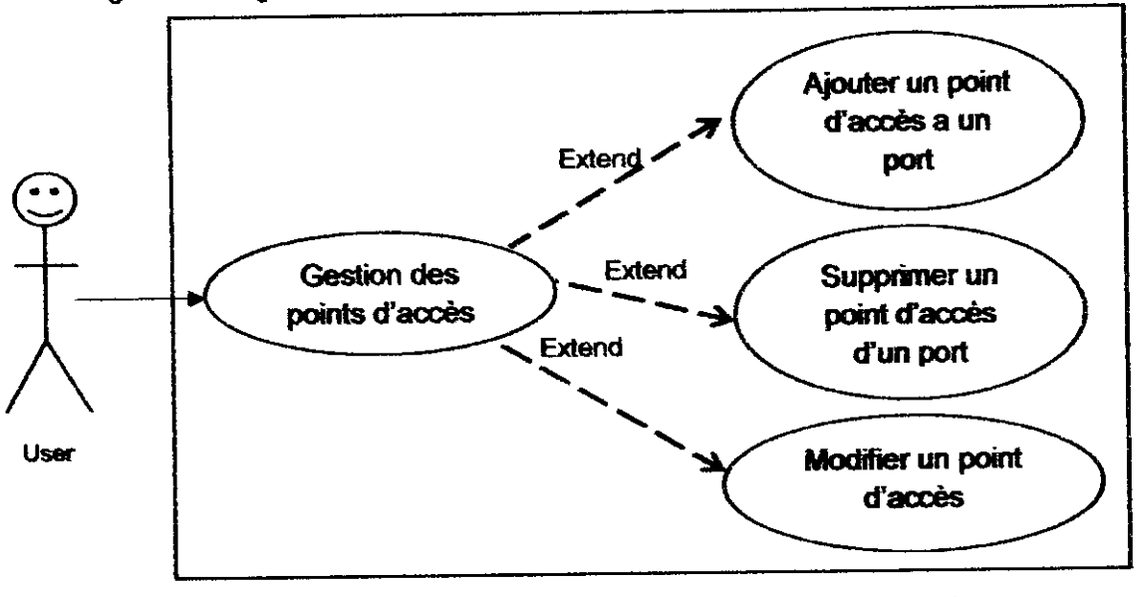

Figure 38: Diagramme de cas d'utilisation : Gestion des points d'accès

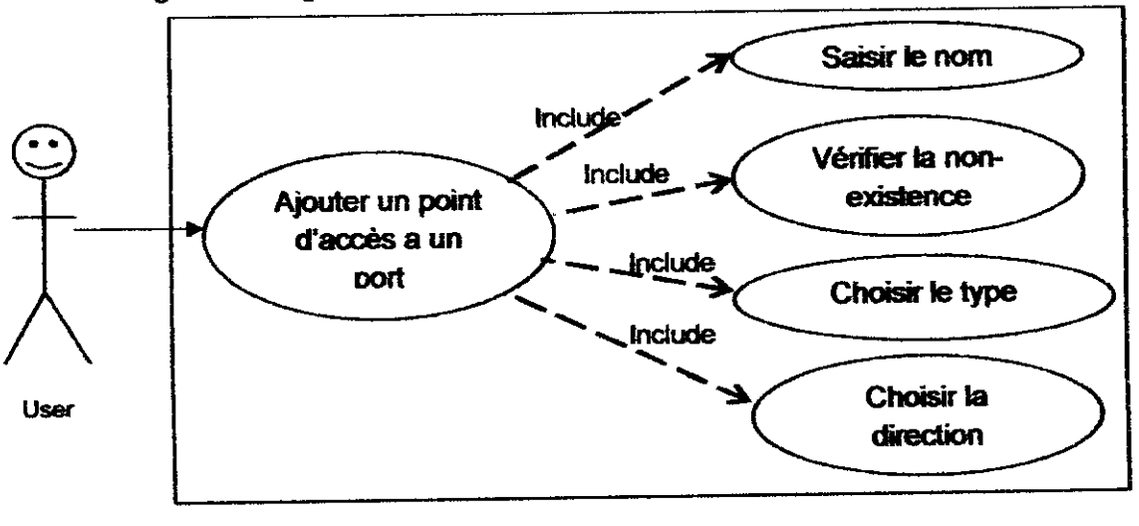

Figure 39: Diagramme de cas d'utilisation : Ajout d'un point d'accès a un port

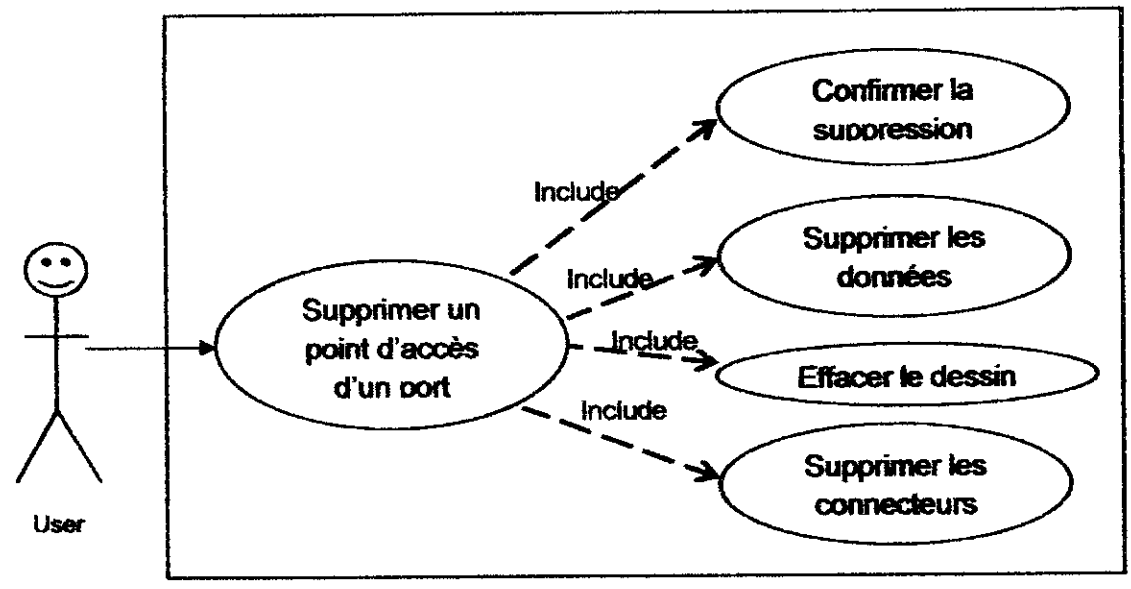

Figure 40: Diagramme de cas d'utilisation : Supprimer un point d'accès d'un port

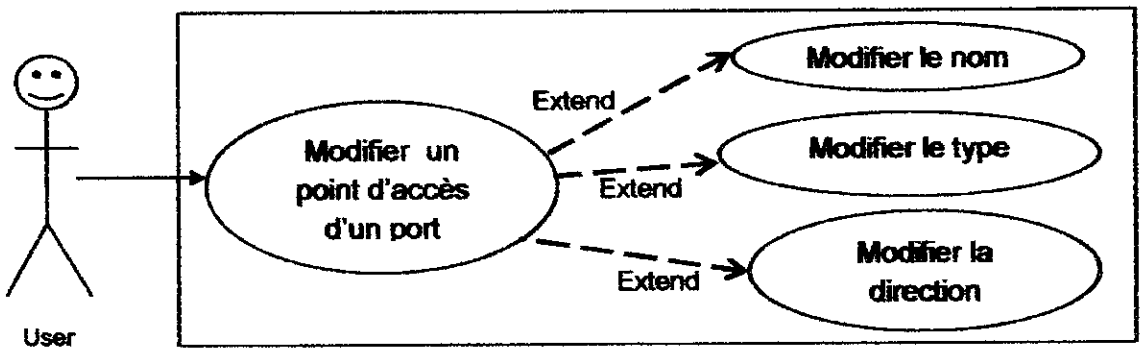

Figure 41: Diagramme de cas d'utilisation : Modifier un point d'accès d'un port

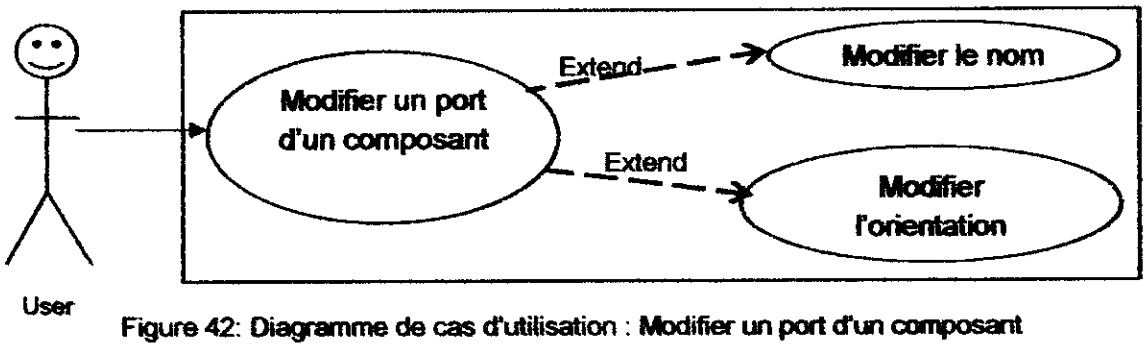

Figure 42: Diagramme de cas d'utilisation : Modifier un port d'un composant

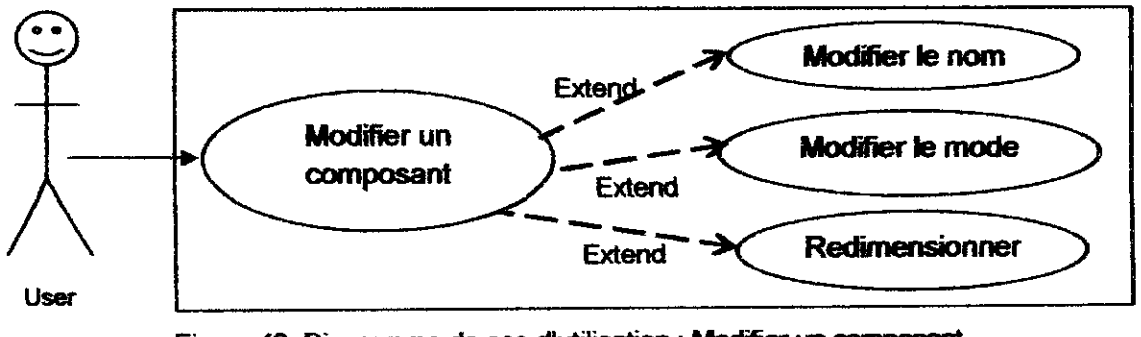

Figure 43: Diagramme de cas d'utilisation : Modifier un composant

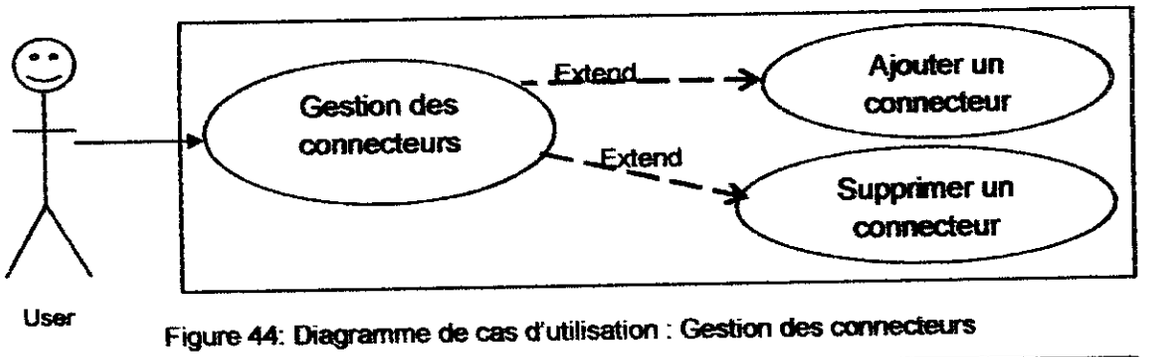

Figure 44: Diagramme de cas d'utilisation : Gestion des connecteurs

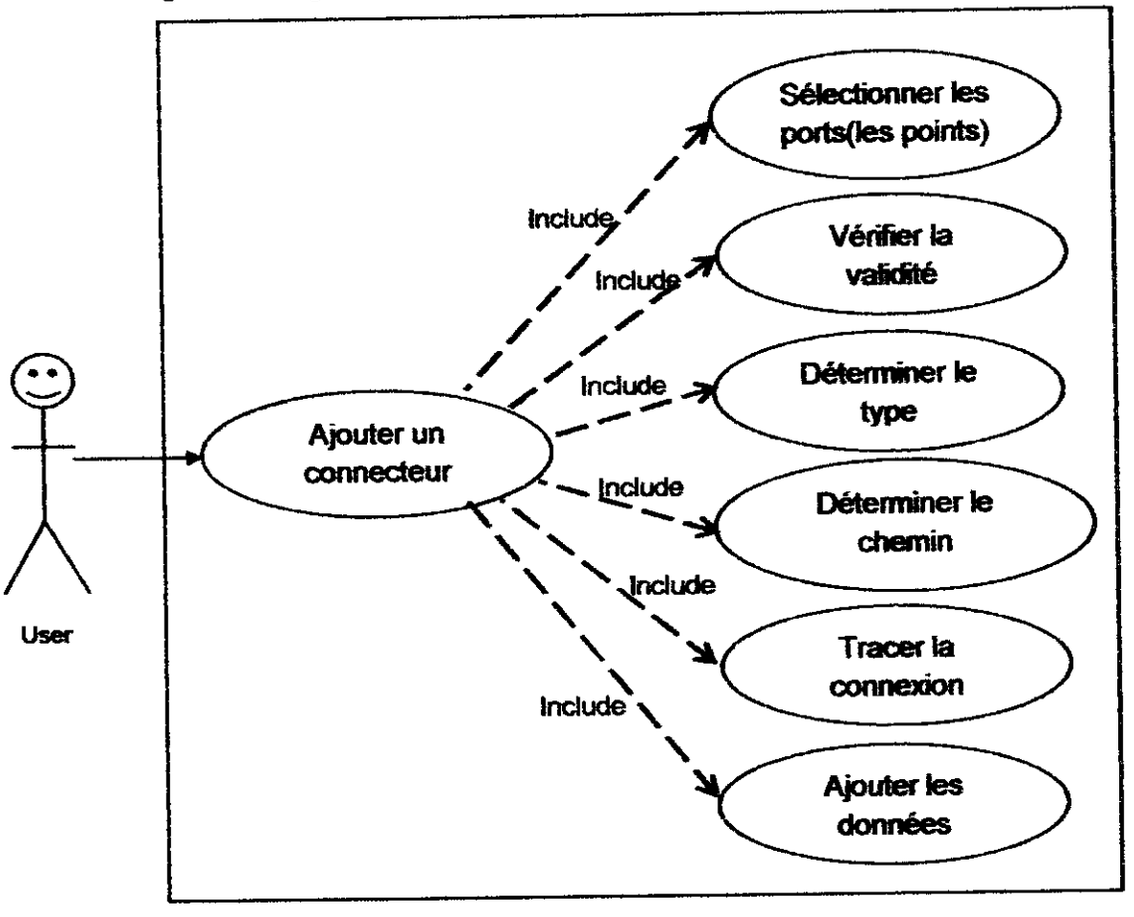

Figure 45: Diagramme de cas d'utilisation : Ajouter un connecteur

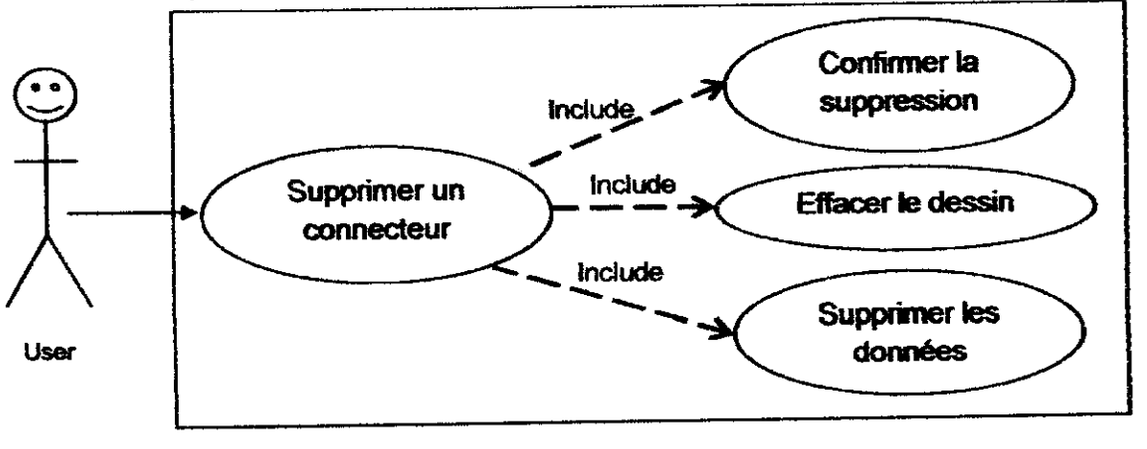

Figure 46: Diagramme de cas d'utilisation : Supprimer un connecteur

II.4 L'analyse

Le bute de cette étape est de déterminer les éléments intervenant dans notre système, ainsi que leur structure et leurs relations.

II.4.1 La représentation graphique

Comme nous avons vu dans les objectifs, l'objectif principal est de permettre le dessin d'une architecture basé sur le modèle IASA. Ainsi nous devons dessiner les composants, les ports et les connecteurs. Donc nous allons présenter les différentes parts de la manière suivante :

- Les points d'accès :

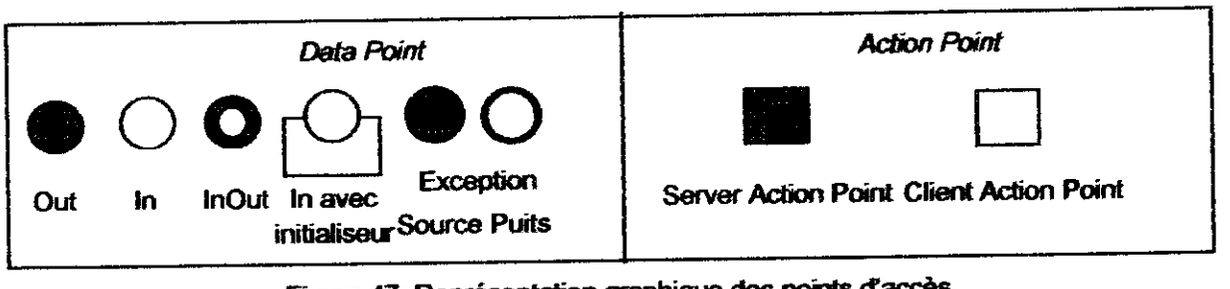

Figure 47: Représentation graphique des points d'accès

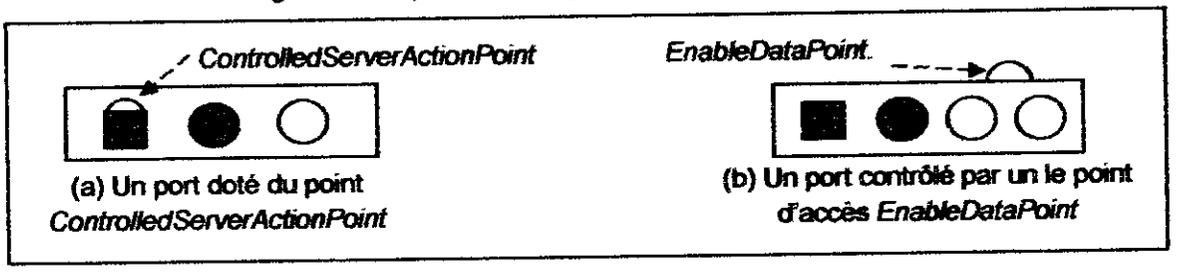

Figure 48: Représentation graphique des points d'accès de contrôle

- Les ports : comme un port est un ensemble des points d'accès, la représentation graphique des ports c'est un rectangle qui contient des points d'accès.

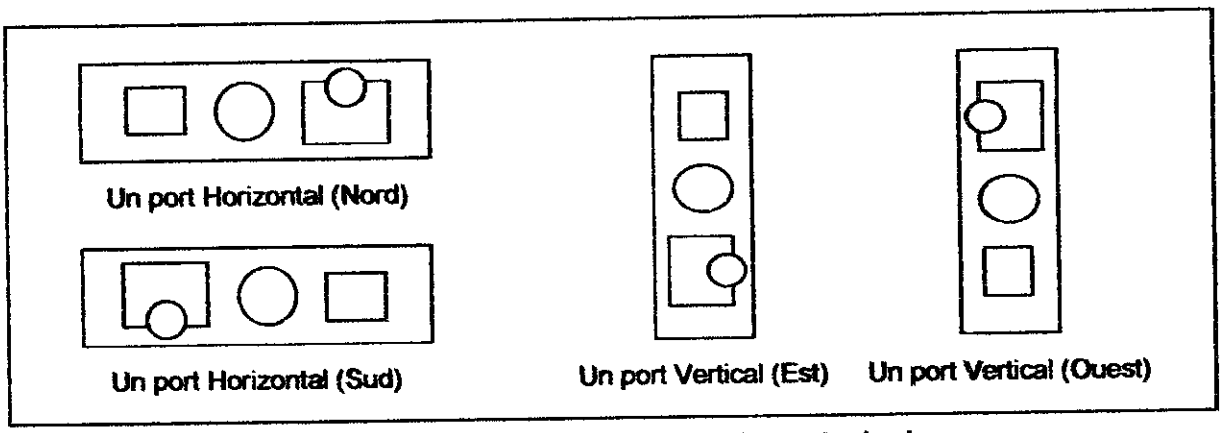

Figure 49: Représentation graphique des ports simples

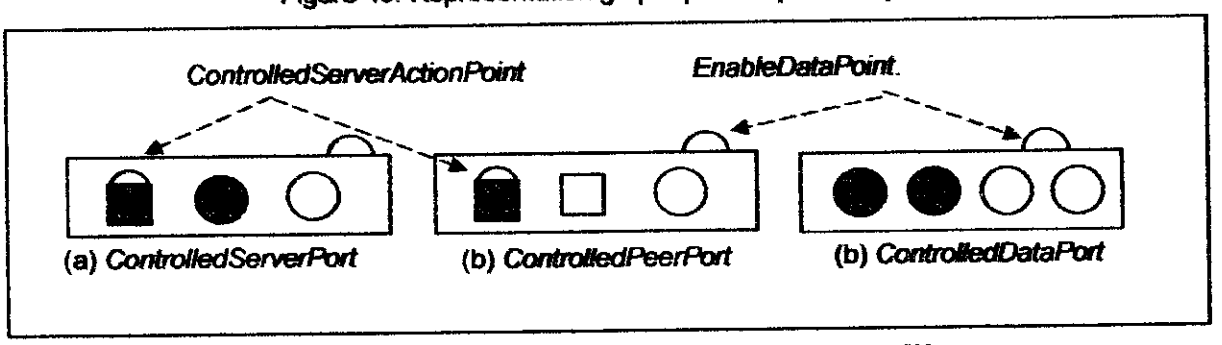

Figure 50: Représentation graphique des ports contrôlés

➢ Les composants : Pour les composants de base, la représentation est un simple rectangle avec le nom du composant (ou instance) dans le coin gauche supérieur. Et pour les composants composites la représentation est en deux rectangles collé verticalement l'un à l'autre. Le premier rectangle (celui au dessus) est pour la partie opérative et l'autre pour la partie contrôle. Le nom de composant est dans le coin gauche supérieur de la partie opérative. Les ports de composant sont collés sur les bords du composant (chacun selon son orientation).

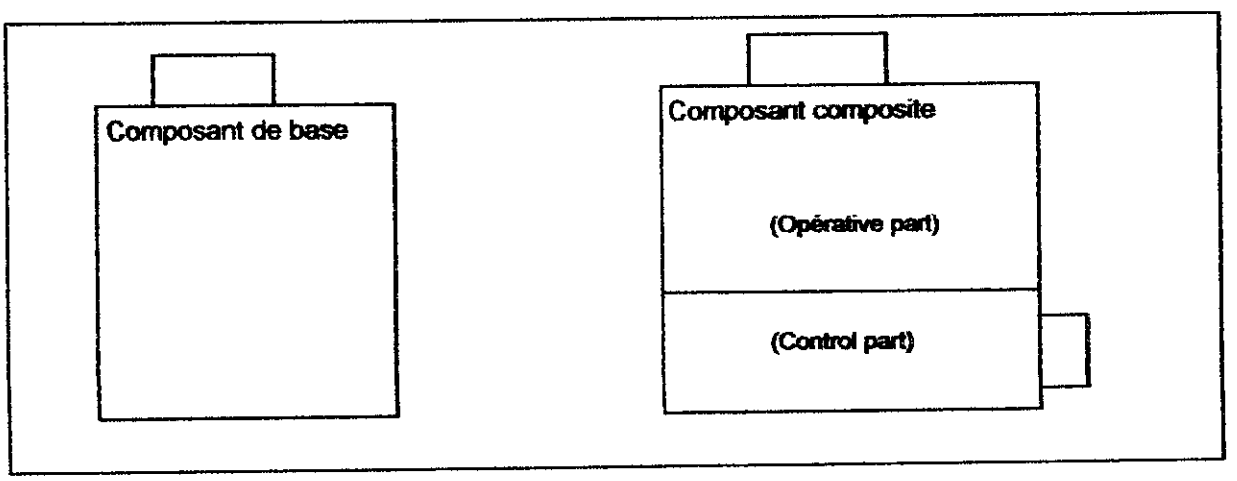

Figure 51: Représentation graphique des composants

➢ Les connecteurs : Nous avons vu en chapitre I, qu'il existe plusieurs types des connecteurs. Pour un connecteur élémentaire, la connexion se fait par un simple trait entre les ports (ou les points d'accès). Mais pour un autre type de connecteurs, nous avons vu que le connecteur est en fait un composant de connexion (voir chapitre I section 5.5). Donc la connexion est composée de deux parts, un composant de connexion et les connecteurs élémentaires entre le composant de connexion et les composants connectés.les connecteurs élémentaires sont les mêmes et le composant de connexion est représenté de la même manière que les autres composants.

II.4.2 Diagramme de classe de la représentation graphique

Nous avons vu précédemment dans la spécification des besoins qu'un projet est un ensemble de composants (simples et/ou composites) interconnectés par un ensemble de connecteurs. Un composant simple contient un ou plusieurs ports et un port contient un ou plusieurs points d'accès. Un connecteur connecte deux ou plusieurs ports (ou points d'accès).

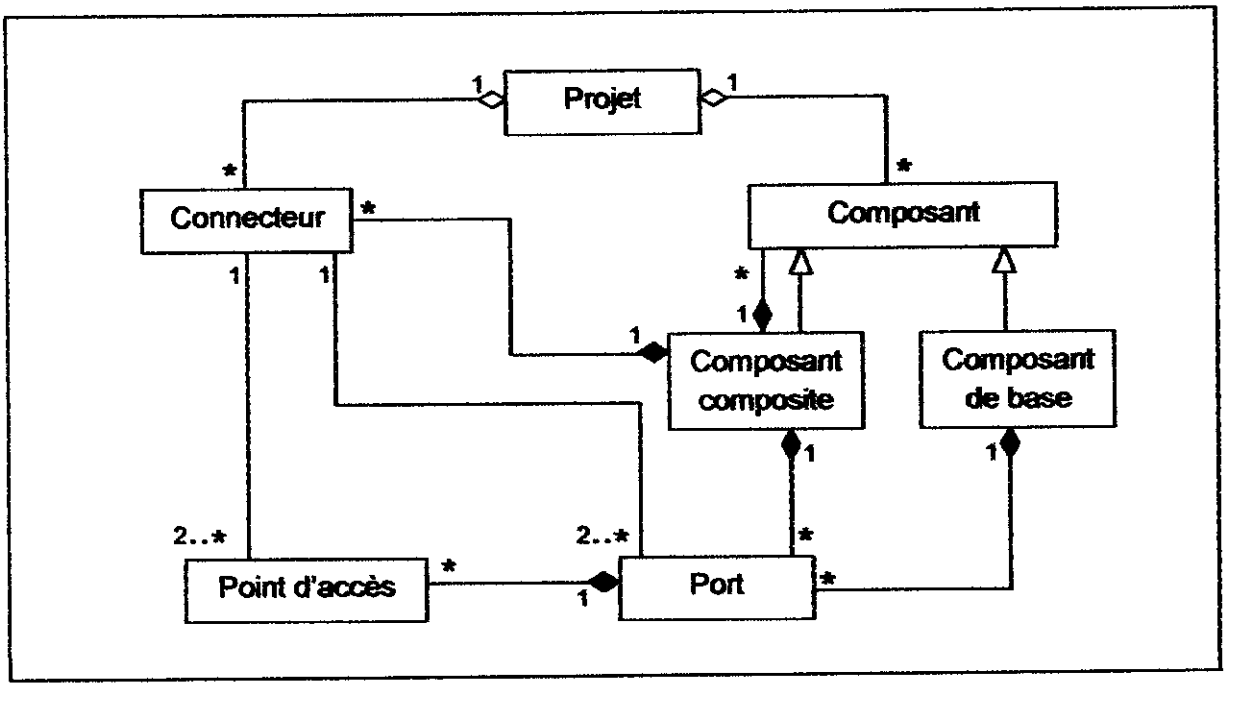

Figure 52: Diagramme de classe de la représentation graphique

II.4.3 La représentation textuelle des données

Après le dessin de l'architecture, l'IDE doit générer un code qui représentera cette architecture. Le forma du code générée est donnée par le langage de spécification de modèle IASA. Le langage de spécification représente les composants sous forme de clause. Une clause composant contient à lui aussi des clauses qui représentent les composants et les ports internes ainsi que les ports externes. Un composant de base dans ce langage de spécification est un composite dont les clauses de composants et de ports interne est vide.

II.4.3.1 Les clauses d'un composant

package nomDePackage;*// c'est le nom du composant*

import listeDePackage *// packages de composants et des connecteurs interne*

component nomDuTypêDeComposant {

// Définition de types internes. Pour être instancié dans d'autres composants,

// Un type doit être définis à l'extérieur du composant, soit dans un même //fichier ou dans un fichier différent.

port {

// Définition de types internes de port

}

connector {

// Définition de types internes de connecteurs.

}

component {

// Définition de types internes de composant.

}

////////////// Fin de la définition des types internes

/// Définition des instances

ports{ *// définition des ports du composant*

}

operativepart{ *// description de la partie opérative:*

// instance de composant, type internes de composant, et connexions

}

controlpart{*// description de la partie contrôle:*

//instance des composants et des connexions interne de la partie contrôle et //les interaction avec les composants de la partie opérative

}

behavior{ *// Comportement du composant; Ce comportement représente*

//celui du composant OpPartController de la partie control, le //comportement est décrit soit en SEAL, transformable directement en

// langage de programmation ou en java ou c'est une référence à un //fichier Java

}

properties{ *// Spécification des propriétés non fonctionnelles.*

}

} *// Fin description type de composant*

II.4.3.2 La clause port de définition d'un type de port

Cette clause est spécifiable indépendamment dans un fichier ou à l'intérieur de la définition d'un composant et dans ce cas le type est considéré comme interne. Dans les deux définitions il devrait être possible de spécifier un certain nombre de type de port dans une clause port englobant ou directement sans

nécessité de le mettre dans la clause port englobant. La clause englobant est introduite uniquement pour l'organisation du code.

```
port { // Clause englobant
    port nomDuType1 {
        accesspoint{}
        actioncontext{}
        behavior {}
        }
    port nomDuType2 {
        accesspoint{}
        actioncontext{}
        behavior {}
        }
} // Fin de la description des types de ports
```

II.4.3.3 La clause accesspoints

Type Du Point D'accès non Instance liste de paramètres

Exemples :

ServerActionPoint pMainAp (Temp, SYNCHRONE/ASYNCHRONE); // Max 1par port

ClientActionPoint pMainAp (0, SYNCHRONE); // Max 1 par port

IntDataPoint pMainStatus (OUT, 0, SYNCHRONE); // un ou plusieurs

StringDataPoint pin1 (IN, 0, SYNCHRONE) ;

Complex DataPoint pin2 (IN, 0, SYNCHRONE) ;

II.4.3.4 La clause actioncontext

action *listeD'actions* **implemented by** *Signature d'une méthode;*

action *listeD'actions ; // Aucune voie pour l'implémentation n'est encore définie. Ce sont des actions abstraites*

Signature Méthode=Instance_Datapoint nomMethodName (liste instanceDataPoint)

II.4.3.5 La clause connector

Comme pour les ports, les types de connecteurs peuvent être définis dans une seule clause ce qui donne plus de clarté et d'organisation, ou dans des clauses connector indépendantes. Dans le cas d'une clause englobant une seule est permise dans une description

connector {

//Définition de type de connecteurs

}

II.4.3.5.1 Définition d'un nouveau type de connecteur

Connector nomDuNouveauType **{**

//Déclaration des rôles;

//Définition de l'architecture du connecteur en utilisant d'autres types de //connecteurs

//Définition du contexte d'action

//Définition du comportement

}

II.4.3.5.2 Déclaration des rôles

Dans la déclaration des rôles, il y a la possibilité d'écrire plusieurs lignes de déclaration. Une ligne est comme suit :

roles liste de rôles séparé par des virgules et se terminant par un point virgules;

II.4.3.6 Définition de l'architecture

Architecture {

//Déclaration des connecteurs utilisés dans l'architecture (instanciation)

//Attachement des rôles des divers connecteurs

}

Remarque :

- **Pour le moment nous n'intéressons pas aux clauses comportement (behavior), donc ces clauses seront vides dans la description textuelle.**
- **Nous supposant que: un port ne remplis qu'une seule action.**

II.4.4 Le diagramme de classe de la représentation textuelle

Comme nous venons de voir, un composant est composé de plusieurs clauses imbriqué. Dans le cas d'un composant de base seul la clause « ports » contient les données de composant (vu que nous n'intéressons pas a la clause « behavior ».

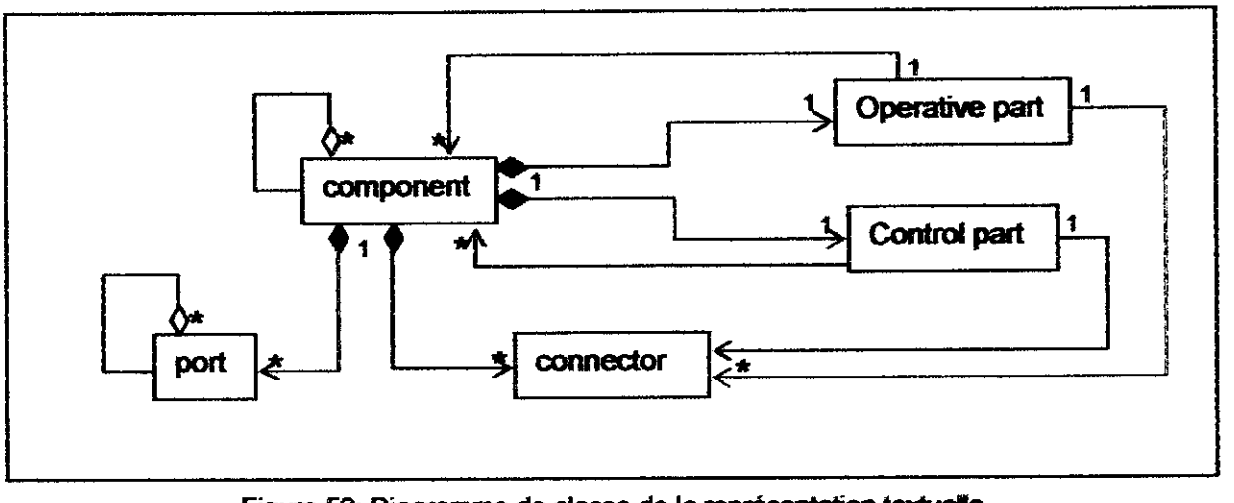

Figure 53: Diagramme de classe de la représentation textuelle

II.4.5 Diagramme d'activité

Les diagrammes d'activités permettent de mettre l'accent sur les traitements. Ils permettent alors de représenter graphiquement le comportement d'une méthode ou le déroulement d'un Cas d'utilisation.

Nous allons par la suite présenter certain cas d'utilisation en utilisant le diagramme d'activité.

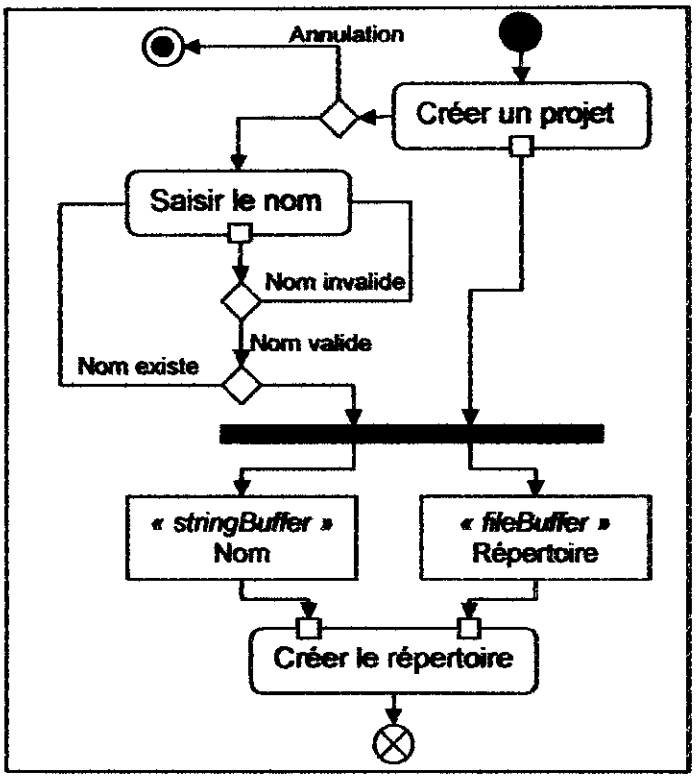

Figure 54: Diagramme d'activité : Création d'un projet

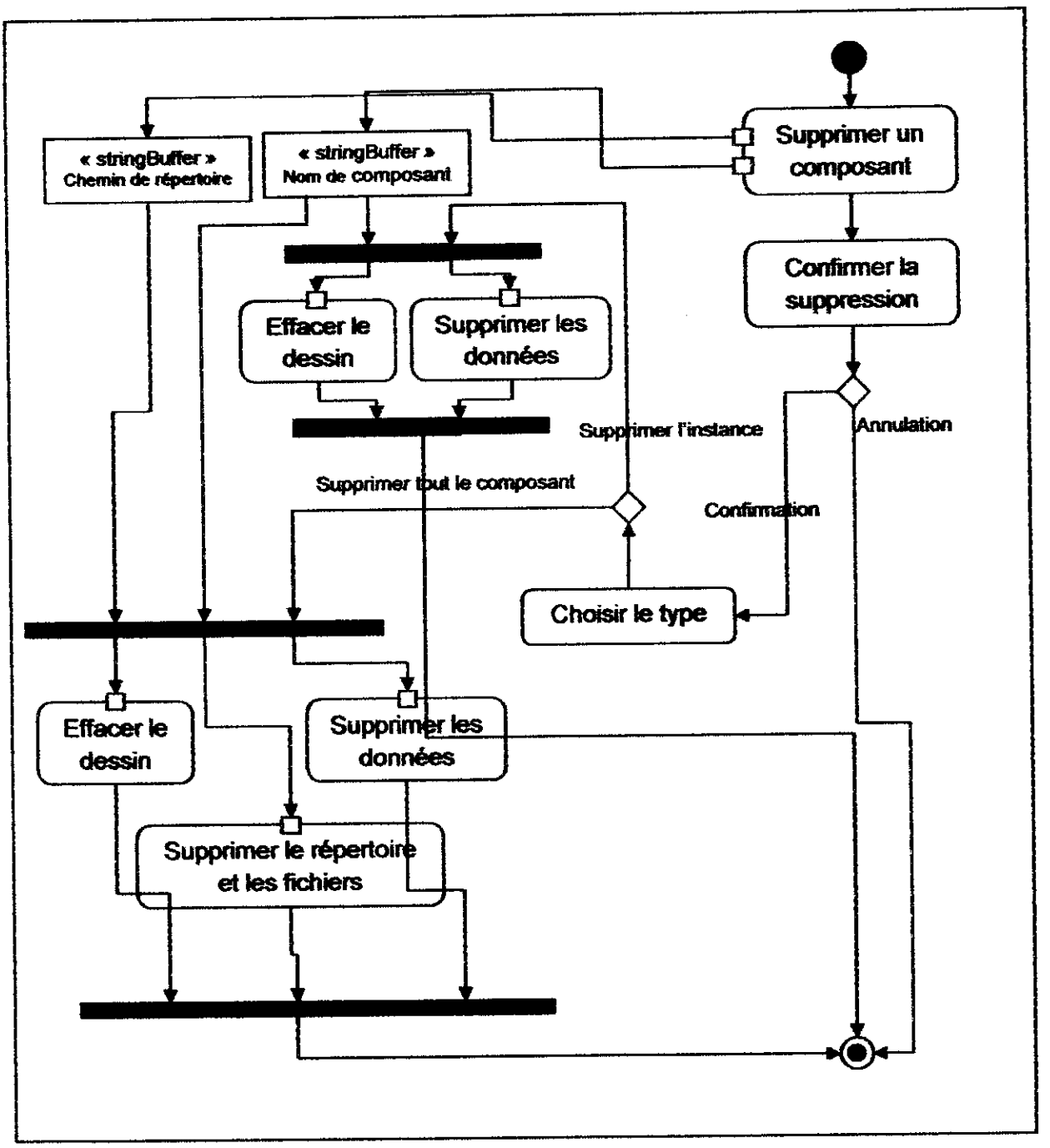

Figure 55: Diagramme d'activité : Suppression d'un composant

II.5 La conception

La conception s'intéresse d'abord au « comment », à savoir la solution du problème énoncée. Elle commence par une conception dite « globale » qui décrit l'architecture de système, elle se poursuit par une conception détaillée. Pour cela nous avons procédé comme suit :

- Le diagramme de classes pour représenter l'architecture globale de notre IDE.

- Les scénarios et les diagrammes de séquences pour une représentation détaillée.

II.5.1 Diagramme de classes

Le diagramme de classes est considéré comme le plus important de la modélisation orientée objet, il est le seul obligatoire lors d'une telle modélisation.

Alors que le diagramme de cas d'utilisation montre un système du point de vue des acteurs, le diagramme de classes en montre la structure interne. Il permet de fournir une représentation abstraite des objets du système qui vont interagir ensemble pour réaliser les cas d'utilisation.

II.5.1.1 Diagramme de classes pour la partie dessin

Pour la partie dessin nous voulons faire la modélisation par espace de dessin. Nous savons qu'un projet est un espace de dessin qui va contenir les composants et les connecteurs. Un composant a son tour est un espace de dessin qui ne va contenir que les ports pour un composant de base, et un ensemble de composants, connecteurs et de ports pour un composant composite. Un port à lui aussi va contenir des points d'accès. Nous avons donc le point d'accès comme étant le plus petit espace de dessin. Nous avons donc le diagramme suivant :

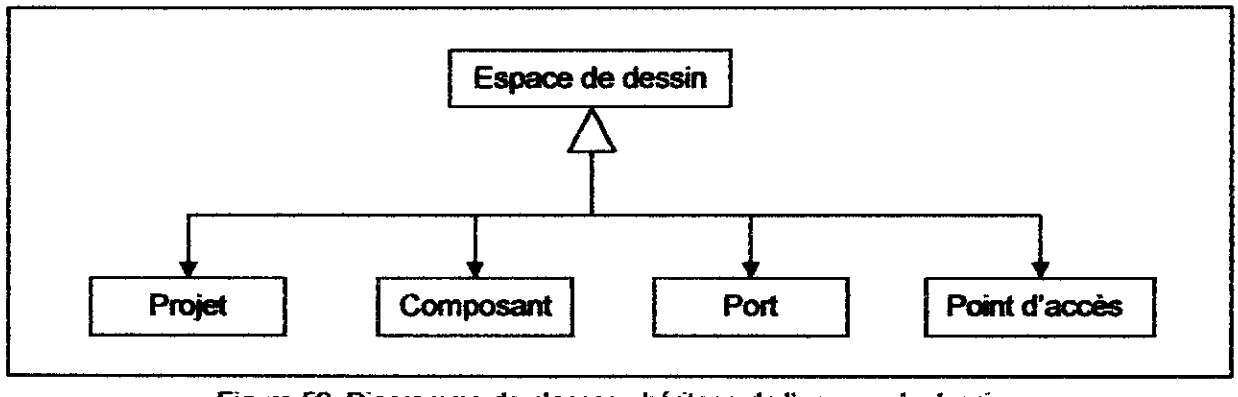

Figure 56: Diagramme de classes : héritage de l'espace de dessin

La classe « point d'accès » est une classe qui a pour attribut le type du point. Grace a cet attribut on peut décider le dessin que va contenir l'espace

de dessin (voir figures 49,50). Il reste à savoir l'orientation du port pour orienter le dessin.

Par contre la simplicité de la classe « point d'accès », la classe composant va contenir 4 autres espaces de dessin (un au Nord, un au Sud, un a l'Est et un a l'Ouest) ce sont les conteneurs des ports. En plus les espaces de dessin réservés aux ports, le composant peut être un composant simple ou composite avec deux représentations différentes (voir figure 53). Un composite va contenir deux autre espace de dessin, un pour la partie opérative et un autre pour la partie contrôle. Nous avons donc le diagramme de classes suivant :

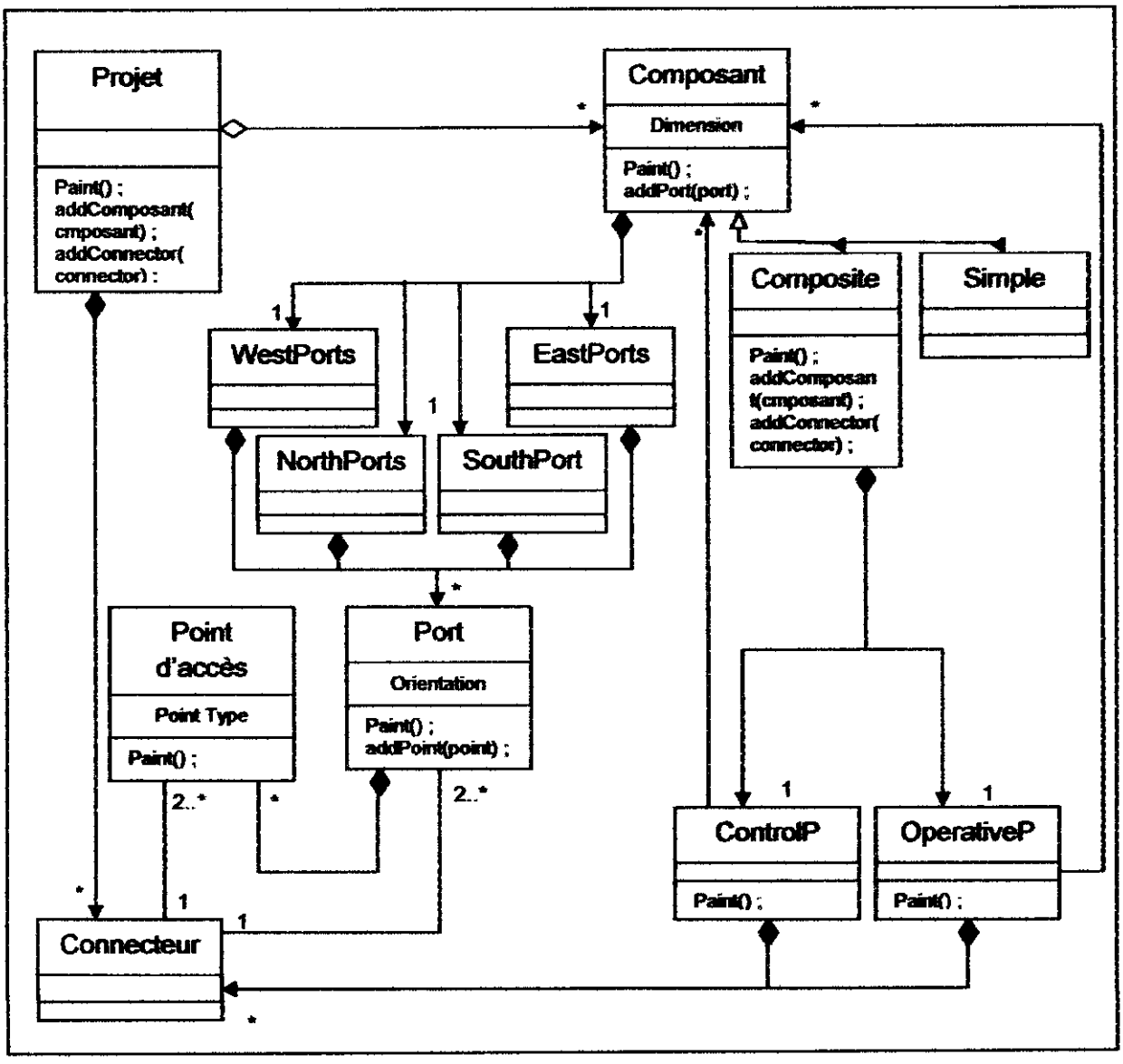

Figure 57: Diagramme de classes : partie dessin

La fonction « Paint() » est invoquer lors de l'ajout du point, port ou composant au dessin de l'architecture, ou lors d'une modification des paramètres. A titre d'exemple, on invoque cette méthode si on change le type d'un point d'accès (par exemple on change le type de IN a OUT), alors cette fonction va se charger d'importer la modification au dessin de point d'accès.

II.5.1.2 Diagramme de classes pour la partie données

Nous avons vu précédemment qu'un composant est un ensemble de clauses. Pour définir ces clause nous avons besoin de certain paramètres comme le nom d'un composant (qui va devenir le nom de package donc le nom de dossier contenant les fichiers qui définissent un composant. Dans le diagramme de classes suivant nous allons voir les paramètres ainsi que les fonctions qui entreront par la suite dans la réalisation.

Nous considérons tout composant (nous nommons la classe par component pour faire la différence par rapport a la classe dédiée au dessin) comme étant un composite, et pour un composant de base c'est un composant dont les parties opérative et contrôle sont vide. Etant donné qu'un projet est un assemblage de composants il sera considéré comme un composant composite, mais on ajoute la classe projet (project) pour faire la différence entre un composite et un projet.

La classe connecteur (connector) ne contient que les instances des ports (ou points d'accès), la génération de clause connector se fera d'après les types des ports (points d'accès) et les données contenant dans ces derniers.

Les classes Port (Ports) et Point d'accès (Points) contient tous les données nécessaire à la génération des clauses ports et port ainsi que la clause connector.

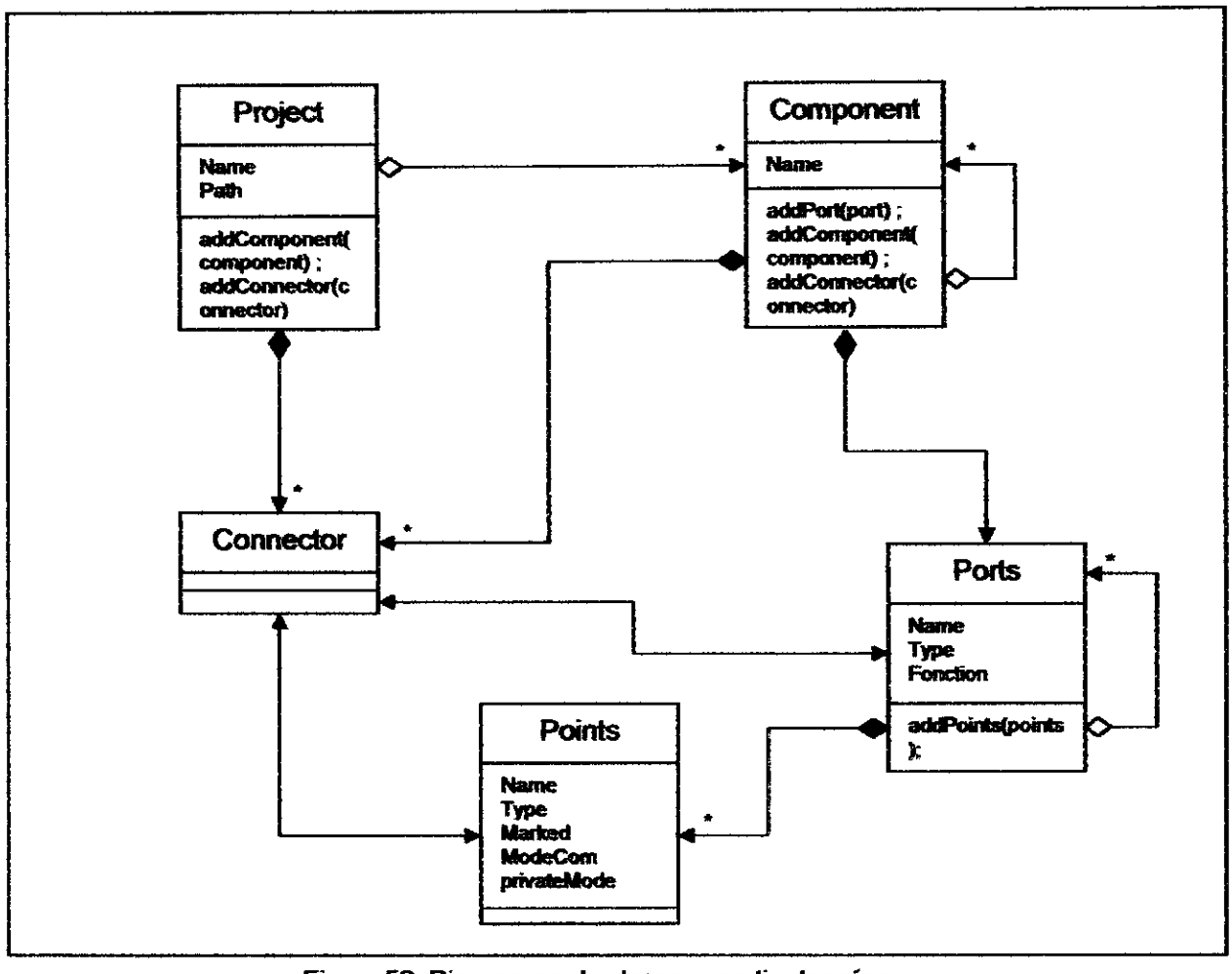

Figure 58: Diagramme de classes : partie données

II.5.1.3 Diagramme de classes général

Dans ce diagramme nous allons faire le lien entre la partie données et la partie dessin. Nous avons dit auparavant que la représentation graphique de composant contient le nom de composant. Alors au lieu de le faire dupliquer dans les deux classes composant (la classe représentant le dessin) et component (la classe représentant les données) nous allons faire une association entre les deux classes. Ainsi le passage des paramètres sera garanti.

Pour les autres classes aussi si l'utilisateur veut savoir le nom ou n'importe quelle autre paramètre, ça sera à partir du dessin, donc nous devons garantie la présence de l'information au niveau de l'affichage sans devoir dupliquer l'information. De ce fait nous avons le diagramme de classes suivant :

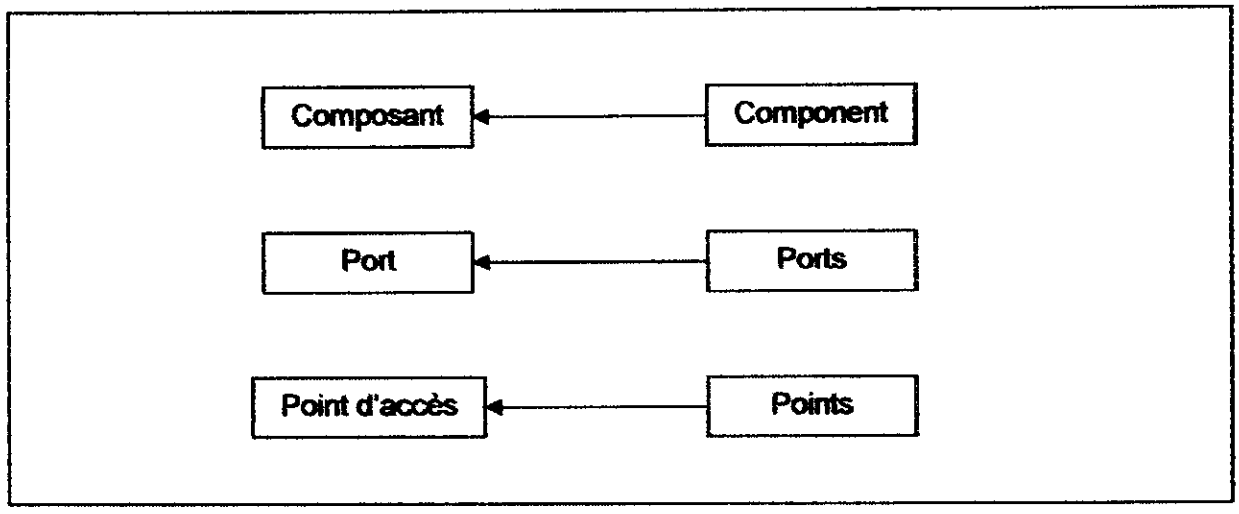

Figure 59: Diagramme de classes : partie données

II.5.2 Scénarios et Diagrammes de séquences

Un scénario est une description détaillée d'une activité, un processus, un traitement ou un cas d'utilisation. Les principales informations contenues dans un scénario sont les messages échangés entre les acteurs, le système et/ou les instances d'objets.

Un diagramme de séquence et la représentation schématique d'un scénario où les messages seront échangés entre les lignes de vie. Une ligne de vie est la représentation dans le temps d'un acteur, un système ou d'une instance d'objet.

Par la suite nous allons présenter quelque action par les scénarios et leurs diagrammes de séquence.

II.5.2.1 Ajout d'un connecteur

Scénario 1 : Ajout réussite

→ L'utilisateur sélectionne un port.

→ Le port envoi un message au programme principale pour voir s'il n y a pas un autre port sélectionné.

→ Le système (programme principale) inscrit le port comme sélectionné et lui envoi un demande de changer le couleur, et change le curseur afin de montrer a l'utilisateur que la sélection est terminée.

→ L'utilisateur sélectionne alors un autre port.

→ Le port envoi le message de vérification au système.

→ Le système étant donné qu'il y a deux ports sélectionnés rétablie le curseur par défaut, envoi un message au premier port afin de retrouver son premier couleur, vérifie la validité de connexion, détermine le type de connecteur et détermine le chemin de connecteur dans l'architecture.

→ Le système crée un connecteur (un objet de la classe connector et un objet de la classe connecteur) et lui envoi le type et les ports qu'on veut connecter.

→ Le connector envoi un message au deux port afin de les marqué comme connecter.

→ Le connecteur envoi un message a l'objet projet afin de lui ajouter au sein de dessin de l'architecture.

Dans ce scénario nous avons mentionné le port (la sélection des ports et l'envoi du message au système). En fait la sélection se fait au niveau de l'objet « port ». Le « port » envoi un signal à l'objet « projet » pour vérifier la sélection. Quand la sélection des deux objets « port » sera fait, l'objet « projet » envoi un signal au système avec les ports sélectionnés pour vérifier la validité de connexion auprès les objets « ports ». Le marquage des ports se fait au nivaux des objets « ports ».

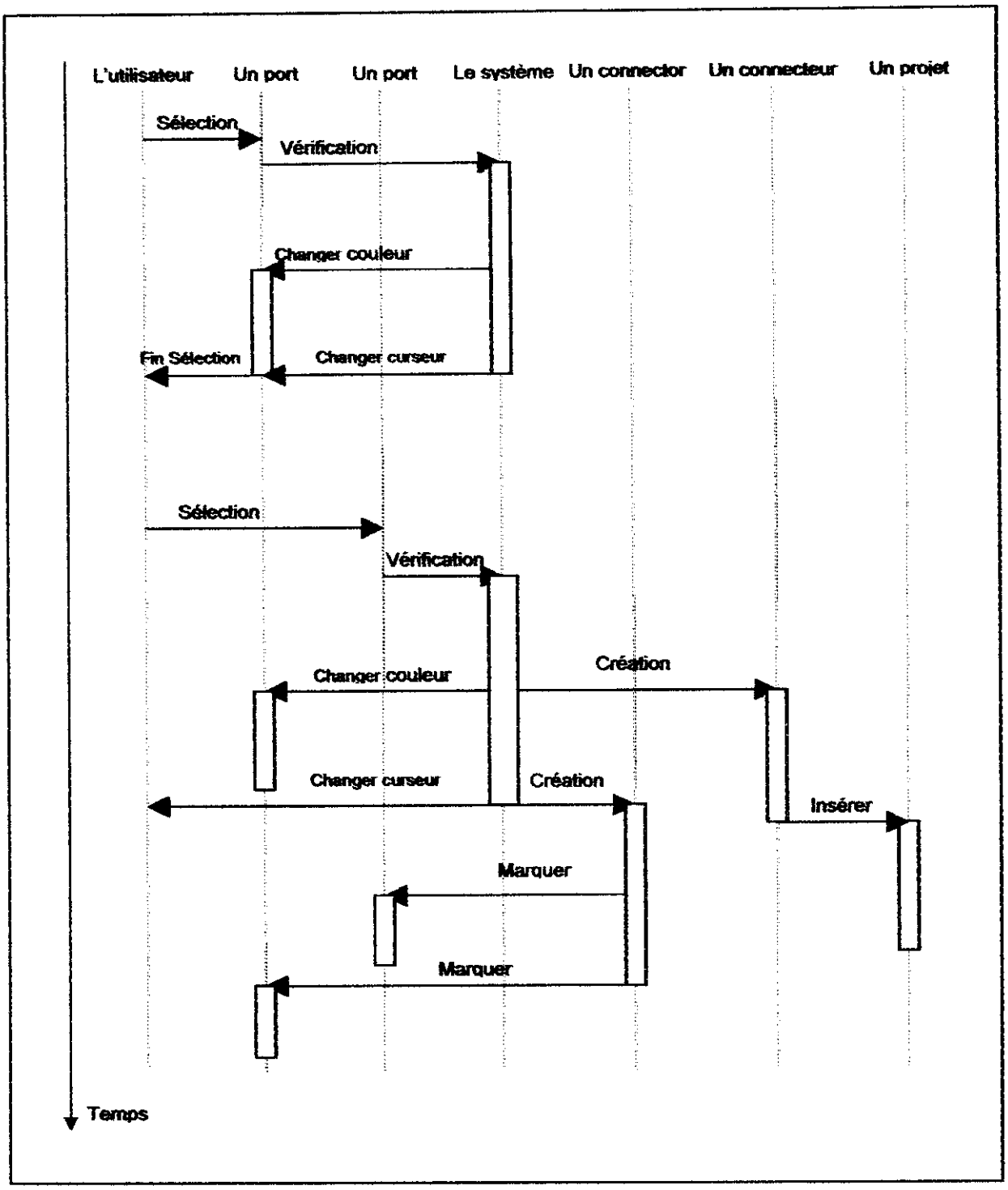

Figure 60: Diagramme de séquence : ajout réussite d'un connecteur

Scénario 2 : Ajout échoué

→ L'utilisateur sélectionne un point d'accès.

→ Le point envoi un message au programme principale pour voir s'il n y a pas un autre point sélectionné.

→ Le système (programme principale) inscrit le point comme sélectionné et lui envoi un demande de changer le couleur, et change le curseur afin de montrer a l'utilisateur que la sélection est terminée.

→ L'utilisateur sélectionne un port.

→ Le port envoi le message de vérification au système.

→ Le système détecte que la connexion peut pas être effectué et envoi un message d'échec à l'utilisateur.

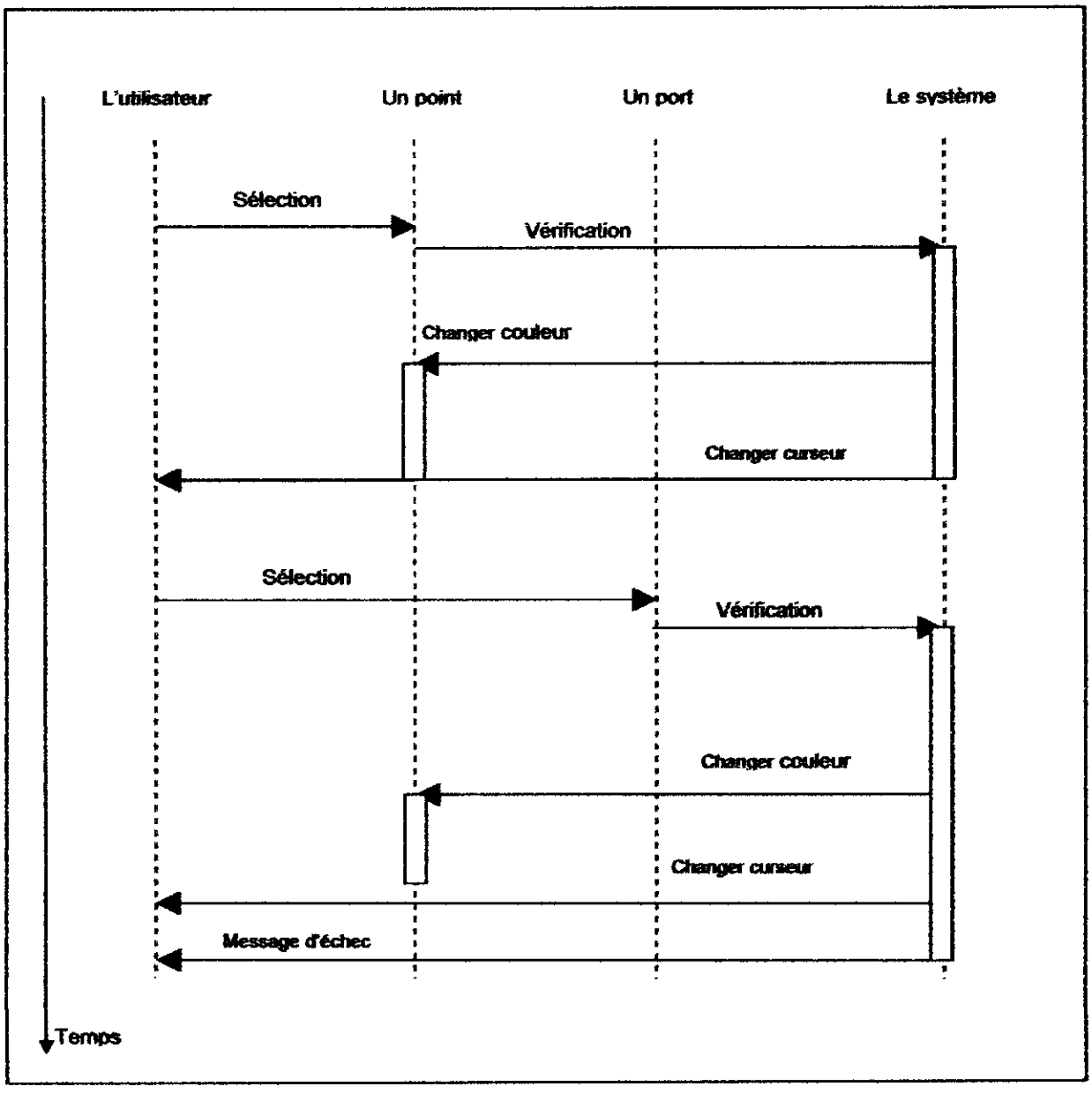

Figure 61: Diagramme de séquence : ajout non réussite d'un connecteur

II.6 Conclusion

Nous venons de voir les différents besoins et les différents qui vont entrer dans la réalisation de l'IDE. Nous avons utilisé pour cela une modélisation orientée objet.

C'est vrai qu'une modélisation objet peut atteindre un niveau très proche du langage de programmation, ainsi indiquer avec précision comment le système sera réalisé, et comment va réagir au différents besoins et utilisations.

Avec son importance, cette étape n'est pas définitive, elle peut très bien contenir quelque erreur.

Les erreurs de la partie conception sont incontournables. Que ce soit par oublie de quelque détaille de la part de concepteur, ou soit par les changements des besoins de l'utilisateur, on devrait toujours changer quelque chose dans la conception.

Ce qui va nous révéler ces erreurs (ou ces manques), n'est autre chose que l'implémentation de l'application dans un langage de programmation, et ça ce que nous allons voir dans le chapitre suivant.

CHAPITRE III

IMPLEMENTATION ET PRESENTATION DE L'APPLICATION

CHAPITRE III : IMPLEMENTATION ET PRESENTATION DE L'APPLICATION

III.1 Introduction

Dans ce chapitre nous allons essayer de montrer le raisonnement que nous avons suivi dans l'implémentation de l'IDE ainsi qu'une présentation de l'IHM (interface homme machine) de notre IDE.

III.2 Implémentation

A ce niveau, il s'agit d'implémenter la solution retenue au niveau de la phase de conception, c'est la phase au cours de laquelle les structures et les algorithmes définis pendant la conception sont traduits dans un langage de programmation

Nous commençons par parler du choix de langage de programmation et les raisons pour les quelles nous avons choisi ce langage. Nous parlons aussi de quelques avantages et inconvénients du langage choisi.

Nous passons par la suite à une brève présentation sur la réalisation des plugins ECLIPSE et sur la méthode de réalisation de plugin que nous avons choisi.

Ensuite nous faisons une présentation générale sur la stratégie de travail que nous avons adapté dans la réalisation, avec quelques problèmes à qui nous avons du faire face avec les solutions proposées.

Par la suite nous essayons d'expliquer un peut le programme principal et présenter les différents packages.

A la fin nous présentons les différentes parts de l'IHM (interface homme machine) de notre application.

III.2.1 Choix du langage de programmation

Notre travail consiste a réalisé un plugin sous Eclipse. Eclipse propose une plateforme de développement des plugins basée sur le langage JAVA. De ce fait, nous n'avons pas d'autre choix que d'utiliser le langage JAVA. Etre obligé

a utilisé le langage JAVA n'est pas un désavantage. Au contraire, le langage JAVA étant un langage orienté objet, la prise on charge des diagrammes de classes sera directe. C'est-à-dire qu'une classe d'un diagramme de classes se traduit directement par une classe JAVA. Et c'est le cas aussi pour les relations du diagramme de classes, chaque relation a une traduction dans le langage JAVA. Par exemple une classe héritée d'une autre classe se traduit par un « *extends* » dans le langage JAVA. Bien que JAVA a l'avantage de l'adaptation directe d'une modélisation objet, il a quelques inconvénients. L'inconvénient majeur qui nous a fait quelques problèmes est le problème de l'affichage dans la bibliothèque des composants visuels « *swing* ».

Si on ajoute un traitement à un composant visuel alors au moment où on lance le traitement l'affichage s'efface. JAVA a mis en place une parade à ce problème, c'est l'utilisation des threads. Un thread pour afficher le composant visuel, et un autre pour le traitement. Ainsi le composant visuel ne s'affecte pas par le traitement.

III.2.2 Réalisation d'un plugin ECLIPSE

La structure d'un plug-in peut être résumé ainsi : un plug-in est un fichier JAR (JAVA archive) classique contenant, en plus de ses classes Java, deux fichiers manifestes (les fichiers META-INF/MANIFEST.MF et plug-in.xml).

Le fichier MANIFEST.MF est exploité par le noyau d'Eclipse pour obtenir des informations sur le plug-in (version, liste des classes visibles, ...) qui serviront notamment à gérer le cycle de vie du plug-in et ses relations avec les autres plug-ins.

Le fichier plugin.xml sert à concrétiser les fonctionnalités d'extensibilité d'Eclipse. Via ce fichier, optionnel, des plug-ins vont déclarer des points d'extension et d'autres se brancher sur ces points d'extension.

Ainsi Eclipse propose deux approches pour développer un plugin. La première approche est d'utiliser les bibliothèques d'ECLIPSE. ECLIPSE propose un ensemble de classe pour la définition intégré de l'interface de plugin. C'est-à-dire l'ensemble des boutons est des menus etc.... serons intégré avec l'interface

d'ECLIPSE. La deuxième approche est de s'appuyer sur une librairie existante (développé auparavant). La première possibilité est d'intégrer directement le fichier JAR dans un plugin existant. L'intégration de fichiers JAR directement dans les plugins pose notamment le problème de la duplication de ces fichiers dans plusieurs plugins. Il est donc généralement préférable de créer un plugin spécifiquement pour chaque librairie. Eclipse propose alors un assistant particulier pour automatiser la création d'un plug-in à partir de fichiers JAR.

III.2.3 Stratégie du travail

Nous venons de voir qu'ECLIPSE propose deux approches pour réaliser un plugin. Dans notre cas nous avons choisi de développer l'IDE indépendamment de l'ECLIPSE en utilisant les librairies de JAVA, puis nous générant le fichier JAR de l'application et on l'insère dans un plugin ECLIPSE crée spécialement pour notre application.

Comme l'IDE comporte plusieurs parties (la partie de dessin, la partie des données et la partie de l'IHM) nous avons pris part de la technique de décomposition par package. Donc nous avons dédié à chaque partie un (ou plusieurs) package. Les package sont :

- *paintPackage* et *actionPackage*: pour la partie dessin.
- *projectPackage* : pour la partie données.
- *mainPackage* : pour le programme principal.
- *popupMenuPackage* et *dialogPackage*: pour l'IHM.

Nous avons prit pour espace de dessin le composant visuel du swing JPanel. Ainsi nous pouvons définir pour chaque classe de dessin (composant, port, point et projet) un JPanel dont il va réaliser les différentes opérations graphiques de chaque classe.

Nous définissons aussi au niveau de chaque JPanel les différents traitements sur les classes (ajout, suppression etc....). Cette technique va nous facilité l'implémentation, au lieu de faire tous le dessin dans un seul JPanel (donc tous le traitement sera dans une seul classe), le partager sur plusieurs JPanel va

partager aussi le traitement, ce qui va nous données plus de lisibilité dans le code, donc tous modification sur le code sera plus facile.

Pour la sauvegarde et la restauration des projets (pour les dessins nous devons sauvegarder la taille et l'emplacement de chaque objet, ces deux derniers ne sont pas présents dans la description textuelle), au lieu de définir un format pour codé le dessin des composants et des connecteurs, nous avons profité de la technique de sauvegarde d'objet (la sérialisation). Avec cette technique nous pouvons sauvegarder un JPanel et le restaurer avec tous ce qu'il contient des objets. Cette technique est un peut couteuse en ce qui concerne la taille (quelque Mo par projet, en comparaison avec quelque Ko par projet si nous utilisons une autre forma), mais ça ne sera pas un grand problème si nous considérons la capacité de stockage des équipements informatiques actuel.

III.2.4 Programme principale

Le programme principal défini dans la classe MainFrame du package mainPackage, contient en plus de la fenêtre principale de notre IDE, les différentes contrôles sur les autres classes (la sélection des classes, des ports...).

De cette classe nous allons lancer les différentes opérations sur les projets. Soit par un lancement d'une boite de dialogue qui réalisera l'opération:

```
jFrame.setEnabled(false);//désactiver la fenêtre principale
AddProject p = new AddProject(jFrame);/*créer une nouvelle boite de
dialogue de type AddProject(dialogPackage)*/
```

Figure 62: Lancement de la boite de dialogue pour créer un projet

Soit par l'exécution de l'opération sur place :

```
JFileChooser f1 = new JFileChooser();
f1.showOpenDialog(null);//explorer le disque pour choisir un projet
FileInputStream f = null;
try {//lire le fichier du projet choisi
      f = new FileInputStream(f1.getSelectedFile());
} catch (FileNotFoundException e1) {
            e1.printStackTrace();
}
ObjectInputStream o = null;
try {//récupérer le panel du projet sauvegardé
      o = new ObjectInputStream(f);
} catch (IOException e1) {
      e1.printStackTrace();
}
try {
      p1 = (ComponentPanel) o.readObject();

      p1.centerPanel.setPreferredSize(p1.centerPanel.originalSize);

} catch (IOException e1) {
      e1.printStackTrace();
} catch (ClassNotFoundException e1) {
      e1.printStackTrace();
}
try {
      o.close();
} catch (IOException e1) {
      e1.printStackTrace();
}
//l'affichage de projet restaurer
(((ProjectPanel) getActiveTabbedPane()).getCompositPanel()).add(p1);
p1.setBounds(100,100,p1.centerPanel.getWidth()+p1.ePortPanel.getWidth()+
p1.wPortPanel.getWidth(),p1.centerPanel.getHeight()+p1.nPortPanel.getHei
ght()+p1.sPortPanel.getHeight());
```

Figure 63 ouverture d'un projet

III.2.5 Description des packages

Nous avons vu précédemment les packages défini, maintenant nous allons une bref description de quelque package et sur les classe qu'il contient.

III.2.5.1 paintPackage

Ce package contient les classes suivantes :

- AnyPointPaint.java : cette classe contient le dessin des différents types des points d'accès (c'est l'adaptation en JAVA de la classe Point du diagramme de classes : partie dessin figure57).

```
public class AnyPointPaint extends JPanel{

PointType pointType;

. . . . . .

//le dessin des différentes point d'accès

protected void paintComponent(Graphics arg0) {
            // TODO Auto-generated method stub
            super.paintComponent(arg0);
            int x=getPreferredSize().height;
            int y=getPreferredSize().width;
            if(pointType==PointType.ClientActionPoint){
                  arg0.drawRect(4, 4,  12, 12);
            }
            if(pointType==PointType.ServerActionPoint){
                  arg0.setColor(Color.BLACK);
                  arg0.drawRect(4, 4, 12, 12);
                  arg0.setColor(Color.GRAY);
                  arg0.fillRect(4, 4, 12, 12);
            }
            if(pointType==PointType.OutDataPoint){
                  arg0.setColor(Color.BLACK);
                  arg0.drawOval(4, 4, 12, 12);
                  arg0.setColor(Color.GRAY);
                  arg0.fillOval(4, 4, 12, 12);
            }

. . . . . .
```

Figure 64 la classe AnyPointPaint

- ComponentPanel.java : cette classe est l'adaptation de la classe composant. Elle contient les différentes part d'un composant (les ports, les partie opérative et contrôle).

```
public class ComponentPanel extends JPanel{

public EWPortPanel ePortPanel,wPortPanel;/*les conteneurs des ports Est
                                          et Ouest*/
public NSPortPanel nPortPanel,sPortPanel; /*les conteneurs des ports Est
                                          et Ouest*/
public CompositPanel operativePart,controlPart;

public Components components;

public Dimension originalSize;

. . . . . .
```

Figure 65 la classe ComponentPanel

Elle contient aussi les opérations de déplacement, de redimensionnement, d'ajout des ports et de modification de type et de nom.

- CompositPanel.java : le composite panel fait office de récepteurs des différents composants et connecteurs que contient un composite. Il contient donc en plus de l'ajout des composants et des connecteurs, la recherche du chemin d'un connecteur (c'est une longue fonction donc nous ne donnons ici que le pseudo).

```
public List<Point> findLink(JPanel startPanel, JPanel finishPanel, Point
startPoint1, Point startPoint2, Point finishPoint1, Point finishPoint2,
int type){
            List<Point> way=new ArrayList<Point>();

            if(startPoint1.x<=finishPoint1.x){
                  way.add(startPoint1);
                  way.addAll(findWay(startPoint2, finishPoint1));
                  way.add(finishPoint2);
            }
            else{
                  way.add(finishPoint2);
                  way.addAll(findWay(finishPoint1, startPoint2));

                  way.add(startPoint1);
            }
            Link link=new Link(type,startPanel,finishPanel,a,a,way);
            links.add(link);
            return way;
      }
```

Figure 66 la fonction findLink

- EPanel.java : Le conteneur des Est point d'accès.
- WPanel.java : Le conteneur des Ouest point d'accès.
- NPanel.java : Le conteneur des Nord point d'accès.
- SPanel.java : Le conteneur des Sud point d'accès.
- NSPortPanel.java : Le conteneur des ports Nord et Sud.
- EWPortPanel.java : Le conteneur des ports Est et Ouest.
- ProjectPanel.java : dans cette classe nous trouvons un CompositPanel pour contenir les différents composants et connecteurs que

contiennent l'architecture et des composants visuels pour contrôler l'architecture (des boutons de zoom, des scrolle bar pour parcourir l'architecture...).

III.2.5.2 actionPackage : bien que la sérialisation d'un objet permette de sauvegarder et restaurer un objet sans changer un seul des composants qu'ils contiennent, nous devons faire face à un problème. Ce problème est que la serialisation ne permet pas de restaurer les listeners d'un objet.

Les listeners sont des interfaces java qui permettent d'intercepter des événements système sur un composant visuel et d'ajouter une action sur cette événement, un click, mouvement de la sourie, taper sur le clavier...).

Ainsi pour parer ce problème (donc pouvoir restaurer un objet avec ses actions sans devoirs les rajouter a chaque restauration) nous avons créé trois classes : MyActionListner.java, MyMouseListner.java, MyMouseMotionListner.java.

```
public class MyActionListner implements ActionListener,Serializable{

      public void actionPerformed(ActionEvent arg0) {
      }
}


public class MyMouseListner extends MouseAdapter implements
Serializable{
      public MyMouseListner() {
            super();
            }
}

public class MyMouseMotionListner extends MouseMotionAdapter implements
Serializable {
      public MyMouseMotionListner() {
            super();
      }
}
```

Figure 67 Les classes : MyActionListner.java, MyMouseListner.java, MouseMotionListner.java.

III.2.5.3 projectPackage : Ce package contient les classes suivantes : Components.java, Link.java, Points.java, Ports.java et Project.java. Ces classes sont une adaptation directe du diagramme de classes de la partie données (figure 58).

III.3 Présentation de l'application

Maintenant que nous avons vu l'implémentation de la partie opérationnelle de l'IDE nous allons présenter la partie de l'IHM.

Nous commençons par le lancement de l'application. Nous savons que notre IDE est un plugin ECLIPSE, donc le lancement se fera à partir de l'interface ECLIPSE (figure 67)

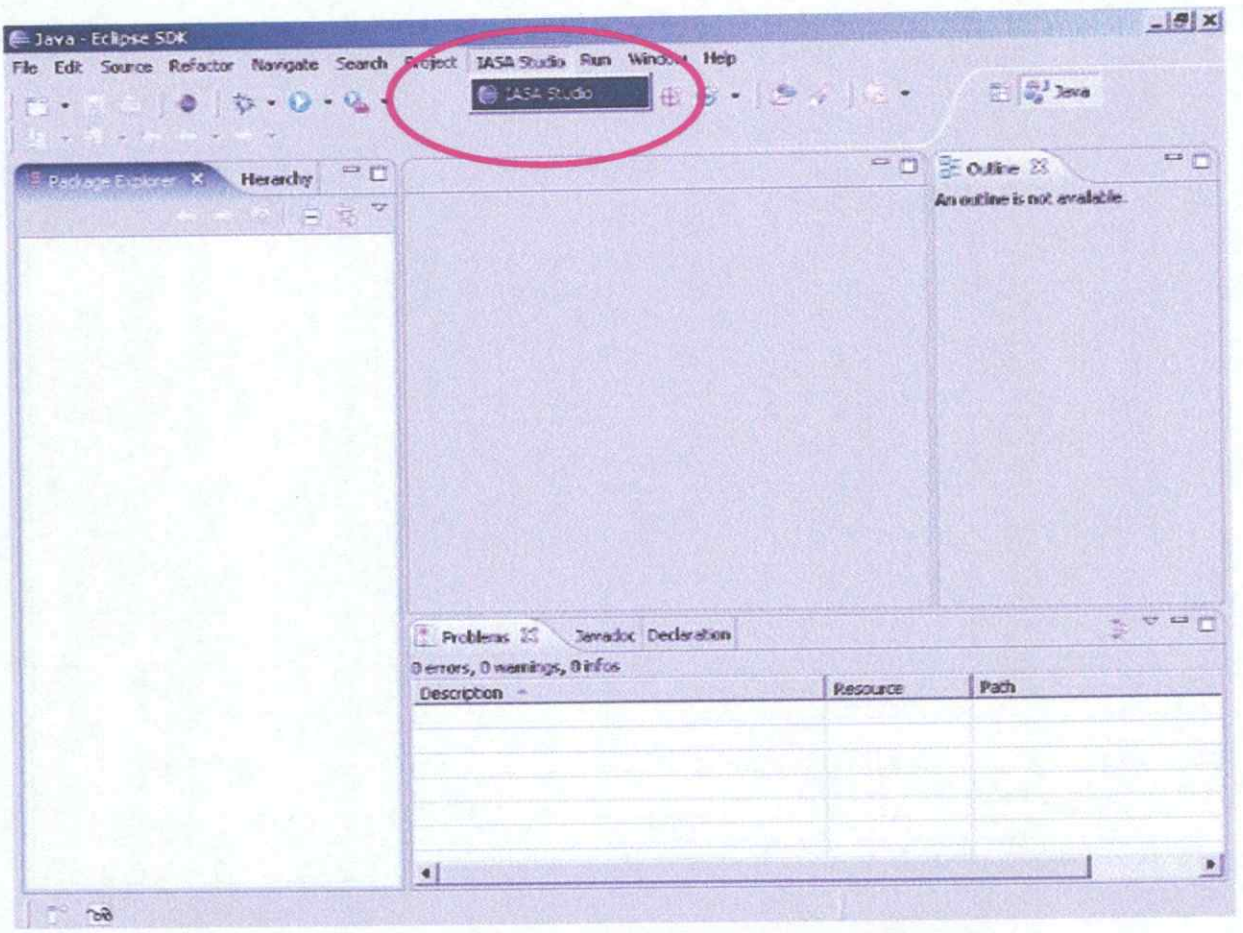

Figure 68 Lancement de l'IDE à partir l'interface d'ECLIPSE

En cliquant sur le menu IASA Studio la fenêtre principale de l'IDE apparaît (figure 68).

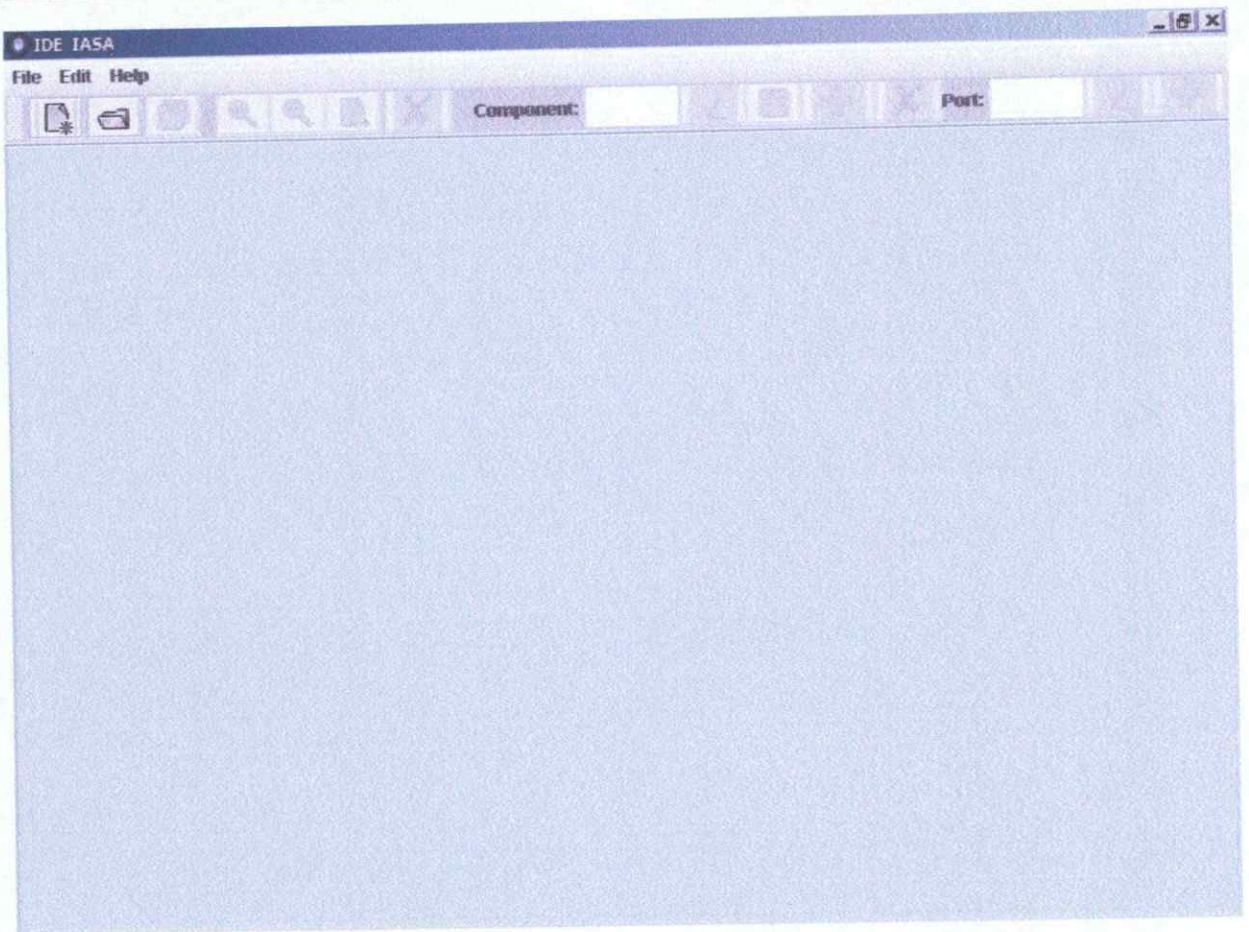

Figure 69 La fenêtre principale

Dans la fenêtre principale nous trouvons une barre d'outils qui contient :

- Trois boutons pour créer un projet, ouvrir un projet et sauvegarder le projet en cours (figure 70)
- Deux boutons pour zoomer et dézoomer un composant sélectionné (figure 70)
- Deux boutons pour ajouter et supprimer un composant (figure 70)
- Un TextEdit pour modifier le nom du composant selectioner et un bouton pour confirmer la modification (figure 71).
- Un bouton pour sauvegarder le composant séléctioné (figure 71).
- Deux bouton pour ajouter et supprimer un port d'un composant (figure 71).

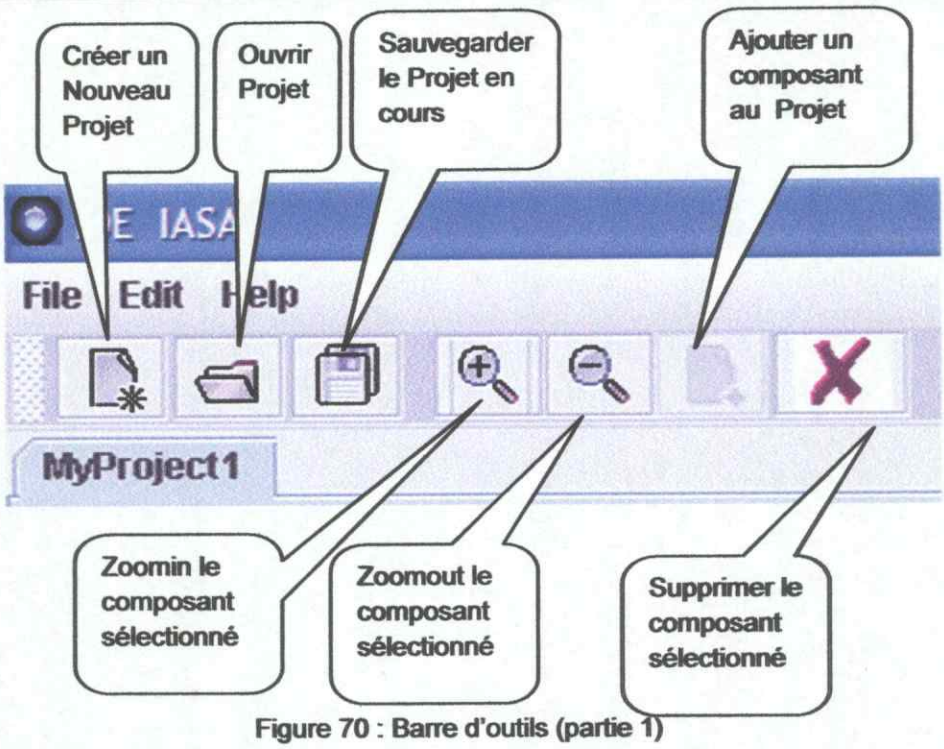

Figure 70 : Barre d'outils (partie 1)

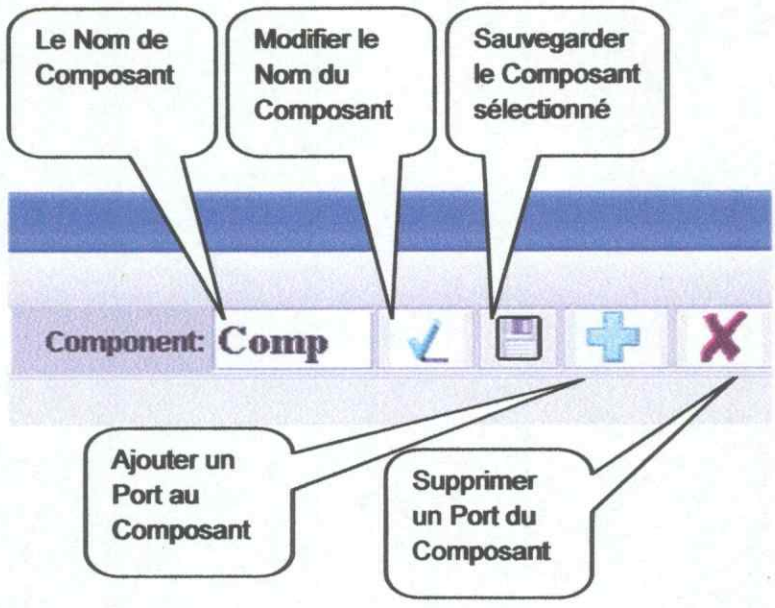

Figure 71: Barre d'outils (partie 2)

- Un TextEdit pour modifier le nom du port selectioner et un bouton pour confirmer la modification (figure 72).
- Un bouton pour ajouter un point data au port sélectionné (figure 72).

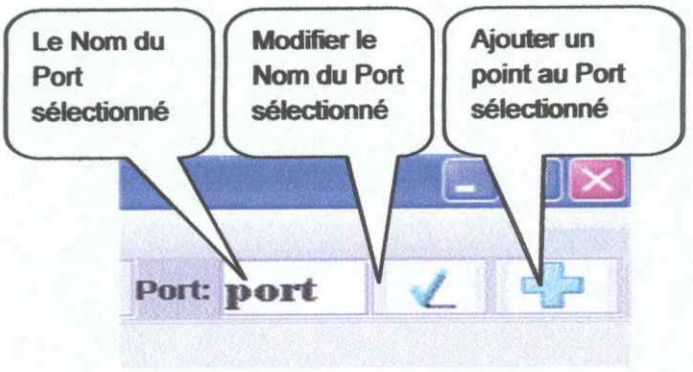

Figure 72: Barre d'outils (partie 3)

Nous pouvons accéder à ces fonctionnalités (les fonctionnalités de la barre d'outils) a partir des PopupMenu : un dans le composant (figure 73), et un autre dans le port (figure 74).

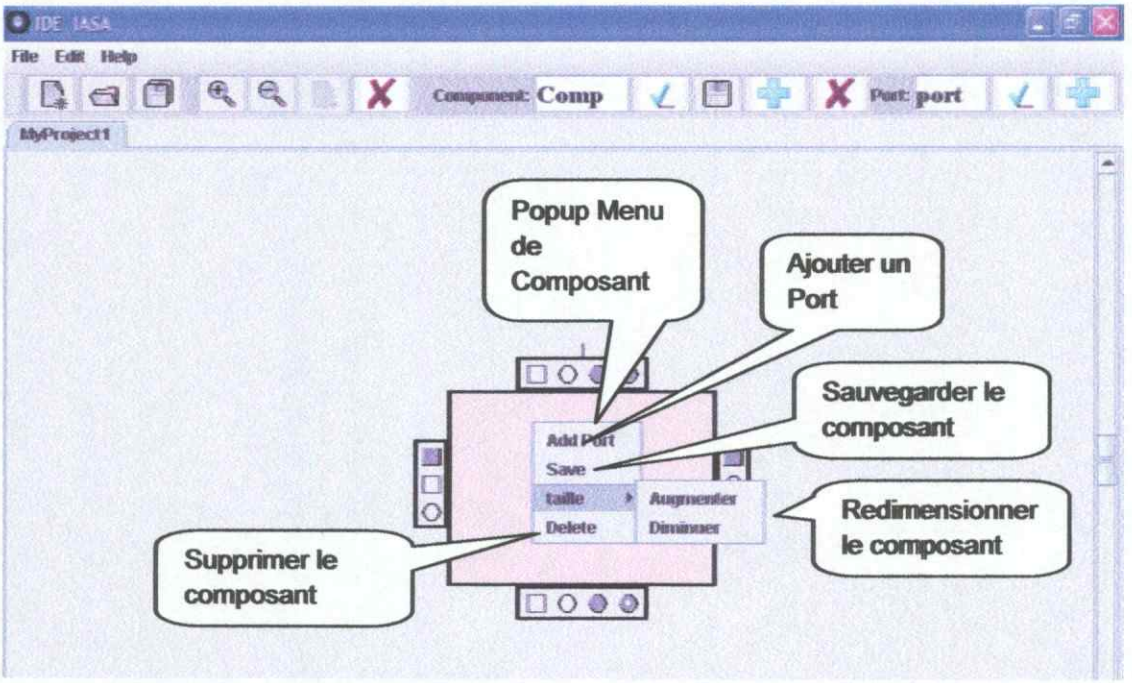

Figure 73: Popup Menu de Composant

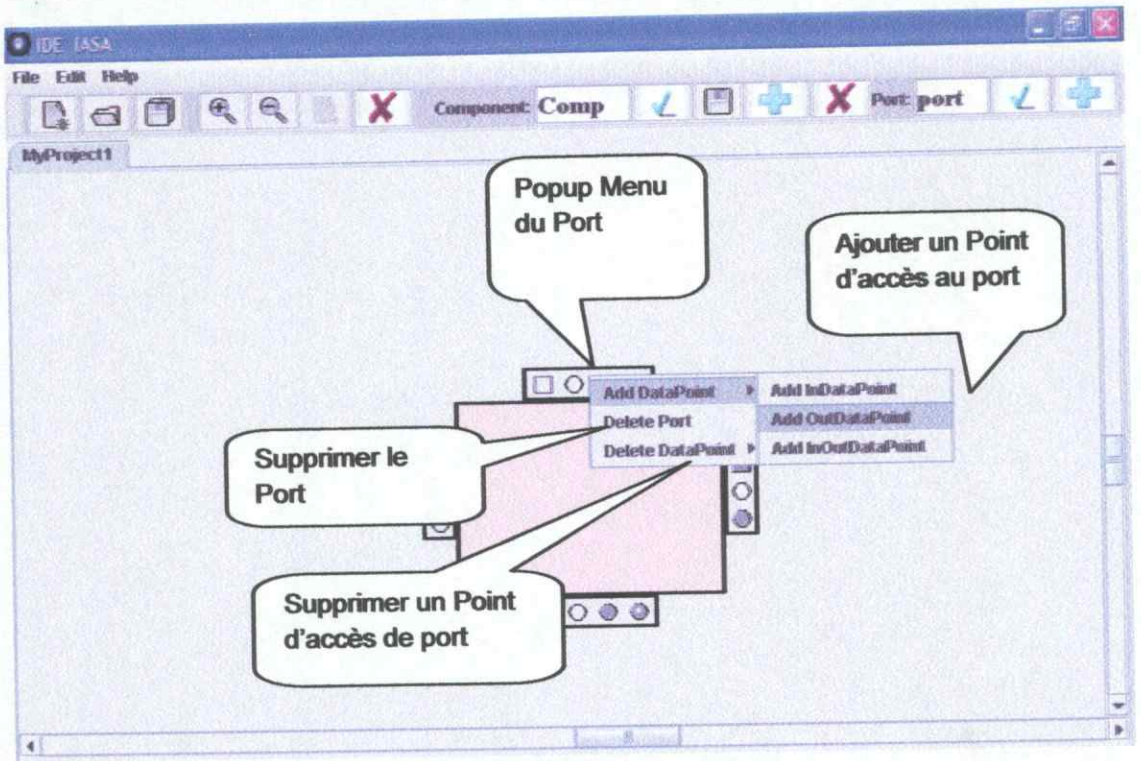

Figure 74: Popup Menu de port

Enfin nous avons les boites de dialogue de la création d'un projet

(Figure 75), de la création d'un composant (Figure 76) et de la création d'un port (Figure 77).

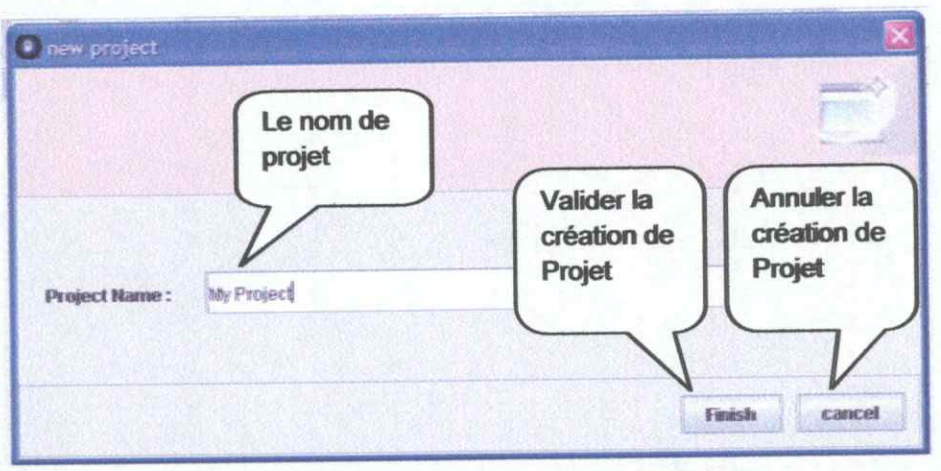

Figure 75: Boite de dialogue : nouveau projet

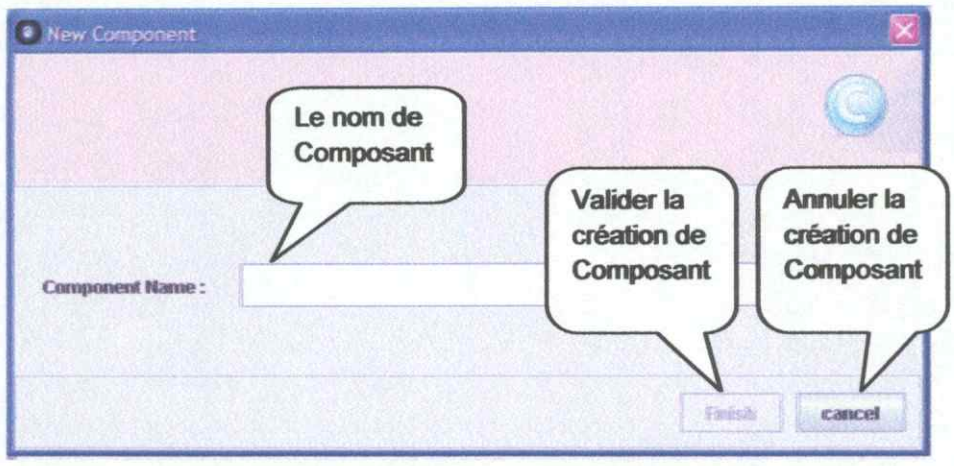

Figure 76: Boite de dialogue : nouveau composant

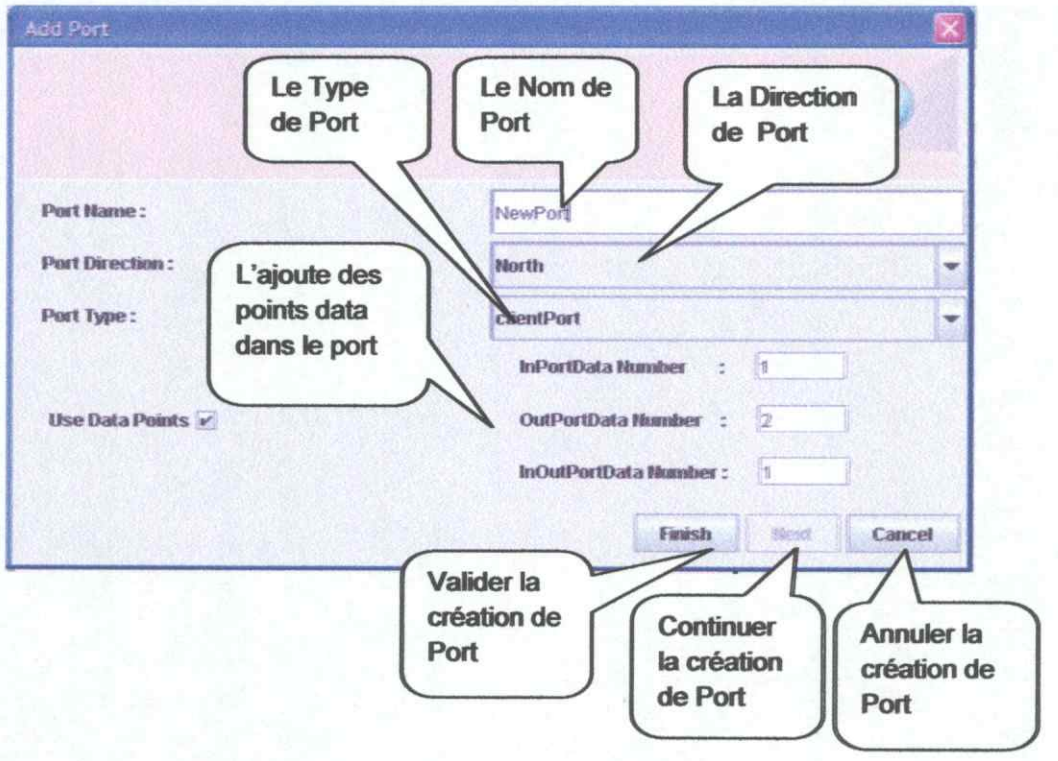

Figure 77: Boite de dialogue : nouveau port

III.4 Conclusion

L'implémentation nous permet de réaliser les différentes solutions que nous avons mises en place dans les parties analyse et conception. Certainement nous avons fait quelque modification et ajout à ces solutions suite aux contraintes du langage de programmation, mais ça ne vous pas dire que les erreurs de conception on été révélé. Ce que nous avons besoins c'est de tester l'application par la réalisation de quelques architecture, ce que nous allons voir dans le chapitre suivant.

CHAPITRE IV

TEST ET VALIDATION

CHAPITRE IV : TEST ET VALIDATION

VI.1 Introduction

Le test permet de réaliser des contrôles pour la qualité du système. Il s'agit de relever les éventuels défauts de conception et de programmation (revue de code, tests des composants,...) [Sommerville, 88].

Donc dans ce chapitre nous allons essayer de tester la capacité de L'IDE à réaliser des architectures basées sur le modèle IASA, et nous allons essayer de mettre la lumière sur quelque point qui donne une grande efficacité à ce modèle.

Dans un premier point nous allons parler sur l'architecture que nous avons choisi pour le teste, et les motivations pour les quelles nous avons choisi cette l'architecture.

Nous passons par la suite a la présentation générale de l'architecture (une vu générale). En fin nous détaillons l'architecture avec le code généré.

VI.1 Choix de l'architecture

Le choix d'architecture va être crucial dans les résultats de test. Donc nous devons faire on sorte que cette architecture va toucher le plus grand nombre de spécification du modèle IASA. Sachant que ce modèle est destiné à développer des applications et des systèmes informatiques, nous avons choisi un système de gestion commerciale des produis d'un réseau X25.

La raisons pour la quelle nous avons choisi ce système est que nous pouvons le réalisé dans un suel composant composite qui va contenir un max des points d'accès différents.

VI.2 Présentation générale

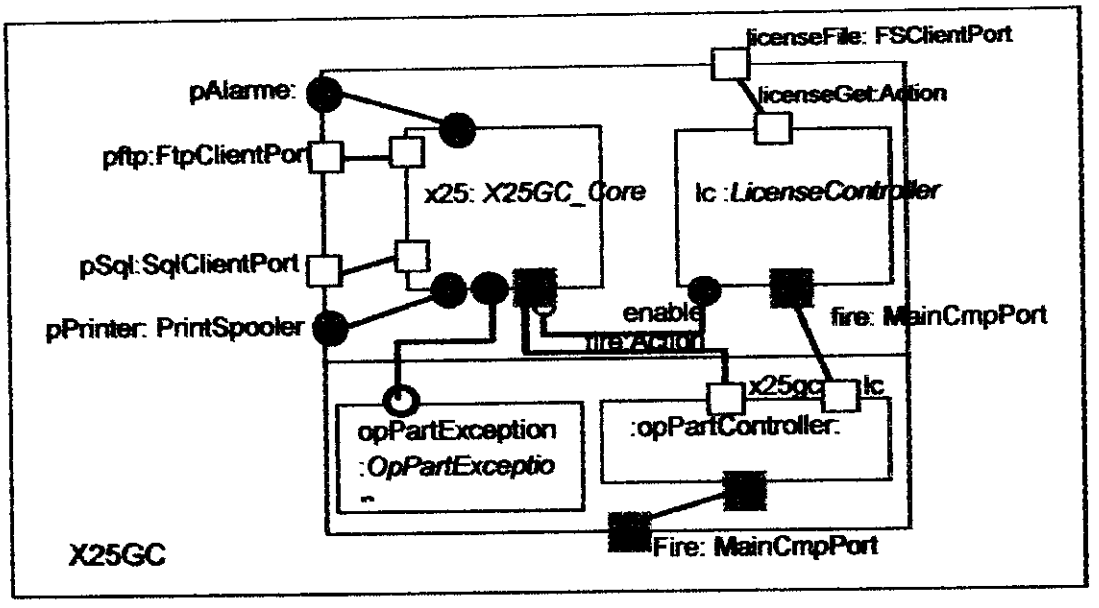

Figure 78: Architecture d'un logiciel de gestion commerciale des produits d'un réseau X25

VI.3 Présentation détaillé

```
/// Description du composant X25GC

package x25gc;

import license.ethernet;

component X25GC {

ports {

        MainCmpPort   pMain {

    accesspoint{ ServerActionPoint pMainAp (0, SYNCHRONE);

        IntDataPoint  pMainStatus    (OUT, 0, SYNCHRONE) ;

    }

                    actioncontext{
```

```
                    action fire implemented by pMainStatus
fireMethod();
}
    behavior{
            rules fireRule;
                    rule fireRule {
precondition: ; // Aucune précondition nécessaire
            pattern: fire,reply,success;
            postcondition:;
                    }
    } // Fin description de pmain

FTPClientPort pFtp {
accesspoint{
                ClientActionPoint cFtpAp (0, SYNCHRONE);
                    StringDataPoint pFtpReplies (IN,
0, SYNCHRONE);
            }
            actioncontext{
                use system.FTPIntercationContext;
            }
behavior{
            boolean ftpConnexionSet = false;
```

```
rules successOpen, errorOpen, closeConnexion,
getTicketFile,
ftpOpenException, ftpReset;
rule successOpen {
precondition: !ftpConnexionSet;
pattern: open(hostname, username, password),
receive(FTP_OPEN_OK),success; fail
ftpOpenException;
postcondition: ftpConnexionSet = true;
}
rule errorOpen {
precondition: !ftpConnexionSet;
pattern: open(hostname, username, password),
receive(FTP_OPEN_ERROR),success;
fail ftpOpenException;
postcondition:;
}
rule closeConnexion {
precondition: ftpConnexionSet;
pattern: close,
receive(FTP_CLOSE_OK),success; fail ftpError;
postcondition: ftpConnexionSet = false;
}
```

```
                                    rule getTicketFile {

precondition: ftpConnexionSet;

                    pattern:    rename(OLD_NAME,    NEW_NAME),
get(NEW_NAME),

                                delete(NEW_NAME), success; fail ftpError;

                    postcondition: ;

                                    }

                                    rule ftpReset {

precondition:;

                    pattern: close, success;

                    postcondition: ftpConnexionSet = false;

                                    }

rule ftpOpenException {

precondition:;

                    pattern: close, success;

                    postcondition:

                                ftpConnexionSet = false;

raise (FTPOpenException);

// Ce patron est réalisé en mode exceptionnel.

// L'exception est soit traité au niveau du composant

// x25: X25GC_CORE ou relayée à la partie contrôle

                                    }

                    } // Fin description des règles d'interactions
```

} // Fin description de pFtp

SqlClientPort pSql{ .... }

PrintSpoolerOutDataPort : pPrinter{ .... }

FSClientPort licenseFile{ .... }

LogDataPort pAlarme{ .... }

} // Fin de la description des ports externes

/// Description de types de connecteurs. Tout connecteur décrit à ce niveau est un type

// interne au composant et ne peut être instancié hors de ce composant

connector {

connector *FireCon* **{**

roles sRole, cRole;

architecture {

CSconnector **csc;**

map sRole to csc.sRole;

map cRole to csc.cRole:

}

actioncontext {fire;**}**

behavior{

interaction fire {

cRole.request(fire); sRole.invoke (fire);

sRole.send(status);cRole.receive(status);

```
success;
            }
        }   // Fin description du type FireCon
} // Fin de la clause définition de nouveau types de connecteurs
operativepart{
    components { X25GC_Core  x25;
    LicenseController lc;
}
connexions { DelegateConnector ftpDelCon { map x25.pftp, pftp; }
    DelegateConnector    alarmeDelCon  {  map  x25.pAlarme,
pAlarme};
    DBasicConnector  enableCon {
     Binding {
            bind  sinkRole  to x25.pMain.enable:;
            bind  sourceRole  to lc.enable;
          }
actioncontext {enable }
behavior{
            interaction enable {
sourceRole.send(enable);sinkRole.receive(enable);
                    }// Fin description de l'interaction enable
}// Fin description des comportements du connecteur
    } // Fin Description du connecteur de donnée DbasicConnector
```

```
}
} // Fin description de la partie opérative
controlpart {
        components {      OpPartController opPartController;
                          ActiveOpPartException opPartException;
        }
connexions {
FireCon fireX25gcCon {
binding  { bind cRole to opPartController.x25gc;
   bind  sRole to x25.pmain;
}
                    // Éventuels Contextes et interaction
additionnelles
actioncontext { } behavior{ }
}
FireCon fireIcCon {
binding  { bind cRole to opPartController.Ic;
   bind  sRole to Ic.pmain;
}
} // Fin clause connexions
}// Fin de la clause controlpart
behavior{ // Définition du comportement du composant
opPartController
```

```
lc.fire;

x25gc.fire;

}



properties{// Description du déploiement des composants

architecture {   // Définition des éléments de l'architecture de déploiement

        environment local {

                machine localhost;

                os UNIX;

        }

environment licenseserver {

machine  licenseserver.test.dz;

os   UNIX;

}

        }

deployment{

deploymentcase {THREAD,  PROCESS; COMPOSITE}

            deploymentmap map1 {

                deploy X25GC as PROCESS in local;

// 1ère ligne; déploiement du composite

                deploy x25 as MAIN_THEAD in  COMPOSITE;
```

```
//lignes suivantes: déploiement des

                deploy lc  as PROCESS in licenseServer;  // instances

        }

}

} // Fin description type de composant X35GC
```

VI.4 Conclusion

Nous avons vu dans ce chapitre un composant composite qui a plusieurs composants interconnectés. D'après la réalisation de ce composant nous pouvons déduire les résultats suivants :

- L'abilité de notre système de réaliser des composants de base.
- L'abilité d'ajouter des port sur ces composants.
- L'abilité d'ajouter des point d'accès sur ces port.
- L'abilité de connecter des composants de base via les connecteurs.
- L'abilité de réaliser des composants composites.

D'après ces résultats nous pouvons dire que notre système à remplir la majorité des termes de la problématique.

Conclusion Générale

CONCLUSION GENERALE

Le bute de ce mémoire est de faire une analyse et une modélisation pour réaliser un IDE pour l'architecture logiciel selon l'approche IASA. Cet IDE devrait permettre la réalisation des architectures proche du modèle mentale pensé par l'utilisateur.

Grace au modèle IASA un utilisateur avec des connaissances de base en informatique peut réaliser, maintenir et modifier des systèmes basé sur ce modèle sans devoir connaitre le contenu des composants qui constitue le système.

Même pour les informaticiens, ils devront être capables de réutiliser des composants développés auparavant avec quelque arrangement avec le développeur de composant.

Avec le principe de réutilisation avant de réaliser un composant on peut chercher si le composant existe auparavant ce qui va nous faire économiser des efforts et du temps si le composant existe.

La présence d'un processus de développement formalisé, bien défini et bien gérer est un facteur de réussite d'un projet [Muller 97]. Pour cette raison nous avons suivis un modèle cité auparavant, en utilisant certain diagrammes d'UML comme un langage de modélisation.

Dans l'implémentation nous avons choisi un langage qui sataisfait les besoins de la modélisation et facilite l'adaptation de ses diagrammes.

Finalement avec le test nous avons pu vérifier la richesse du modèle IASA, ainsi que la satisfaction des besoins par la modélisation et l'implémentation.

PERSPECTIVE

Dans notre IDE nous avons réalisé une interface pour dessiner des architectures selon l'approche IASA et générer un code de spécification.

Reste à améliorer :

- La qualité de dessin avec l'ajout des nouveaux forme pour les composants et les ports ce qui va améliorer la lisibilité de l'architecture.

- L'automatisation des connexions: l'utilisateur n'aura pas donc a se soucie de la disposition des composant pour dessiner les connecteurs.

Reste à ajouter :

- Un moteur de validation d'architecture : un moteur de validation d'architecture va permettre de vérifier si l'architecture dessinée est valide ou non. Ainsi ça nous évitera de générer un code non complet ou erroné.

- Un générateur de code source : le générateur de code source va nous permettre d'avoir un résultat pratique des architectures réaliser.

Bibliographie

Bibliographie

[Muller, 97] Pierre-Alain Muller, modélisation objet avec UML, EYROLLES, 1997.

[Sommerville, 88] Lan Sommerville, le génie logiciel et ses applications, paris, 1988.

[ISPS07i] Rapport interne

[cours-UML] ftp://ftp-developpez.com/laurent-audibert/Cours-UML/pdf/Cours-UML.pdf