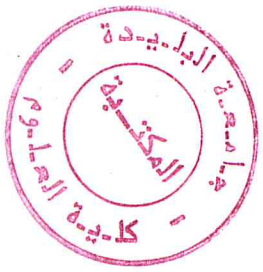
République Algérienne Démocratique et Populaire.
Ministère de l'Enseignement Supérieur et de la Recherche Scientifique.

Université Saad Dahlab, Blida
USDB.

Faculté des sciences.
Département informatique.



Mémoire pour l'obtention
d'un diplôme d'ingénieur d'état en informatique.
Option : Système d'information

Sujet :

Conception et réalisation d'un IDE pour l'architecture logicielle

Présenté par : Hadroug Ali
Hallah Mohamed Amine

Promoteur : Mr. Djamel Bennouar

Organisme d'accueil : Laboratoire LRDSI, Université Saad Dahlab, Blida

Soutenue le: 02/10/2005 , devant le jury composé de :

Mlle Aoussat	**Président**
Mlle Boussetia	**Examinateur**
Mme Sellali	**Examinateur**

Promotion : 2004/2005

DÉDICACES

A ma mère, mon père, mes frères et mes sœurs,
A toute ma famille,
A tous mes camarades et amis, en particulier
Mohamed Ali, Amar, AbdAllah, Rachid et Mahfoud
A mon oncle Youssef, sa femme, et ses fils.
A mon binôme Mohamed Amine

Je dédie cet humble travail
HADROUG ALI

A mes très chers parents, à mes frères et sœurs,
A ma grande famille, en particulier kamel et monsif
A tous mes camarades et amis,
A mon binôme Ali

Je dédie ce humble travail
HALLAH *MOHAMED AMINE*

Résumé

MCC (Modèle en couches de connecteurs) est un modèle conçu pour une méthodologie de conception des systèmes par assemblage de composants logiciels et qui permet la prise en charge directe des modèles mentaux des architectes des logiciels.
Ce projet se veut une contribution à la réalisation d'une interface graphique qui aide l'architecte à créer, valider ses applications (configurations) et y générer un code des connecteurs correspondant.
Ce modèle d'interconnexion est chargé de l'échange de deux éléments fondamentaux : les données et le flux de contrôles (cet aspect ne fait pas objet de notre réalisation).

Mots clés :
Architecture logicielle, Composant, connecteur, configurations, modèle en couches de connecteur.

Abstract

LCM (layered connector model) is conceived for a methodology of software components gathering systems design that enables to take in charge directly architects software mental models.

This project is regarded as a contribution to carry out a graphic interface that helps the architect to create, validate his applications (configurations) and to generate the corresponding connectors' code.
This interconnection model is in charge of exchanging two main elements: data & flux checking (this is not the subject of our realization).

Key words:
Software architecture, Component, connector, Configuration, layered connector model.

REMERCIEMENTS

Ce travail n'aurait pas vu le jour sans aide de ALLAH.

Nous tenons à remercier notre promoteur Mr D.BENNOUAR pour avoir proposé ce sujet et dirigé notre travail ainsi pour la marque de confiance qu'il nous a manifesté.
Nous le remercions aussi pour leurs conseils et leur soutien qui ont entretenu notre motivation, sans oublier tous et toutes, qui, de près ou de loin nous ont apporté de l'aide durant ce travail.
A tous, un grand merci.

Table des matières

Liste des figures

Liste des Tableau

Introduction générale

I. PRESENTATION DU CONTEXTE

L'un des buts de tout développeur informatique est de fournir des outils de qualité, c'est-à-dire rapide, robuste, performant, fiable et efficace. Ces qualités ne sont pas toujours compatibles, c'est pourquoi le développeur doit tant que possible d'y arriver.
Lorsque l'on développe une application coopérative (basée sur les interactions), on est en face de plusieurs variabilités des besoins en matière de propriétés attendues des interactions de communications.
L'apparition de notion architecture logicielle nous permit de réduire les coûts et les délais de développement des applications, elle modélise un système logiciel en termes de composants et d'interactions entre composants.
Dans ce contexte, le besoin est grand pour montrer les outils d'aide à la réalisation de ces interactions, ainsi on aura l'interaction des composants d'un système complexe, par le bais des connecteurs.
Ils sont des entités très importantes de modélisation. Ils ne sont malheureusement pas souvent pris en charge par les modèles de composants conventionnels. les recherches dans le domaine des architectures logicielles n'y ont pas suffisamment mis le point .
Le concept connecteur est un des aspects qui nécessite une compréhension approfondie, des éclaircissements et une maîtrise.
Dans l'élaboration d'une architecture, l'architecte concentre de manière intense son génie sur la détermination des composantes de base qu'il combine ou configure selon une topologie bien précise permettant d'atteindre les objectifs de son système. Il relègue au second plan les interconnexions qu'il définit vaguement et essaie durant le processus de conception de les disperser au niveau des composants. Un des inconvénients de cette démarche est l'incapacité de réutiliser l'infrastructure d'interconnexions établie et dispersée au sein des composants.
D'où la Nécessite d'un modèle de connecteurs qui nous permet d'unir les points de vues sur les connecteurs, et ça ne se réalise qu'avec une démarche bien précise.
Alors on définit le modèle de connecteurs comme un outil qui permet de générer les connecteurs aux différentes catégories (simple ou complexe, point à point ou multipoints).
Le présent projet, qui s'appuie sur la notion du connecteur et de connectivité, vise à offrir aux développeurs (architectes) des outils et des techniques nécessaires pour produire des architectures fiables des systèmes informatiques plus ou moins complexes.

II. PRESENTATION DU SUJET

II.1. L'existant

Il existe certains logiciels qui se concentrent sur la conception et l'évaluation d'architecture de logiciel, comme AcmeStudio.
AcmeStudio est un outil graphique qui s'utilise [17] pour gérer des conceptions architecturales de logiciel basées sur le langage de description architectural d'Acme (ADL).
Bien qu'efficaces dans leur domaine d'application, ces logiciels n'ont pris assez de Considération sauf:

- Les connecteurs faisant partie intégrante des composants. Cette approche est à l'encontre d'un objectif très important du génie logiciel à savoir la réutilisation.

- Au niveau implémentation dans les styles architecturaux basés sur les objets distribués tels les EJB, CCM, ou Microsoft.NET.

On ne veut pas essayer de créer un éditeur graphique permettant la conception architecturale de tous genres de systèmes, mais de développer une interface graphique simple, intuitive et surtout évolutive.
Elle se devait d'être 'générique' c'est à dire indépendante d'un langage particulier et capable de proposer à son utilisateur les seules fonctionnalités décidées préalablement par le travail du méta développeur.
Ce nouveau concept ne permet pas seulement de faire une programmation plus rapide et plus propre au code, mais il offre aussi plus de souplesse à la réutilisation des composants et des connecteurs par rapport aux éditeurs graphiques classiques.

II.2. Développement et méta-développement

Au notre niveau le terme de 'méta-développeur' désigne le développeur non pas d'une application mais d'une architecture du système plus ou moins complexe, avec ses propres spécificités.
L'application que nous avons développée est appelée MCC-CASIC (**M**odèle en **C**ouches de **C**onnecteurs pour la **C**onception par **A**ssemblage des **S**ystèmes **I**nformatiques **C**omplexes), faisant référence aux langages de description d'architectures dont elles conservent le formalisme graphique.
MCC-CASIC est destinée, quant à elle, au méta-développement. Cette dernière va permettre au méta-développeur, par l'intermédiaire de menus proposant des choix multiples, de développer une interface graphique personnalisée, c'est à dire, offrant des caractéristiques représentant les choix du méta développeur en termes de concepts de modèle étudié (chapitre le modèle de connecteurs).
Ainsi le méta développeur décrit sa propre architecture du système grâce à l'outil MCC-CASIC.
Par la suite tous ses choix seront enregistrés et classés dans un fichier selon une convention particulier appelée *langage de spécification.*
Il ne reste plus au programmeur qu'à utiliser l'interface pour le développement de son système.
On voit donc bien ici que le méta développeur offre au développeur une application puissante pour obtenir au final un outil lui aider a adapté les configurations des systèmes voulus.

III. PROBLEMATIQUE ET OBJECTIFS

III.1. Problématique

III.1.1. Introduction

La plupart des langages de description ou de programmation de composants, lors de la phase d'implémentation, n'offrent aucun moyen d'exprimer les connecteurs de façon explicite. En général, soit-ils introduisent des types de connecteurs prédéfinis (si toutefois ils les proposent), soit les connecteurs sont programmés au sein des composants, et même parfois au sein des programmes d'application. En tout cas, ces langages ne fournissent pas de mécanismes permettant à l'utilisateur de définir de nouveaux types de connecteurs avec des sémantiques spécifiques à ses applications.
Notre vue consiste à définir un modèle pour modéliser et décrire les architectures logicielles de systèmes à base de composants, dans lesquelles les connecteurs sont définis comme des entités de première classe.
Cette approche permet aux connecteurs un niveau de granularité et de réutilisation semblable à celui des composants.

D'où un architecte sera alors accrue si celui-ci dispose d'une bibliothèque de techniques d'interconnexions et de connecteurs. Avec une telle facilité, l'architecte peut définir et évaluer rapidement différentes architectures dont l'une peut être élue comme le point de départ pour un processus de réalisation.

III.1.2. La formulation du problème

« Connecteurs comme entités de première classe »

La première étape importante dans la compréhension de comment faire intégrer les composants, est de permettre *la description et l'analyse explicites des types de connecteur*, En rendant des abstractions de connecteur explicites. On peut faire des choix principaux de conceptions entre différents arrangements d'interaction, supportent l'analyse de ces interactions, et, dans certains cas, produisent automatiquement des réalisations pour ces interactions.

Un certain nombre de langages de description d'architecture ont donc pris cette approche, y compris Wright, UniCon, et d'autre. Ces langages permettent au créateur de choisir parmi une sélection riche des types de connecteur, et pour indiquer de nouveaux genres de types de connecteur.

Malheureusement, en tout de ces langages, les connecteurs sont si comme primitifs, ou doivent être définis à partir de zéro. Par exemple, dans UniCon, alors qu'on peut ajouter de nouveaux connecteurs au système, la tâche de faire ainsi exige la connaissance détaillée des outils et des représentations architecturaux. Dans la pratique, ceci le rend difficile de présenter de nouveaux types de connecteur, et limite notre capacité de comprendre des rapports entre différents types de connecteur. Par exemple, on pourrait aimer aux quels contextes un connecteur partagé de mémoire tampon est équivalent a une pipe.

III.1.3. Cycle de vie de connecteur

Le cycle de vie de connecteur diffère sensiblement forme le cycle de vie composant. Il peut être visualisé comme ordre du temps de conception, temps d'instanciation, temps de déploiement et de génération, et phases d'exécution.

- **Conception de connecteur**

  Le connecteur est indiqué comme descripteur dans l'ADL. Pour chacune de ses types primitifs d'élément, d'un cahier des charges fonctionnel et de la définition des traces correspondants (au moins un) doivent être fournis. Puisque les connecteurs sont des entités en soi distribuées, l'architecture de connecteur est divisées en nombre de disjoignent *des unités de déploiement*. Une unité de déploiement est constituée par les instances de rôle et les éléments internes ont conçu pour partager le même dock de déploiement.

- **Instanciation de connecteur**

  Le connecteur est instancier dans les applications. Puisque les types réels d'interface des entités attachées par l'instance de connecteur deviennent connus en ce moment, les paramètres de type d'interface des rôles du connecteur peuvent être résolus. En outre le besoin réel de certaines de ces éléments primitifs indiqués en tant que facultatif au temps de conception (par exemple interface des adaptateurs). Une partie de l'exemple de connecteur demeure générique - en raison des paramètres non définis de propriété ont associé à un futur déploiement du connecteur.

- **Déploiement et génération de connecteur**

Des connecteurs sont déployés en même temps que les interactions des composants dont ils transportent ces données. A chacune des unités de déploiement du connecteur, un dock spécifique de déploiement est assigné. Pour un connecteur d'architecture simple, les docks réels de déploiement des unités de déploiement du connecteur peuvent être impliquées des emplacements des composants interconnectés par le connecteur. Le déploiement du potentiel des composants internes d'un connecteur est indiqué de la même manière que le déploiement des composants " ordinaires " dans l'application.
Une fois le déploiement d'un connecteur est connu, le code d'implémentation de connecteur est généré (semi automatiquement) pour utiliser les primitifs de transmission ont offert par le dock de déploiement. Notez que le code générer des éléments primitifs l'un ou l'autre suit leur tracer à l'environnement de programmation fondamental, ou il peut être nulle (par exemple, aucun besoin d'adaptateur). Notez que les connecteurs avec le primitif des architectures sont considérés pour la génération de code tandis que les connecteurs des architectures composées sont créés par la composition de leurs éléments internes.
Un scénario typique de la génération de code d'un connecteur est comme suit:
(1) Utilisation d'un outil de déploiement, le déploiement des composants (l'interaction de ce que le connecteur transporte) est indiqué.
(2) de chacun de déploiement choisi des docks est alors invités pour produire automatiquement du code de mise en place de ces éléments internes du connecteur qui sont destinés pour être déployés dans celui-ci.
(3) le dock de déploiement répond la liste de technologies offertes par son fondamental environnement sur lequel la mise en place produite pourrait être basée (la liste retournée peut être vide).
(4) toutes les listes retournées sont examinées par outil de déploiement afin de trouver une allumette en technologies offertes.
(5a) si a la technologie assortie existe, les docks de déploiement sont invitées à produire du code de la mise en place du connecteur pour la technologie.
(5b) Si n'aucun apparier la technologie existe, l'utilisateur est donné les options à l'un ou l'autre changement de déploiement de l'application, ou pour fournir la mise en place du connecteur manuellement.

III.2. Objectifs

Notre objectif principal est de fournir un outil d'aide à la conception par assemblage des systèmes complexes à travers la structuration en architecture logicielle.
C'est un outil informatique convivial et interactif permettant au architecte en construction architectural des systèmes d'accomplir les deux taches :

- Construire un diagramme de configuration, contient les composants de système conçue.
- Passage à la validation du système.

Il s'agit donc de réaliser:

- Un éditeur graphique pour la création et l'édition des configurations des systèmes, en exprime touts les concepts du modèle en couches de connecteurs étudié.
- En plus, cet éditeur donne la possibilité au architecte de générer le code de connectivité après la validation du système.

IV. SYNOPTIQUE

D'abord Ce mémoire ayant comme entête une introduction générale contient la problématique et un ensemble d'objectifs tracés.
il s'articule en trois parties :

- La première partie est composée de cinq chapitres :
  - Le premier chapitre aborde les deux approches par objet et par composant.
  - Le second chapitre explique la notion de réutilisation.
  - Le troisième chapitre présente un survol de l'architecture logicielle.
  - Le chapitre quatre propose l'assemblage dans l'architecture logicielle.
  - Le chapitre cinq expose les ADLs (*architecture description languages*).
- La deuxième partie est composée de deux chapitres :
  - Le premier chapitre présente les connecteurs dans le modèle COSA (*Component-Object based Software Architecture*).
  - Dans le second chapitre nous expliquons en détaille les concepts de modèle de connecteurs étudié.
- La troisième partie est composée de deux chapitres :
  - Le premier chapitre qui est la conception, où on explique les grands axes du projet, ouis on passe à la conception et modélisation UML de l'application où on se base sur les diagrammes de classes.
  - Le second chapitre présente la réalisation de l'application, elle est considérée comme l'étape final de notre projet, elle consiste à mettre en ouvre les différents modules issues de l'étude conceptuelle.
  -

Enfin on terminera notre étude, en montrant les apports de notre travail ainsi que les compléments futurs et les améliorations qu'on peut lui apporter.
Des annexes complètent ce mémoire en décrivant des outils et un glossaire qui nous ont servi de référence pour expliquer les types d'éléments de notre modèle

Partie I

Etat de l'art

Chapitre I

L'approche par objet et l'approche architecturale

La modélisation orientée objet et la description architecturale sont deux approches de description de l'architecture d'un logiciel. La modélisation objet décrit un système comme un ensemble de classes et d'associations entre les classes. Les interactions entre les classes sont décrites au niveau implémentation sous forme d'appels de procédures ou d'accès à des données partagées. La description architecturale modélise un logiciel comme un ensemble de composants qui interagissent entre eux par le biais de connecteurs. Même si la description architecturale sépare le concept de composant du concept de connecteur, ce dernier est décrit de manière explicite au niveau architecture logicielle mais implicite au niveau implémentation. Dans cet article, nous présentons une approche d'architecture logicielle qui se base sur la modélisation objet et la description architecturale où le concept de connecteur est défini comme une entité de première classe au même titre que le concept de composant. Aussi, nous définissons certains mécanismes opérationnels associés aux composants et aux connecteurs permettant leur réutilisation et leur évolution. [1]

I. INTRODUCTION

Ces dernières années, un grand intérêt est porté au domaine de l'architecture logicielle [2].
Cet intérêt est motivé principalement par la réduction des coûts et des délais de développement des applications [3]. En effet, on prend moins de temps à acheter (et donc à réutiliser) un composant que de le concevoir, le coder, le tester, le déboguer et le documenter. Une architecture logicielle modélise un système logiciel en termes de composants et d'interactions entre composants. L'architecture logicielle joue le rôle d'une passerelle entre l'expression des besoins du système logiciel et l'étape de codage du logiciel. Elle fournit une description abstraite ou un modèle du système logiciel. Les études menées dans le domaine du génie logiciel ont montré l'importance de la notion d'architecture logicielle. Cette dernière permet d'exposer de manière compréhensible et synthétique la complexité d'un système logiciel et de faciliter l'assemblage de composants logiciels.
Pour décrire l'architecture d'un logiciel et modéliser ainsi les interactions entre les composants d'un système logiciel, deux principales approches ont vu le jour depuis quelques années : la description architecturale et la modélisation orientée objet [4].
L'approche de description architecturale, qui a émergé au sein de la communauté de recherche « Architectures logicielles », a permis notamment de prendre en compte les descriptions de haut niveau de systèmes complexes, et de raisonner sur leurs propriétés à un haut niveau d'abstraction (protocole d'interaction, conformité avec d'autres architectures, ...). Cette approche décrit un système logiciel comme un ensemble de composants qui interagissent entre eux par le biais de connecteurs. Elle décrit aussi explicitement les composants et définit des mécanismes opérationnels permettant leur réutilisation. Cependant, les fonctions des connecteurs sont implicitement cachées à l'intérieur des composants. La description implicite des interactions entre les composants rend difficile la construction de composants hétérogènes fournissant des fonctionnalités complexes et englobant des relations complexes. Pour qu'un système logiciel fonctionne correctement, il est impératif que les interactions entre ses composants soient considérées de la même manière que les composants eux-mêmes. Malgré cela, la communauté scientifique s'intéressant aux architectures logicielles ne définit pas de manière explicite la nature exacte d'un connecteur. Les connecteurs sont souvent considérés comme explicite au niveau architecture logicielle mais plutôt implicite au niveau implémentation du système [5]. Notons que certains langages de description d'architectures modélisent les connecteurs comme des composants (cf. la notion de composant-connexion dans le langage Rapide [6]).

L'approche orientée objet est devenue un standard de description d'un système logiciel durant les dernières années. Avec l'unification des méthodes de développement objet sous le langage UML [7], cette approche est largement utilisée et bien appréciée dans le monde industriel. Dans la modélisation orientée objet, les relations sont principalement implémentées par des appels de procédures (invocation de fonctions) et des accès à des données partagées. Pour prendre en compte l'aspect communication d'un système complexe ayant des composants hétérogènes (langages différents, plates formes différentes), les appels de procédures et les accès aux données partagées ne sont plus adéquats. Ainsi, des relations complexes et composites sont nécessaires pour prendre en compte les interactions complexes et pour résoudre les problèmes d'incompatibilités entre les langages et entre les plates formes.

Enfin, il est à noter que les approches « modélisation orientée objet » et «description architecturale » sont dites respectivement « Architecture Logicielle à base d'Objets » et « Architecture logicielle à base de composants » [8] [9].

II. LA PROGRAMMATION ORIENTEE COMPOSANT

Depuis plus de 40 ans, la réutilisation est au centre des préoccupations de l'ingénierie logicielle. En effet, il n'est plus envisageable de concevoir un système. Parallèlement, on se tourne vers des concepts au pouvoir d'expression de plus en plus élevé. Ainsi, les mécanismes de la programmation orientée objet (OOP) à ses débuts ont constitué un premier élément de réponse à ces besoins.
Cependant, malgré de nombreux apports du «tout objet», on s'est rendu compte rapidement que les problèmes de composition et d'évolution n'étaient pas si évidents. En effet l'approche objet souffre de limitations importantes en termes de lisibilité de l'application et de clarté de l'assemblage. Aucune représentation de l'architecture n'est proposée. On est fortement limité par la connaissance des classes et types utilisés. De plus l'application, vue comme un assemblage d'objets, ne rend pas explicites les relations entre objets.
Enfin, il serait souhaitable d'avoir une séparation claire et visible des préoccupations non fonctionnelles (les propriétés extérieures au comportement métier de l'entité), comme par exemple la gestion de la persistance et de la sécurité qui sont souvent mêlées au code fonctionnel.
Toutes ces limitations ont conduit à l'élaboration d'une approche permettant une véritable composition d'entités indépendantes et une vision claire des dépendances entre ces unités : la programmation par assemblage de composants.

II.1 Composant : représentation étendue d'objets

Le composant est la brique de base de la programmation orientée composant. C. Szyperski le définit de la façon suivante : *«A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties. »* [10]
D'une manière générale on peut dire qu'un composant est un élément ou une entité vu comme une boîte blanche ou noire (selon que le contenu est rendu visible ou non depuis l'extérieur) qui étend la notion d'objet. Il est doté de points d'entrée et de sortie (des interfaces) qui lui permet d'échanger des messages ou des références vers d'autres composants. Ces points d'accès représentent les services offerts et requis par le composant.
Un composant doit être réutilisable et peut être déployé de manière indépendante, dans différentes applications ou contextes d'application. Il est à noter qu'il existe de nombreux modèles à composants industriels et académiques (EJB, CCM, Fractal) et que chacun introduit ses propres définitions, mais que pratiquement toutes s'accordent sur les quelques principes que nous venons d'énoncer.

II.2. Programmation orientée composant : assemblage de briques.

La programmation orientée composant repose sur l'intégration de différentes entités logicielles [10]. Cette intégration se réalise à travers deux phases distinctes.
La première consiste à avoir une vue architecturale générale du système complet et la seconde à écrire ou réutiliser des composants conformes à cette vue. Tout un travail de spécification des interfaces des composants (les points d'accès aux composants), des liaisons entre les composants, de l'encapsulation de composants

dans d'autres composants que l'on appelle composant-composites ou composites, est alors à la charge de l'architecte de l'application. Finalement les développeurs auront à charge de réaliser les implémentations des composants en tenant compte de ces spécifications.
On sent ici dans le rôle de l'architecte d'applications orientées composant le besoin d'avoir une description claire et précise de l'architecture logicielle. La section suivante décrit les ADL (Architecture Description Language) qui ont vu le jour bien avant la notion même de composant, mais dont on perçoit l'apport évident pour une description abstraite et conviviale de systèmes complexes. C'est donc tout à fait naturellement que les deux approches ont récemment convergé.

III. SIMILARITES ET DIFFERENCES ENTRE LES DEUX APPROCHES

La modélisation orientée objet et la description architecturale ont plusieurs points communs [8] [9]. Les deux approches se basent sur les mêmes concepts qui sont l'abstraction et les interactions entre les entités. Dans la description architecturale, les composants et les connecteurs sont les principaux concepts pour décrire un système où les composants sont des abstractions de modules et les connecteurs sont des descriptions de la communication et des interactions entre ces composants. Dans les systèmes orientés objets, les classes sont des abstractions de données et les associations permettent de décrire les relations et les communications entre les classes et les objets.

En terme d'architecture logicielle en général, la similarité entre les deux domaines est évidente. En terme d'intention, les deux approches ont pour but de réduire les coûts de développement d'applications et d'augmenter la production de lignes de codes [2] puisqu'elles permettent la réutilisation et la programmation à base de composants. Enfin, les deux approches se focalisent plutôt sur les éléments architecturaux de grosse granularité et leur structure d'interaction que sur les lignes du code des programmes.

Ainsi, tenant compte de ces similarités, nous pouvons nous poser la question de connaître les différences entre ces deux approches. Nous répondons à cette question en présentant les avantages et les limites qui nous semblent être les plus pertinents de chaque approche.

IV. Conclusion

Ces deux approches présentent l'avenir des systèmes informatiques en tant que les deux concepts (objet et composant) représentent la base de ces systèmes.

Bibliographie

[1] T. Khammaci, A.Smeda et M. Oussalah : « ALBACO : Une architecture logicielle à base de composants et d'objets pour la description de systèmes complexes », LINA – FRE CNRS 2729 - Université de Nantes, France.

[2] D. Perry and A. Wolf: « Foundations for the Study of Software Architectures », ACM SIGSOFT Software Engineering Notes, 17(4), (1992), pp. 40-52

[3] Ofta: «Architecture de logiciels et réutilisation de composants », Collection ARAGO, No 24, Editions TEC & DOC, Paris, octobre 2000.

[4] T. Khammaci, A. Smeda et M. Oussalah: «Object-Oriented Modeling and Architectural Description: How Could they Co-exist? », Submitted paper.

[5] N. Medvidovic, R. N. Taylor: « A Classification and Comparison Framework for Software Architecture Description Languages », *IEEE Transactions on Software Engineering,* Vol. 26, 2000, pp. 70-39.

[6] D.C. Luckham, L. M. Augustin, J. J Kenny, J. Vera, D. Bryan, W. Mann: « Specification and analysis of system architecture using Rapide », *IEEE Transactions on Software Engineering,* Vol. 21, no. 4, April 1995, pp. 336-355.

[7] G. Booch: « *Object Oriented Design with applications* », The Benjamin/Cummings Publishing Company, Redwood City, California, 1991.

[8] D. Garlan: « Software Architecture and Object-Oriented Systems », Proceedings of the Information Processing Society of Japan (IPSJ), Object-Oriented Symposium 2000, Tokyo, Japan, (2000).

[9] N. Medvidovic, D.S. Rosenblum, J.E. Robbins, D.F. Redmiles: « Modeling Software Architecture in the Unified Modeling languages », ACM Trabsacations on Software Engineering and Methodology, Vol. 11, N° 1, January 2002, pp. 2-57.

[10] Clemens Szyperski: « *Component Software: Beyond Object-Oriented Programming* », Addison-Wesley Longman Publishing Co., Inc., 2002.

Chapitre II

La réutilisation logicielle

Ce chapitre se fixe comme objectif de faire l'inventaire de tous les problèmes à prendre en compte si l'on veut réutiliser et de proposer des pistes pour les résoudre. Ce n'est pas un recueil de recettes directement applicables pour faire réutilisable, c'est un recensement d'idées.

C'est à chacun de voir, selon son contexte, comment adapter ce qui est proposé pour arriver aux meilleurs résultats [1].

I. INTRODUCTION

La réutilisation est perçue depuis longtemps comme un moyen d'améliorer la qualité et de diminuer les coûts et les délais dans la production de logiciels.
Elle se présente en deux activités :

- Développement des systèmes à partir de composants réutilisables.
- Développement de composants réutilisables.

II. DEFINITION DES PRINCIPES DE BASE

II.1. Définition de la réutilisation

La réutilisation consiste à développer du logiciel en partant systématiquement d'un stock de briques de construction, afin d'exploiter les architectures et besoins communs à plusieurs applications. La réutilisation a pour objectif l'amélioration de la qualité et de la productivité, l'industrialisation de la production de logiciel [2].
La définition ci-dessus est basée sur quatre caractéristiques clés de la réutilisation du logiciel.

II.2. Caractéristiques de la réutilisation

II.2.1. La réutilisation est une pratique systématique

La réutilisation du logiciel comporte un vaste éventail d'interprétations possibles. D'une part, ont expliqué une grande partie du progrès réalisé ces cinquante dernières années dans les pratiques logicielles par l'accroissement du niveau de la réutilisation. D'autre part, des approches de réutilisation évoluées et sophistiquées existent, à la pointe des techniques, entre la recherche et la pratique courante [2].
La réutilisation systématique du logiciel signifie :

- comprendre comment la réutilisation peut contribuer à atteindre les objectifs de l'entreprise.
- Définir une stratégie au niveau technique et organisationnel de manière à obtenir le maximum de bénéfices de la réutilisation.
- Intégrer la réutilisation dans le processus complet de production du logiciel ainsi que dans les programmes d'amélioration de ce processus.
- S'assurer que toutes les équipes concernées ont les compétences et la motivation nécessaires.
- Créer un support organisationnel, technique et budgétaire approprié.
- Utiliser les métriques adaptées à l'évaluation de rendement de la réutilisation.

II.2.2. La réutilisation met en oeuvre un stock de briques de construction

le terme " brique de construction "est employé dans la définition, car les propriétés des briques et la façon dont elles sont utilisées sont parfaitement connues, et véhiculent très bien les concepts de la réutilisation.
Soient quelques propriétés importantes des briques de construction.

- Les briques de construction peuvent être assemblées pour former des éléments plus grands.
- elles peuvent ou non avoir été conçues pour être utilisées comme des briques

de construction.

- elles peuvent ou non avoir été conçues pour s'ajuster d'une manière standard.
- il peut exister des grandes briques qui définissent une architecture partielle dans laquelle de plus petites peuvent s'ajuster.
- les brique de construction peuvent être regrouper en sous ensembles, de façon à ce que ceux-ci puissent être incorporés dans différents produits finis.

II.2.3. La réutilisation exploite les similitudes entre applications, au niveau des exigences fonctionnelles et techniques

Une application est un ensemble d'un ou plusieurs programmes fournissant un ensemble de fonctions à des utilisateurs [2], afin de supporter certaines de leurs tâches, alors les applications intègrent donc des éléments communs afin de répondre à des exigences récurrentes.

II.2.4. La réutilisation peut apporter des bénéfices importants en terme de productivité, de qualité et de performance de l'entreprise.

La réutilisation peut générer de la valeur selon trois axes d'améliorations : la productivité, la qualité et la performance de l'entreprise.
Les améliorations de productivité sont obtenues essentiellement parce que la réutilisation signifie moins d'efforts en développant moins. Ainsi, augmenter la productivité en réduisant les efforts permet en retour de réduire les coûts de développement et de rapprocher la date de livraison. Cela peut aussi augmenter niveau de qualité [2].

III. LES TECHNOLOGIES POUR LA REUTILISATION

Plusieurs techniques ont été proposées afin de faciliter la réutilisation de composants logiciels, d'architectures, et même de l'expérience des concepteurs dans la résolution de problèmes spécifiques [2]. Toutes ces techniques, qui se recouvrent parfois, proposent des points de vue et de solutions partielles aux problèmes de l'industrialisation du développement logiciel. Chacune ne porte que sur un sous-ensemble des éléments et des phases du cycle de vie du logiciel. Nous allons voir leurs contributions et leur place dans ce cycle de vie.

III.1. La modélisation et la programmation objet

Les techniques objet fournissent des méthodes et des mécanismes permettant de structurer les systèmes selon les concepts manipulés dans le contexte du problème. Une application n'est plus considérée comme une fonction complexe qui doit être décomposée en sous-programmes par une approche descendante. Le concepteur identifiera plutôt les entités majeures, plus stables que des fonctions, du domaine d'application, leurs responsabilités et leurs relations.

L'analyse objet permet de spécifier la structure et le comportement d'un système, en appliquant des techniques objet. De telles méthodes d'analyse fournissent des langages graphiques de modélisation ainsi que des démarches pour les mettre en œuvre.

La conception objet va transformer le modèle d'analyse en un modèle de conception respectant les contraintes d'architecture et d'implémentation.
Les principales méthodes d'analyse et de conception objet sont Booch et OMT, et le formalisme UML qui est adopté comme langage standard pour la modélisation objet.
La programmation objet est basée sur le concept d'objet comme unité de programme

encapsulant à la fois les données et les algorithmes. Elle permet la réutilisation sous la forme de bibliothèques de classes au travers d'un ensemble de techniques telles que la composition, l'héritage et le polymorphisme.

Le développeur construit une application en réutilisant les bibliothèques de classes de deux manières différentes. L'une consiste à construire de nouvelles classes à partir de classes existantes (composition), fournies par la bibliothèque.

III.2. Les patterns

Les patterns de conception sont apparus récemment dans le monde orienté objet comme une technique de documentation de solutions standard de conception. La définition la plus générale d'un pattern décrit le contexte d'application du pattern, le problème de conception qui est traité et une solution validée par l'expérience.
Un pattern documente un savoir-faire réutilisable, concernant un problème récurrent.
Actuellement, plusieurs bibliothèques de patterns ont déjà été élaborées. Dans ce cas, le terme « bibliothèque » désigne un document qui décrit un ou plusieurs patterns de conception.
Les patterns s'appliquent à des domaines allant de l'analyse et de l'architecture générale jusqu'aux détails très précis de la conception ou de l'implémentation. Un assemblage structuré de design patterns pour un domaine d'application spécifique se nomme langage de pattern. La structure de langage de pattern est représentée par les dépendances entre les patterns.
Malheureusement, les patterns ne permettent pas la réutilisation du code de façon systématique, contrairement aux bibliothèques de classes. En général, une bibliothèque de classes pourra difficilement s'adapter aux solutions apportées par les langages de patterns.

III.3. Les Frameworks

Un framework [2] est un ensemble intégré de produits logiciels réutilisables, extensibles et personnalisables pour un domaine d'application spécifique. Un framework objet est constitué d'un ensemble de classes et de relations décrivant des structures d'objets coopérants.
Des applications concrètes sont générées en adaptant les éléments variables du framework.
Par rapport à la majorité des techniques de réutilisation, telles que les bibliothèques de classes et les design patterns, un framework comporte à la fois du code et une architecture réutilisable.

III.4. Le développement à base de composants

Le développement à base de composants [2] consiste à construire des applications en assemblant des composants logiciels existants. Les composants sont des unités de programmes disponibles sous forme binaire. Ils sont réutilisables comme des boites noires en proposant une interface prédéfinie. La granularité des composants peut varier de petits objets graphiques d'interface utilisateur jusqu'à des applications complètes.
Un composant diffère d'un objet, principalement par le fait qu'il est dissocié de son entourage :

Un objet a généralement besoin d'autres objets pour fonctionner (c'est ce que le rend difficilement réutilisable en dehors de son contexte), tandis qu'un composant est constitué d'un groupe d'objets.

Cela signifie qu'un composant fournit un ensemble de services à une application cliente, mais n'a pas de dépendances externes. Dans certains domaines d'application (interfaces graphiques, gestion...), des composants sont déjà disponibles sous la forme de produits sur étagère.
Le développement à base de composants permet de réellement différencier le développement de bien logiciels de leur réutilisation.

En résumé, le développeur d'application doit fournir l'architecture d'intégration d'application, alors que la conception et l'implémentation de chaque composant sont déjà fixées.

L'approche par composants améliore la production de logiciel en réduisant la quantité de code réalisée par le concepteur d'application.
Le développement par composants est une technique de réutilisation très puissante. La principale difficulté réside dans le niveau de compétence technique et l'organisation nécessaire dans une équipe projet.

III.5. Les systèmes à base d'agents

Le concept d'agent logiciel [2] a récemment été proposé comme une extension du modèle d'objet. Les agents sont des systèmes intégrés qui:

- incorporent certaines capacités essentielles extraites de domaines tels que l'intelligence artificielle, les bases de données, les langages de programmation et l'algorithmique.
- Communiquent avec d'autres agents en utilisant un langage de communication indépendant de leurs caractéristiques et de leurs structures. Il est ainsi possible de concevoir la plupart des systèmes logiciels complexes et des applications comme des organisations d'agents coopératifs.
- L'idée d'agent logiciel offre une métaphore naturelle pour la modélisation des composants d'un système et de leurs interactions.

III.6 Les architectures logicielles

Ces dernières années, il y a eu une évolution dans l'étude des architectures des systèmes d'information. Chaque type d'architecture est décrit en termes d'éléments d'architecture et de mode de communication (Protocoles de la communication, synchronisations, contraintes).
Cela a mené à l'identification de certaines architectures universelles apparaissant fréquemment dans les systèmes logiciels, telles que le client serveur, les architectures à trois niveaux, les architectures en couches (l'exemple des réseaux de télécommunications), etc.
Le développement des architectures logicielles a déplacé l'attention de la réutilisation de code vers la réutilisation de conception. Cela a été réalisé en prédéfinissant les types de relations possibles entre les éléments d'une architecture. Contrairement à l'approche des bibliothèques de classes, c'est la structure des interconnexions entre les composants qui est réutilisée, et non pas les composants eux-mêmes.
La réutilisation d'architectures logicielles est basée sur la définition de « styles d'architecture », qui décrivent des familles d'architectures partageant un ensemble de caractéristiques. Un style architectural définit un vocabulaire spécialisé pour les éléments d'architecture (par exemple : client, serveur, serveur d'application...), un vocabulaire pour les connexions entre ces éléments (par exemple : socket, pipe, flux

de données, appel synchrone, appel asynchrone, événement...) et une série de règles pour construire des topologies spécifiques en utilisant des composants et des connecteurs. Les styles architecturaux améliorent la communication entre les concepteurs, qui peuvent faire référence à une terminologie commune pour la réutilisation de solutions de conception.

IV. COMPARAISON DES TECHNIQUES DE REUTILISATION

Nous comparons ces techniques [2], en soulignant les points forts et les points de vue du développeur d'application et du développeur de biens logiciels (composants) réutilisables.

Tab II.1 : Comparaison des techniques qui permette de mettre en oeuvre la réutilisation.

	Points forts	Points faibles
Les techniques objet et bibliothèques de classes	Evolutivité améliorée grâce au masquage des données et à la classification (relation de généralisation/spécialisation).	Demande un effort considérable de modélisation. Les objets sont difficilement extractibles de leur contexte.
Design patterns	Facilitent la récupération de solutions de conception, fournissent une ligne directrice pour le processus de développement. Elèvent le paradigme de conception car ils permettent de concevoir en réutilisant des groupes d'objets prédéfinis.	Mise en application (pas de réutilisation au niveau du code).
Frameworks	Réutilisation du modèle objet et de l'architecture.	Demande une haute expertise et une compréhension approfondie du domaine.
Les composants	Packagés pour être échangés. Développement d'un marché externe.	La réutilisation boite noire est plus contraignante.
Agents logiciels	Hautement personnalisables et adaptables, facilitent la reconfiguration des systèmes complexes.	Manque de maturité et de mise en œuvre industrielle.
Architectures logicielles	Simplifient la réutilisation d'objets techniques et métier.	Trop génériques : elles doivent être spécialisées pour chaque domaine d'application.

Bibliographie

[1] B.Coulange : « réutilisation du logiciel », collection méthodes informatiques et pratique des systèmes, édition Masson, Paris, 1996.

[2] M.Erzan, M.Maurizio, C.Tully : « réutilisation logiciel », collection Informatiques magazine, édition Eyrolles, Paris, 1999.

Chapitre III

Un survol de l'architecture logicielle

L'architecture logicielle est un domaine récent du génie logiciel qui a reçu une attention particulière Ces dix dernières années. Les éditeurs de logiciels ont pris conscience qu'une architecture est un facteur critique dans la réussite du développement et facilite la maintenance et l'évolution du logiciel.

Elle contribue la maîtrise des logiciels complexes. Les langages de description d'architecture constituent l'un des deux aboutissements des travaux sur l'architecture logicielle.

Ils permettent de formaliser l'architecture logicielle, de spécifier les modules de l'architecture de manière abstraite et de définir de manière explicite les interactions entre ces modules [1].

I. INTRODUCTION

Dans les dernières années, la conception de systèmes informatiques complexes passe par la mise en œuvre d'une méthodologie centrée sur l'architecture logicielle (Figure1).

Figure III.1 : Article de compréhension de principe de base

Dans cette approche, et à un haut niveau d'abstraction, un système est considéré comme un ensemble de composants interconnectés en une configuration permettant de réaliser des fonctionnalités bien précises [2].

Actuellement, un grand intérêt est porté au domaine de l'architecture logicielle. Cet intérêt est motivé principalement à la réduction des coûts et des délais de développement des applications. En effet, emprunts moins de temps à acheter (et donc à réutiliser) un composant que de le concevoir, le coder, le tester, le déboguer et le documenter. Une architecte logicielle modélise un système logiciel en termes de composants et d'interactions entre composants. L'architecture logicielle joue le rôle d'une passerelle entre l'expression des besoins du système logiciel et net par le de codage du logiciel. Elle fournit une description abstraite ou un modèle du système logiciel. Les études menées dans le domaine de l'architecture logicielle ont montré l'importance de la notion d'architecture. Cette dernière permet d'exposer de manière compréhensible et synthétique la complexité d'un système logiciel est de faciliter l'assemblage de composants logiciels. Les axes de travail autour de cd thème sont multiples. L'idée fondamentale est de profiter des avantages d'une architecture clairement explicitée pour ne laisser en second plan que la programmation d'application.

Deux éléments essentiels sont à la base de la définition d'une architecture logicielle : le style architectural et le langage de description d'architecture.

Un style architectural décrit une architecture logicielle en s'appuyant sur les concepts de composant, de connecteur et de configuration il caractérise une famille de systèmes qui partagent des propriétés structurelles et sémantiques. Il y a fourni un vocabulaire des éléments de conception, les types des composants et les connecteurs comme les pipes, filtres, clients, serveurs. Il dessinait un ensemble de règles de configuration qui déterminent les compositions possibles de ces éléments [1].

II. NOTION D'ARCHITECTURE LOGICIELLE

Le terme « architecture logicielle » a souvent été utilisé dans deux contextes. Dans le premier contexte, c'est la structure d'un logiciel en termes de ses composantes qui est spécifiée. Ainsi, l'architecture d'un système est vue comme étant l'interconnexion de sous-systèmes fonctionnellement bien définis. Dans le deuxième contexte, l'architecture est utilisée pour désigner une structure d'exécution (ou un style) bien définie telles que l'architecture client-serveur, l'architecture en couches,

l'architecture centrée sur les données, l'architecture n-tiers, l'architecture en bus, etc.[3]

L'architecture logicielle prend son départ du modèle mental que se fait un architecte du système qu'il entreprend de réaliser, Ce modèle est souvent représenté à un haut niveau d'abstraction par un ensemble de blocs interconnectés, souvent appelé architecture du logiciel à réaliser [2].

La recherche sur les architectures logicielles et appareils à la fin des années soixante [4]. Son émergence s'explique par l'approche généralement adoptée pour la construction de logiciels complexes. La structure de ses logicielles s'appuie principalement sur des méthodes et des techniques informelles.
Typiquement, la structure d'un logiciel est présentée au moyen d'un schéma comprenant des boîtes interconnectées et des phrases explicatives comme la structure client serveur [1].

Dans Le modèle en blocs et lignes d'interconnexions, l'architecte définit l'objectif de chaque bloc (souvent c'est la fonction) ainsi que la sémantique des interconnexions. Ensuite, les blocs et les interconnexions sont raffinés et l'architecture recentrée dans un processus d'analyse et de conception itératif et incrémental.
L'architecte raisonne dans plusieurs domaines de conception, tels que le domaine de l'organisation, la gestion du code source et le domaine de déploiement du logiciel.

Dans le domaine de l'organisation du code source par exemple, l'architecte définit les différents modules et librairies ainsi que les relations de dépendances entre ces modules pour la construction des différents modules exécutables [2].

II.1. Définition de l'architecture logicielle

Le terme « architecture logicielle » ne possède pas une définition universelle ayant reçue l'unanimité de toute la communauté scientifique travaillant dans ce domaine, Plusieurs définitions ont été proposées dans la littérature. Ces définitions se basent sur plusieurs critères tels que la structure des composants, la représentation abstraite de l'architecture logicielle, le processus de construction d'une architecture et enfin le style architectural est adopté pour la construction du logiciel.
Une définition proposée par **Medvidovic** est la suivante :

> ***«L'architecture logicielle est un niveau de conception qui comprend la description des éléments à partir desquels un système est construit, les interactions entre ces éléments, les configurations qui guident leur composition et les contraintes définies dans ces configurations.»***

L'architecture logicielle en tant que discipline du développement génie logiciel adresse toutes les étapes du cycle de conception de logiciels, définit rigoureusement et sans ambiguïtés ses éléments fondamentaux (une sorte de méta-modèle) qui seront mis en œuvre dans le processus de construction de logiciel, définit les différentes vues qui seraient utilisées dans un cycle de vie de logiciel et les mécanismes assurant la cohérence entre les états du système décrits dans les différentes vues (ou domaine de conception), supporte la réutilisation des composants et de l'architecture, et enfin offre les outils et les méthodes prenant en charge un processus de développement de système logiciel qui débute par une définition architecturale[3].

II.2. Rôle des architectures logicielles

L'architecture logicielle joue le rôle d'une passerelle entre les étapes d'expression des besoins du système logiciel et du codage. Elle fournit une description abstraite ou un modèle du système [11]. Elle ne se focalise pas sur des problèmes liés à l'algorithmique et aux structures de données mais à de nouvelles structures qui sont au dessus du code. En général, cette représentation fournit un guide du logiciel. Ce qui permet aux concepteurs de raisonner sur la capacité du système à satisfaire certains besoins de l'application.
Par ailleurs, l'architecture permet la compréhension des grands systèmes en Les représentant à un niveau élevé d'abstraction, la réutilisation en se basant sur l'établissement des bibliothèques des composants, la construction de logiciel en fournissant un modèle partiel du développement par indication des composants majeurs et de leurs connexions, l'analyse pour vérifier la conformité des contraintes imposées par un style architectural [5], la communication et la structure architecturale.
Enfin, l'architecture logicielle à elle seule ne suffit pas à décrire des systèmes. La description d'une architecture est réalisée par plusieurs structures ou vues architecturales.

II.3. Spécification de l'architecture logicielle

La définition de l'architecture est l'action principale dans les premières phases d'un cycle de développement de logiciels. Elle peut être réalisée selon les deux types approches indiquées précédemment : une approche basée sur le modèle objet (UML) ou une approche basée sur le modèle à composants (ADL).
Dans le monde objet, la description architecturale est réalisée à l'aide du langage UML, un langage très largement accepté, tant au niveau des laboratoires de recherche qu'au niveau industriel. Dans le monde des modèles à composants, qui est le noyau de notre sujet la description est réalisée par des langages de description d'architectures ADL (sujet du prochain chapitre).

III. CONCEPTS DE BASE DE L'ARCHITECTURE LOGICIELLE

Le *composant*, le *connecteur* et la *configuration* représentent les éléments fondamentaux de spécification d'architecture logicielle.
Les composants et les connecteurs sont décrits par les aspects suivants: l'interface, le type, la sémantique ou comportement, les contraintes d'exploitation, la possibilité d'évolution et les propriétés non fonctionnelles.
Une configuration représente une topologie qui indique la manière correcte avec laquelle chaque composant est exploité, comment les différents composants sont interconnectés, quels sont les connecteurs utilisés et comment ces connecteurs sont utilisés. Un composant peut avoir sa propre configuration indiquant comment ce composant est ou doit être réalisé.
Les aspects importants gérés au niveau configuration sont : le raffinement et la traçabilité, l'hétérogénéité, la dynamique de la taille, l'évolution, les contraintes et les propriétés non fonctionnelles.
Le *style* d'architecture et ADLs sont aussi des notions très répandue dans l'architecture logicielle. Un style est représenté par un ensemble typique de composants, de connecteurs, de propriétés, de règles de conception et de règles d'interconnexions permettant de guider et contrôler la construction d'une architecture. Le style restreint le domaine de conception et facilite l'analyse, la réutilisation, et la prise en charge par les outils des différentes phases du cycle de vie de logiciel.

Une architecture est souvent associée à un ensemble de propriétés non fonctionnelles, indiquant à titre d'exemple, les objectifs en terme de performance et de sécurité.
Dans ce qui suit, nous présentons un bref aperçu des concepts de composants, connecteurs, configuration qui sont les éléments fondamentaux de l'architecture logicielle (Figure).

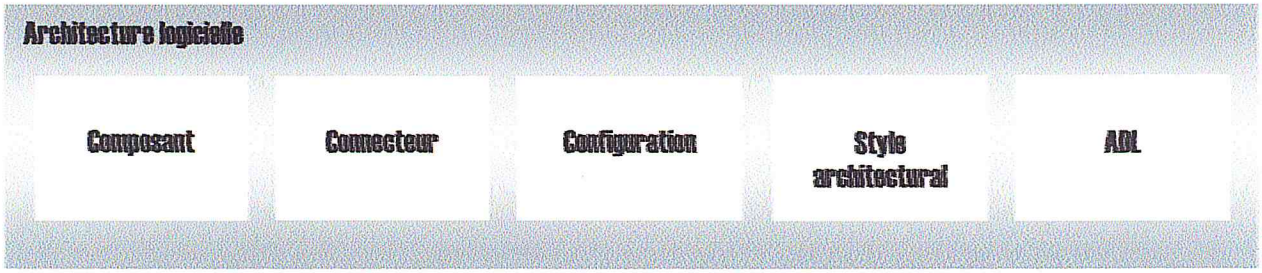


Figure III.2 : les éléments de spécification d'architecture logicielle.

III.1. Le concept composant

Un composant logiciel 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 [3].
C'est une entité de calcul ou de stockage à laquelle est associée une unité d'implantation [5].
Il interagit avec les autres composants pour réaliser un ou plusieurs objectifs d'une architecture. Un composant peut être très simple, comme une fonction C, un objet d'une classe C++ ou complexe comme un serveur HTTP ou un SGBD.
Le composant peut être un composant déjà réalisé et testé et il ne s'agit que de l'instancier et bien l'intégrer dans la configuration. Dans ce cas, il ne sera pas nécessaire de le raffiner [3].
Les **caractéristiques globales** d'un composant définies par Medvidovic et Taylor sont les suivantes:

- *L'interface* d'un composant est la description de l'ensemble des services offerts et requis par le composant sous la forme de signature de méthodes, de type d'objets envoyés et retournés, d'exceptions et de contexte d'exécution. L'interface est un moyen d'expression des liens du composant ainsi que ses contraintes avec l'extérieur [5]. Le service peut être décrit de plusieurs manières, tels que la signature de méthode ou le code d'un message. Une interface est composée d'un certain nombre de points d'interaction appelés *ports* de communication, C'est à travers cette interface qu'un composant est joignable par les autres composants afin de leur offrir ses services, et c'est à travers cette interface qu'il est connectable aux services des autres composants [5].
- Le *type* d'un composant est un concept représentant l'implantation des fonctionnalités fournies par le composant. Il s'apparente à la notion de classe que l'on trouve dans le modèle orienté objet.
  Ainsi, un type de composant permet la réutilisation d'instances de même fonctionnalité soit dans une même architecture, soit dans des architectures différentes [5].
- La *sémantique* du composant est exprimée en partie par son interface. Cependant, l'interface telle que décrite ci-dessus ne permet pas de préciser complètement le comportement du composant. La sémantique doit être

enrichie par un modèle plus complet et plus abstrait permettant de spécifier les aspects dynamiques ainsi que les contraintes liées à l'architecture [5].

- Les *contraintes* définissent les limites d'utilisation d'un composant et ses dépendances intra composants. Une contrainte est une propriété devant être obligatoirement vérifiée sur un système ou une de ces parties. Si celle-ci est violée, le système est considéré comme un système incohérent et inacceptable. Elles permettent ainsi de décrire de manière explicite les dépendances des parties internes d'un composant comme la spécification de la synchronisation entre composants d'une même application [5].
- *L'évolution* doit donc être simple et s'effectuer par le biais de techniques comme le sous typage ou le raffinement [5].
- Les *propriétés non fonctionnelles* (propriétés liées à la sécurité, la performance, la portabilité) doivent être exprimées à part, permettant ainsi une séparation dans la spécification du composant des aspects fonctionnels (aspects métiers de l'application) et des aspects non fonctionnels ou techniques (aspects transactionnel, de cryptographie, de qualité de service). Cette séparation permet la simulation du comportement d'un composant à l'exécution dès phase de conception, et de la vérification de la validité de l'architecture logicielle par rapport à l'architecture matérielle et l'environnement d'exécution.

III.2. Le concept connecteur

Le connecteur correspond à un élément d'architecture qui modélise de manière explicite les interactions entre un ou plusieurs composants en définissant les règles qui gouvernent ces interactions [5]. D'où ils sont des blocs de construction utilisés pour modéliser l'interaction entre composants [6].
Un connecteur comprend également deux parties. La première correspond à la partie visible du connecteur, c'est-à-dire son interface, la seconde partie correspond à la description de son implantation [5].
Il permet également de vérifier l'intégrité de la communication, ainsi il permet la réutilisation et l'adaptation des interfaces de composants déjà existants que l'on cherche à relier.
Medvidovic et Taylor définies des caractéristiques importantes des connecteurs sont :

- *L'interface* d'un connecteur définit les points d'interactions entre connecteurs et composants.

  L'interface ne décrit pas des services fonctionnels comme ceux du composant mais s'attache à définir des mécanismes de connexion entre composants. Certains ADLs nomment ces points d'interactions comme étant des rôles [5].
- Le *type* d'un connecteur correspond à sa définition abstraction qui reprend les mécanismes de communication entre composants ou les mécanismes de décision de coordination et de médiation. Il permet la description d'interactions simples ou complexes de manière générique et offre ainsi des possibilités de réutilisation de protocoles [5].
- La *sémantique* des connecteurs est définie par un modèle de haut niveau spécifiant le comportement du connecteur. A l'opposé de la sémantique du composant qui doit exprimer les fonctionnalités déduites des buts ou des besoins de l'application, la sémantique du connecteur doit spécifier le protocole d'interaction. De plus, celui-ci doit pouvoir être modélisé et raffiné lors du passage d'un niveau de description abstraite à un niveau d'implantation [5].

- Les *contraintes* permettent de définir les limites d'utilisation d'un connecteur, c'est-à-dire les limites d'utilisation du protocole de communication associé. Une contrainte est une propriété devant être vérifiée sur un système ou sur l'une de ses parties. Si celle-ci est violée, le système est considéré comme un système incohérent et inacceptable [5].
- Les *propriétés non fonctionnelles* d'un connecteur concernent tout ce qui ne découle pas directement de la sémantique du connecteur. Elles spécifient des besoins qui viennent s'ajouter à ceux déjà existants et qui favorisent une implantation correcte du connecteur [5].

III.3. Le concept configuration

La configuration de l'architecture, encore appelées topologies, est un graphe de composants et de connecteurs pour réaliser une architecture. Cette information est nécessaire pour déterminer si les composants et connecteurs sont composés correctement [6].

Elle définit les propriétés architecturales de connectivité et de conformité aux heuristiques de conception, ainsi que des propriétés de concurrence et de répartition.

Ainsi, doit être compréhensible et permettre une description avec différents niveaux d'abstraction avec une continuité entre les niveaux et idéalement jusqu'à la création d'un système exécutable.

Elle définit également le schéma d'instanciation des composants au moment de l'initialisation de l'application, ainsi que le placement des composants sur les sites au moment du démarrage du système et leur évolution pendant la vie de l'application.

Les caractéristiques précisées par Medvidovic et Taylor pour évaluer une configuration sont:

- un formalisme commun.
- la composition.
- le raffinement.
- l'hétérogénéité.
- le passage à l'échelle.
- l'évolution de la configuration.
- l'aspect dynamique de l'application.
- les contraintes.
- les propriétés non-fonctionnelles.

IV. CONCLUSION

L'architecture d'un système logiciel joue un rôle déterminant pour le succès d'un système. La conception d'une bonne architecture peut amener à un produit qui répond aux besoins des clients et qui est facilement modifiable pour prendre en compte de nouveaux besoins. Cependant, les architectes logiciels ne possèdent pas d'outils pour les assister dans la conception de l'architecture d'un système logiciel. Les langages de description d'architecture permettent d'aider ces architectes en décrivant l'architecture d'un système logiciel à partir de notations formelles.

Bibliographie

[1] T. Khammaci, M. Oussalah, A. Smeda, Les ADLs : une voie prometteuse pour les architectures logicielles, Actes du congrès Agents, Logiciels, Coopération, Apprentissage & Activité Humaine, (ALCAA 2003), Septembre 2003, Bayonne, France.

[2] D. Bennouar : «Vers la définition d'un modèle de connecteurs logiciels pour la description de systèmes complexes » Laboratoire LSDRI, Département d'Informatique, Faculté des Sciences, Université de Blida, 2004.

[3] D. Bennouar : « Vers une mise en œuvre conjointe des ADL et UML dans le développement de logiciels complexes » Laboratoire LSDRI, Département d'Informatique, Faculté des Sciences, Université de Blida, 2004.

[4] R.Kazman, Software Architecture, « Handbook of Software Engineering and knowledge Engineering » Vol1: Fundamentals, pp.47-67, World scientific Publishing, 2001.

[5] ACCORD, Etat de l'art sur les Langages de Description d'Architecture (ADLs), Projet ACCORD (Assemblage de composants par contrats en environnement ouvert et réparti), juin 2002.

[6] Projet SEP, « Comparaison avec les langages de description d'architectures », doc pdf.

Chapitre IV

L'assemblage dans l'architecture logicielle

L'assemblage dans l'architecture logicielle nécessite un modèle de connecteurs puissant sert à réaliser des infrastructures d'interconnexion efficaces.

I. INTRODUCTION

Nous présentons l'architecture d'assemblage selon un découpage en deux parties. Nous traitons tout d'abord de la composition d'un composant, c'est-à-dire d'un point de vue ensembliste sur les différentes entités qui constituent un composant. Suivant le point de vue UML 2.0, un composant 'composite' est composé d'entités logicielles (baptisées 'parties' à un moment de l'évolution de la norme UML 2.0). Elles réalisent le schéma de contrôle et le schéma de données du composant. Un composant est également composé de ports qui permettent la réalisation des interactions de communication.
Dans ce document nous traitons d'une part de la composition d'un assemblage (l'aspect ensembliste de l'assemblage) et d'autre part des interactions entre composants (l'aspect de communication entre composants réalisé par des connecteurs).

II. ASSEMBLAGE ET CONNECTEUR

Un composant logiciel est une unité de réutilisation et d'intégration. Il est donc conçu pour fonctionner en coopération avec d'autres composants. Pour cela, il faut réaliser l'étape de composition ou d'assemblage d'un ensemble de composants caractérisés essentiellement par leurs interfaces offertes et leurs interfaces requises.
La notion de connecteur recouvre l'ensemble des moyens nécessaires pour assurer l'interaction entre les composants par la connexion des interfaces requises et offertes. Ces moyens peuvent être vus à deux niveaux. La spécification des propriétés des interfaces requises et des interfaces offertes peut être utilisée pour définir les connecteurs sur un plan conceptuel. Dans ce premier cadre, un connecteur est vu comme la spécification d'un mode de communication. Dans un second temps, si nécessaire, le connecteur est implanté par un ensemble de moyens logiciels et matériels réels et assure concrètement l'adaptation des propriétés requises aux propriétés offertes (par exemple pour adapter le typage de certains paramètres, pour réaliser un protocole de communication). En termes de séparation des préoccupations, il est souhaitable de ne pas faire apparaître les traitements associés aux fonctions des connecteurs dans les composants mais d'en faire des entités bien distinctes.

III. ASSEMBLAGE

Un composant est destiné à être relié avec d'autres ; l'opération consistant à définir ces relations est l'assemblage. Les deux principales approches actuelles sont les langages de description d'architectures et les langages de coordination.

III.1. Langages de description d'architectures

La définition d'une architecture logicielle est une étape importante dans la conception d'un logiciel. Elle permet d'avoir un niveau d'abstraction élevé et de disposer de modèles qui s'approchent du modèle mental du développeur. Une architecture logicielle décrit l'ensemble des composants qui le composent et donne les règles de leur assemblage. Elle permet la conception d'applications en se détachant de détails techniques propres à l'environnement et en respectant les conditions fixées par les futurs utilisateurs. Elle correspond à l'établissement du plan de construction du logiciel. En maîtrisant l'architecture conceptuelle, il est alors plus facile de gérer ses éventuelles évolutions. En effet la modification d'un plan est plus simple que la modification d'un système réalisé.

III.2. Langages de coordination

Les applications actuelles profitent des systèmes parallèles et distribués pour offrir de meilleures propriétés non-fonctionnelles (passage à l'échelle, fiabilité, disponibilité, etc.). Cependant, le développement de telles applications reste complexe et la présence de multiples sites ajoute un nouveau défi : la coordination de la coopération d'un grand nombre d'activités concurrentes actives. De ce fait, la programmation d'un système parallèle ou réparti pourrait être vue comme une combinaison de deux activités distinctes : la partie calcul comprenant un nombre de processus impliqués dans la manipulation des données, et la partie coordination responsable de la partie communication et coopération entre les processus. Cette approche rend les modules plus indépendants, permet l'évolution d'applications en modifiant seulement les éléments concernés et non en intervenant sur toute l'application. Ainsi, elle permet de réutiliser d'une part les objets sans la façon dont ils sont coordonnés, et d'autre part les entités de coordination sans les objets à coordonner [1].
Cette théorie a conduit à la proposition d'un grand nombre de modèles de coordination et de leur langage de programmation associé. Avant d'en faire un survol, nous allons préciser quelques concepts de base.

IV. CONNECTEURS D'INTERACTIONS, NECESSITE DES CONNECTEURS

Lorsque l'on développe une application coopérative (basée sur les interactions), on est confronté à l'extrême variabilité des besoins en matière de propriétés attendues des interactions de communications. Deux solutions sont envisageables.
Dans l'approche la plus simple, on considère qu'il existe un seul mode d'interaction (mode primitif). Il peut s'agir d'un mode de base très simple (et monolithique) ou d'une interaction présentant des modes de fonctionnement plus complexes et différenciés (synchrone, asynchrone, en diffusion, centralisé, en appel léger, en appel distant, etc.).
Même dans ce second cas la variabilité disponible n'est en général pas suffisante. On peut souhaiter introduire de multiples propriétés concernant tant le schéma de communication (interaction multipoint, quorums) que le schéma de contrôle de l'invocation (en termes de sécurité, tolérance aux pannes, performances etc.) ou encore le schéma des données échangées. La seule solution qui reste alors pour réaliser tous ces modes, consiste à développer les adaptations nécessaires dans les codes clients et serveurs. Cette solution est mauvaise car elle est assez lourde (elle peut nécessiter des modifications en de nombreux points des codes clients et servants). Par ailleurs elle amène à modifier les codes de base client et serveur ce qui après modifications successives est très peu lisible. Les modifications effectuées sont peu réutilisables. Par ailleurs pour des environnements divers on doit développer des versions multiples du même code. Remarquons enfin que la modification des codes client et servant peut même être impossible si l'on ne dispose que d'un binaire.
Dans une approche plus élaborée, on traite le problème précédent en considérant que les différents modes d'interaction nécessaires aux composants sont réalisés au moyen d'un ensemble de logiciels ou matériels de communication, rassemblés dans la notion de connecteur.
Le connecteur est alors l'ensemble des logiciels de communication capables d'adapter les besoins définis par une spécification d'interface requise et la fourniture définie par les spécifications d'interface offerte. Le connecteur est donc le support essentiel de la séparation des propriétés fonctionnelles (réalisées par un composant) et des propriétés non fonctionnelles.

Insistons sur le point important suivant. En termes de spécifications, l'intervention de la notion de connecteur se justifie complètement dans le cas où la spécification d'un port offert (d'une opération ou d'une interface offerte par le servant) est incompatible avec celle du port requis. Le connecteur est l'ensemble des moyens matériels et logiciels qui réalisent dans ce cas, l'interaction entre les deux composants c'est-à-dire l'adaptation, indispensable au fonctionnement d'ensemble, des deux composants.

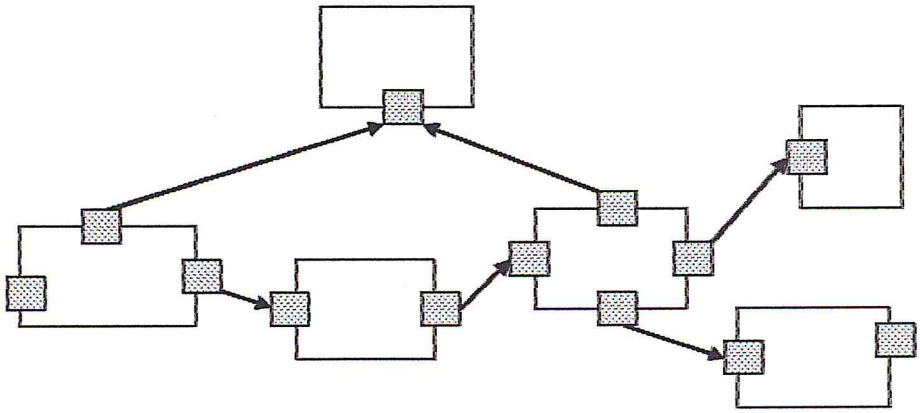


Figure IV.1 : Une Application

V. NOTION D'APPLICATION

Dans le cadre des définitions précédentes on ne dispose que de la notion d'instance de composant pour décrire une entité logicielle susceptible de s'exécuter. Or il est tout à fait envisageable de concevoir des logiciels informatiques qui sont construits comme un assemblage de composants. On évite ainsi de passer par l'étape de packaging de cet ensemble en un composant nécessitant par exemple la description dans un cadre général du déploiement des instances de ce composant.
Dans le projet, une application (Figure IV.1) est un assemblage correct d'un ensemble de composants en vue d'être déployé une seule fois (qui ne présente donc pas toutes les spécifications nécessaires lui permettant d'être déployé de façon générale).

VI. CONCLUSION

Cette étude sur l'« assemblage de composants par contrats » montre l'ampleur et la complexité du domaine. Alors on résume que la base de l'assemblage au terrain architectural signifier que : deux composants sont en fait reliés par un connecteur indispensable pour réaliser la sémantique de la communication. [7]

Bibliographie

[1] F. Arbab, G.A. Papadopoulos ; Coordination models and languages, Software Engineering (SEN), SEN-R9834 December 31, 1998.

[2] T. W. Malone and K. Crowston, The Interdisciplinary Study of Coordination, ACM Computing Surveys 26, 1994, pp. 87-119.

[3] B. Singh; Interconnected Roles (IR): A Coordinated Model. Technical Report CT–84–92, Microelectronics and Computer Technology Corp.,
Austin, TX, 1992.

[4] N. Carriero, D. Gelernter ; Linda in context. Communication of the ACM 35 (2), 1992 pp. 97-107.

[5] F. Arbab, I. Herman and, P. Spilling ; An Overview of Manifold and its Implementations. Concurrency: Practice and Experiences 5(1), 1993, pp.664-677.

[6] Manuel Günter, Explicit Connectors for Coordination of Active Objects, Master Thesis of Faculty of Science, University of Bern 1998.

[7] G.Florin: « Composants Logiciels », CNAM -Laboratoire CEDRIC-, France.

Chapitre V les langages de description d'architecture ADL

Vu le volume important des travaux de recherche, il apparaît que la discipline « architecture logicielle » tend à se doter de ses propres outils et méthodes permettant de prendre en charge les diverses étapes du processus menant à la réalisation d'un système logiciel. Parmi les outils de base de l'architecture logicielle, il y a les langages de description d'architecture ou ADL [1].

I. INTRODUCTION

La définition d'une architecture logicielle est une étape importante dans la conception d'un logiciel. Elle permet d'avoir un niveau d'abstraction élevé et de disposer de modèles qui s'approchent du modèle mental du développeur. Une architecture logicielle décrit l'ensemble des composants qui le composent et donne les règles de leur assemblage. Elle permet la conception d'applications en se détachant de détails techniques propres à l'environnement et en respectant les conditions fixées par les futurs utilisateurs. Elle correspond à l'établissement du plan de construction du logiciel.
En maîtrisant l'architecture conceptuelle, il est alors plus facile de gérer ses éventuelles évolutions. En effet la modification d'un plan est plus simple que la modification d'un système réalisé.
Afin de répondre à ce besoin de description d'architecture, de nombreux travaux ont été menés par la communauté scientifique dès le début des années 90. Ces travaux concernent la définition de langages de description d'architecture [2] (*Architecture Description Languages* ou ADL).

II. DEFINITION DU LANGAGE DE DESCRIPTION D'ARCHITECTURE (ADL)

Qu'est ce qu'un ADL ?

> *Un langage de description architectural (ADL) - ou encore appelés langages de configuration - est un langage employé pour décrire une architecture de logiciel, il apparaît en tant qu'outil viable pour décrire formellement les architectures des systèmes. Il peut être un langage descriptif formelle ou semi formelle, un langage graphique, ou tous les deux. Un ADL peut être associé à un ensemble d'outils pour rendre l'analyse d'architectures plus facile.*

III. LE BUT DES ADLS

Les langages de description d'architecture permettent de formaliser les architectures logicielles.
L'exploitation du niveau architecture logicielle permet de réduire le coût et d'augmenter la performance du système logiciel. Plusieurs travaux de recherche ont montré que, pour une meilleure exploitation de l'architecture logicielle, celle-ci doit fournir son propre langage de spécification et une technique d'analyse, tel qu'un langage qui permet de représenter les propriétés du système.

IV. PROPRIETES DES LANGAGES DE DESCRIPTION D'ARCHITECTURE

Les concepteurs des langages de description d'architecture déploient actuellement un effort important sur la qualité de l'interface utilisateur. Il s'agit de rendre accessible leur langage par des utilisateurs sans grande culture informatique, de

faciliter la maintenance des applications, d'autoriser une réutilisation des composants réduisant ainsi le coût de développement.
Grâce à cette interface, les langages de description d'architecture offrent ainsi une représentation graphique claire et « métier » des applications, comme elle a bien d'autres propriétés [3].

IV.1. Systèmes volumineux et vues multiples

Les ADLs autorisent le développement de système volumineux (en termes de taille mais aussi de complexité), car ils offrent une vue globale de l'application le rendant ainsi beaucoup plus maîtrisable à son concepteur. De plus, un ADL permet de bien séparer les parties algorithmiques de son application avec sa structure, et le concepteur peut alors focaliser ses efforts sur le squelette de son application [3].

IV.2. Modularité, réutilisation

Dans le domaine de l'ingénierie où la durée de développement est un facteur qui conditionne le coût économique d'un projet, un ADL doit obligatoirement permettre la réutilisation de composants préalablement conçus, et auquel le concepteur attribue un niveau de confiance élevé. Pour les rendre réutilisables, il faut donc concevoir des composants logiciels, ou *modules*, indépendamment du contexte où ils sont censés évoluer par la suite. Cela permet ensuite d'échafauder des architectures par composition de ces « briques » de base. Avec un langage de configuration, chaque module fait donc l'objet d'une conception séparée. Et, pour que chaque composant développé ainsi puisse communiquer avec les autres, il lui est associé une interface clairement définie, visible des composants externes.
Cette démarche, proche de la programmation objet, sépare ainsi le comportement du module de son environnement et de son implémentation, et propose ainsi une vision « boîte noire » du module où les seuls points de perméabilité sont définis au niveau de l'interface. La réutilisation est ainsi une propriété intrinsèque des langages de description d'architecture, à condition qu'y soient intégrés des mécanismes de typage et d'instanciation des composants [3].

IV.3. Hiérarchisation

Une application présente souvent une nature hiérarchique en ce sens qu'elle peut être définie à des niveaux d'abstraction différents. En prenant pour exemple la hiérarchie fonctionnelle, une application peut, au cours de sa spécification, se décomposer d'abord en fonctions qui ne sont pas nécessairement toutes élémentaires (une fonction élémentaire est une fonction de base qui, par essence, ne peut plus être décomposée).
Le besoin de procéder à la spécification d'une application par raffinements successifs implique l'emploi de modules composés, qui intègrent alors les fonctionnalités à raffiner. Un *module composé* n'est plus l'enveloppe d'un code exécutable, mais une boîte encapsulant d'autres modules.
Les modules encapsulés restent alors invisibles à l'environnement du module composé : ainsi l'aspect boîte noire reste garanti. Autrement formulé, un module composé apparaît comme un module élémentaire à son environnement ; mais son interface (visible) constitue (de façon invisible) une voie d'accès aux modules encapsulés.

La nécessité de décomposer une fonction, à une étape donnée de la spécification, est un choix très subjectif. Mais la granularité finale de la décomposition est de toutes façons limitée par des contraintes opérationnelles, car un module ne peut être inférieur à un *atome* de distribution, un atome de distribution étant défini comme le plus petit élément de code insécable vis-à-vis de la répartition.
Tous les ADLs autorisent généralement le ***développement hiérarchique*** d'une application (au niveau des ***composants***, voire des ***connecteurs***) [3].

IV.4. La flexibilité

La flexibilité fait référence à la capacité de modifier la configuration d'un système en remplaçant un élément par un autre, ou en la complétant par d'autres éléments. La modification d'un système peut être une exigence de son utilisateur, et ce pour suivre l'évolution de la technologie et pour en accroître les possibilités de maintenance en phase de fonctionnement opérationnel. Cette propriété d'un système est appelée sa maintenabilitée, et elle est plus difficile à garantir si la granularité des composants est importante.
Cette flexibilité peut, dans certains cas, aboutir jusqu'à des systèmes auto adaptables : de tels systèmes sont amenés à se modifier d'eux-mêmes pour satisfaire des contraintes opérationnelles. Ce peut être par exemple pour rééquilibrer la charge de processeurs, pour redistribuer des ressources, ou encore pour réduire les coûts de communication en redistribuant des atomes d'exécution [3].

IV.5. Style d'architecture

Le style d'architecture [4] [5] permet de définir une classe générique de système (par exemple « client serveur », « pipe Unix », etc.) d'où va dériver la description architecturale. Cette notion est très utile aux concepteurs confrontés au développement de systèmes similaires. À partir d'un style bien défini, il pourra ainsi personnaliser sans trop d'effort une architecture spécifique, et étendre la qualité de « réutilisation » des composants à l'architecture des applications. Un style d'architecture doit comprendre :

- Un vocabulaire sur les composants et les connecteurs utilisés (par exemple des composants clients et serveurs),
- Des règles d'association entre ces éléments qui donnent un sens non ambigu à la structure (par exemple deux clients ne doivent pas être reliés ensemble),
- Des outils d'analyses spécifiques (par exemple détection des inters blocages entre un client et le serveur) [3].

IV.6. La vérification

Les langages de configuration ont été longtemps considérés comme utiles à des fins d'implémentation et de documentation, et non pas comme les supports de vérifications poussées des propriétés d'une application. Mais cette restriction est en passe d'évoluer.
En effet, si un langage de configuration impose un typage fort des différents objets manipulés, alors il devient aisé à un outil de vérification de détecter toutes incohérences (des objets mal connectés) ou inconsistances (des objets non connectés) présentes dans une description. Il faut naturellement que le langage attribue une sémantique non ambiguë aux entités qu'il manipule, et surtout formaliser sans équivoque les connexions [6].

Alors, si le comportement des objets est clairement défini (par un modèle «data-flow» par exemple), et si les connexions font l'objet d'une formalisation non ambiguë, l'analyse de la seule description d'une application sous sa forme architecturale permet de détecter ses risques d'interblocage ou de famine. Des langages comme Wright [7] et Rapide permettent ainsi d'effectuer des analyses comportementales de la description d'une application.

V. DESCRIPTION DES PRINCIPAUX ADLS

V.1. Le langage Rapide (D. C. Luckham, et Frank Belz 1995)

Rapide [8] est un langage de description d'architecture dont le but est de vérifier, par la simulation, la validité d'une architecture logicielle donnée [9] [Rapideweb].
Avec le langage Rapide, une application est construite à partir de modules ou de composants communiquant par échange de messages ou d'événements. Rapide fournit également un environnement composé d'un simulateur permettant de vérifier la validité de l'architecture.
Les concepts de base du langage Rapide sont les suivants: la notion d'événement, de composant, et d'architecture.
Ce langage est constitué d'un ensemble de langages (architecture, types, spécification, exécution, et patterns[2]) appelé le framework du langage de programmation de Rapide qui permet de fournir une capacité de construire facilement des composants complexes en réutilisant ou en adaptant des composants, de fournir des outils d'analyse et de supporter le raffinement des prototypes.

V.2. UniCon (M. Shaw, CMU, 1995) "Universal Connector Support"

(Support Universel De Connecteur)

Est un langage [8] de description d'architecture crée pour faire la construction et la vérification d'architectures à partir d'éléments architecturaux prédéfinis [10] [UniConweb].
Comme la plupart des langages, il est basé sur trois concepts fondamentaux qui sont le composant, le connecteur et l'architecture (configuration) [11].

V.3. Aesop (D. Garlan & al, CMU, 1994)

Est un outil conçu pour la fédération d'une famille d'environnements pour la conception et l'analyse d'architectures logicielles.
Ses éléments de bases sont le composant, le connecteur et l'architecture. Il permettre la définition de plusieurs styles d'architectures. [12] [Aesopweb].

V.4. Darwin (J. Magee, Imperial College, 1994)

Il est proposé pour la configuration et l'instanciation dynamique des systèmes distribués [13] [Darwinweb].
La sémantique associée à un composant est celle du processus. Ainsi, une instance de composant correspond à un processus créé, Les services requis ou fournis (*require* et *provide*) correspondent à des types d'objets de Communication que le composant utilise pour respectivement communiquer avec un autre composant ou recevoir une communication d'un autre composant.
Ainsi, Ils décrivent les types d'objets de communication utilisés ou autorisés à venir appeler une fonction du composant.
Parmi les types d'objet, le *port* est le plus courant : il s'agit d'un objet envoyant des requêtes de manière synchrone ou asynchrone entre composants répartis ou non.
Comme il permet de décrire un schéma d'instanciation de composants très évolué

[10]. Le langage Darwin permet de décrire la répartition des composants sur des sites différents. Celle-ci est décrite au moment de la déclaration d'instances de composants en y ajoutant un numéro de site de création.

V.5. Wright (R Allen, D Garlan, 1997)

Wright est un langage d'architecture logicielle. Il est basé sur les quatre concepts suivantes : composant, le connecteur, la configuration et le style [15] [16] [Wrightweb].

Il ne définit pas la génération de code, ni de plate-forme permettant de simuler l'application,

Alors qu'il définit Vérification formelle.

Parmi ces caractéristiques : l'absence de blocage et la description formelle de comportement d'une architecture à l'aide de la spécification CSP.

V.6. C2 (R. N. Taylor, N. Medvidovic, UCI, 1995)

Le langage C2 [10] s'appuie sur un style d'architecture logiciel en couche et des communications par échange de message. Il possède trois abstractions principales qui sont le composant, le connecteur et la configuration d'une architecture.

L'idée principale de cet ADL est de définir une application sous forme de réseau de composants s'exécutant de manière concurrente, liés par des connecteurs et communiquant de manière asynchrone par envoi de messages.

V.7. Olan (L Bellissard, M Riveill, INRIA Rhône Alpes, 1995)

Le but principal d'OLAN est de fournir un environnement complet pour la construction, la configuration et le déploiement d'applications réparties. [17][Olanweb].

Il permet la spécification et le déploiement d'un système réparti en décrivant et en automatisant le placement des composants logiciels sur des nœuds physiques.

Tab V.1 : Les services possibles dans Olan.

Syntaxe OIL pour le typage de contrôle	Contraintes liées aux paramètres	Contraintes liées au comportement
Provide	Le service a des paramètres en entrée et en sortie.	Le service fournit est un service synchrone (exemple : RPC).
React	Le service a des paramètres en entrée uniquement.	Le service fourni est un service asynchrone (exemple : service d'événement).
Require	Le service a des paramètres en entrée et en sortie.	Le service requis est un service synchrone.
Notify	Le service a des paramètres en entrée uniquement.	Le service requis est un service asynchrone.

V.8. ACME (D Garlan, R Monroe, D Wile, CMU, 1997)

Acme a été conçu comme langage passerelle pour permettre aux différents ADL's d'inter opérer.

Alors il est définit comme langage d'architecture générique. ACME apparaît plus comme un langage fédérateur de ce qui existe que comme un langage réellement novateur [18], [Acmeweb].

Les éléments pour la description de la structure de l'architecture sont les suivants :

Le composant, Le connecteur, Le système représente la configuration d'une application, et la représentation. ACME fournit un moyen de décrire des gabarits de conception (*templates*).
Cette notion est équivalente à la notion de style d'architecture que l'on peut trouver dans la plupart des ADLs. Enfin il a été conçu pour spécifier une architecture de manière syntaxique et ne se focalise pas sur la sémantique.

VI. EVALUATION DES ADLS

Tab V.2 : Evaluation des ADLs.

Points forts des ADLs	Points faibles des ADLs
▪ La possibilité de description hiérarchique des composants. ▪ Les interactions sont décrites de façon explicite. ▪ Certains ADLs travaillent sur la notion de type, classe et instance. ▪ Certains ADLs proposent la définition de style d'architecture. ▪ Certains environnements associés aux ADLs sont très complets.	▪ L'approche proposée par la plupart des ADLs est plutôt structurale, c'est-à-dire statique. ▪ Les propriétés non-fonctionnelles ne sont pas toujours prises en compte. ▪ Aucun ADL ne propose de démarche associée complète. ▪ Aucune projection vers les plates-formes à base de composants actuelles, telles que les EJB ou CCM, n'est proposée. ▪ plusieurs langages sont proches de la réalisation. Ils manquent d'abstraction.

VII. LES DOMAINES D'UTILISATION

Les ADLs ont été développés pour répondre aux besoins de différents domaines tels que les systèmes distribués ou la simulation. La table* montre les domaines d'application de chaque ADL.

Tab V.3 : Domaines d'utilisation des ADLs.

ADL	Domaine d'utilisation
Rapide	Prototypage, simulation de systèmes logiciels.
UniCon	Construction basée sur des styles architecturaux.
Wright	Spécification des propriétés comportementales d'une architecture.
Aesop	Génération d'environnement de configuration suivant plusieurs styles architecturaux.
Darwin	Systèmes distribués.
ACME	n'est pas spécifique à un domaine particulier, style générique.
C2	Systèmes distribués et évolutifs (initialement pour les Systèmes possédant une interface graphique).
Olan	la construction, la configuration et le déploiement d'applications réparties.

VIII. la vocation des langages

Les langages de description d'architecture peuvent avoir deux principales vocations suivant qu'ils intègrent des outils pour la génération des configurations exécutables (Langages de configuration) ou pour la vérification, la formalisation et la validation d'architecture (Langages de description).

Tab V.4 : La vocation des langages.

ADL	vocation du langage
Rapide	Langage de description.
UniCon	Langage de configuration.
Wright	Langage de description.
Aesop	Langage de description.
Darwin	Langage de configuration.
ACME	Langage de description et de configuration.
C2	Langage de description et de configuration.
Olan	Langage de description et de configuration.

IX. Conclusion

Cette description Permet d'indiquer que les ADLs ainsi que leurs outils associés favorisent le développement de systèmes à base de composants.

Enfin, Les ADLs sont des langages de conception qui permettent la définition de modèles abstraits pour l'assemblage, ceci dès la phase de conception. Ces modèles abstraits ne sont pas complets et manquent de liens avec les modèles de réalisation.

Bibliographie

[1] D. Bennouar : « Vers une mise en œuvre conjointe des ADL et UML dans le développement de logiciels complexes » Laboratoire LSDRI, Département d'Informatique, Faculté des Sciences, Université de Blida, 2004

[2] ACCORD, Etat de l'art sur les Langages de Description d'Architecture (ADLs), Projet ACCORD (Assemblage de composants par contrats en environnement ouvert et réparti), juin 2002.

[3] E. DURAND A.M. DEPLANCHE Y. TRINQUET. «Un aperçu sur quelques langages de description d'architecture». IRCyN –LangArchi, mars 1999.

[4]D. GARLAN and M. SHAW. « An Introduction to Software Architecture ». Advances in Software Engineering & Knowledge Engineering, vol. II : 1-39, 1993.

[5] D.E. PERRY and A.L. WOLF. « Foundations for the Study of Software Architecture ». ACM SIGSOFT Software Engineering notes, vol. 17(4) :40-52, october 1992.

[6]R. ALLEN and D. GARLAN. « Formalizing Architectural Connection ». In proceedings of the 16th International Conference on Software Engineering. p. 71-80, Sorrento, Italy, may 1994.

[7]R. ALLEN and D. GARLAN. « A Formal Basis for Architectural Connection ». ACM trans. On Software Engineering and Methodology, July 1997.

[8] T. Khammaci, M. Oussalah, A. Smeda, Les ADLs : une voie prometteuse pour les architectures logicielles, Actes du congrès Agents, Logiciels, Coopération, Apprentissage & Activité Humaine, (ALCAA 2003), Septembre 2003, Bayonne, France.

[9] David C. Lukham: "Rapide, A langage Toolset for Simulation of Distributed System By Partial Ordering of Events" In Proceedings of DIMACS Workshop on Partiel Order Methods in Verification (POMIV) page 329-358, July 25-26, 1996.

[10] Gregory Zelesnik: "The UniCon langage Reference Manual", Carnegie Mellon University, Pittsburgh, Pensylvania, 1996

[11] N. Medvidovic, R. N. Taylor. «A Classification and Comparison Framework for Software Architecture Description Languages ». IEEE Transactions on Software Engineering, Vol 26, no1, pp. 70-93, Jan. 2000.

[12] David Garlan: «An Introduction to the Aesop System». Version of 11 July 1995. http://www-2.cs.cmu.edu/Groups/able/aesop/html/aesop-overview.ps

[13] Naranker Dulay: "Darwin Tutorial & Reference" http://www-dse.doc.ic.ac.uk/dse-papers/darwin/ref-manual.ps.gz

[14] J. MAGEE, N. DULAY and J. KRAMER. « Regis: A Constructive Development Environment for Distributed Programs ». Distributed Systems Engineering, vol. 1(5):304-312, September 1994.

[15] R. ALLEN. « A Formal Approach to Software Architecture ». PhD thesis, School of Computer Science, carnegie Mellon University, Pittsburgh, 1997.

[16] Robert J. Allen: "A Formal Approach to Software Architecture", PhD. Thesis, Carnegie Mellon University, Technical Report Number: CMU-CS-97-144, May, 1997. http://www-2.cs.cmu.edu/~able/publications/rallen-thesis/

[17]Luc Bellissard: "Construction et configuration d'application répartie", thèse Informatique, Systèmes et Communication, IMAG/INRIA, Grenoble, 1997. http://publications.imag.fr/publications/theses/annees/1997/Bellissard.Luc/these.dir

[18] David Garlan, Robert Monroe, and Dave Wile: "Acme: An Architecture Description Interchange Language" In Proceedings of CASCON 97, Toronto, Ontario, November 1997, pp. 169-183. http://www-2.cs.cmu.edu/~able/publications/acme-cascon97/

Sites Web

[Aesopweb] "Aesop, Software Architecture Design Environment Generator", Home Page: http://www-2.cs.cmu.edu/Groups/able/aesop/aesop_home.html
[Rapideweb] "The Stanford Rapide Project", Home Page: http://pavg.stanford.edu/rapide/
[UniConweb] "UNICON", Home Page: http://www-2.cs.cmu.edu/People/UniCon/
[Darwinweb] "The Darwin Architecture description Langage", Home Page: http://www.doc.ic.ac.uk/~igeozg/Project/Darwin/
[Wrightweb] "The Wright Architecture Description Language", Home Page. http://www-2.cs.cmu.edu/afs/cs/project/able/www/wright/index.html
[C2web] "Chiron-2, A Component and essage Based Architectural Style for GUI Software" (1995) http://www.ics.uci.edu/~arcadia/C2/c2.html
"The C2 architectural style»: http://www.isr.uci.edu/architecture/c2StyleRules.html
[Olanweb] "Olan: Un environnement de développement d'applications coopératives" (1995) http://rangiroa.essi.fr/riveill/recherche/95-olan-francais.html
[Acmeweb] "The Acme Architectural Description Langage": http://www-2.cs.cmu.edu/~acme/

Partie II

Le modèle de connecteur en couches et son contexte

Chapitre VI

Le Modèle COSA

I. INTRODUCTION

Les langages de description d'architectures ADLs décrivent un système complexe comme un ensemble de composants qui interagissent entre eux par l'intermédiaire de connecteurs. En général, ces interactions sont définies au sein des composants sous forme de fichiers et d'instructions d'entrées/sorties (les détails des interactions sont entièrement encapsulés dans les composants). Ainsi, la description implicite des interactions (connecteurs) rend difficile la construction de composants hétérogènes lesquels fournissent des fonctionnalités complexes et sont reliés par des relations complexes. Cette approche consiste à séparer explicitement les concepts de composants et de connecteurs en tenant compte que les concepteurs appuyant sur les paradigmes des architectures logicielles et des architectures objets. Ils ont baptisé cette approche, l'approche COSA : *Component-Object based Software Architecture*[1]. Les concepts de base sont les composants, les connecteurs, les interfaces, les configurations, les contraintes et les propriétés. COSA décrit un système en termes de types et d'instances. Les composants, connecteurs et les configurations sont des types qui peuvent être instanciés pour construire plusieurs architectures. Dans cet article, nous nous focalisons uniquement sur le concept de connecteurs. Ces derniers sont considérés comme des entités de première classe, et peuvent être, au même titre que les composants, décrits explicitement, réutilisés (utilisés) et maintenus en empruntant des mécanismes issus des systèmes à objets tels que l'instanciation, la composition, l'héritage, la généricité et le raffinement [2].

II. CONCEPTS DE BASE DE COSA

L'architecture logicielle concerne la description de systèmes à base de composants (unité de calcul) et de connecteurs (unité d'interactions entre les composants) [3]. Ainsi, l'activité de développement de logiciels ne se focalise plus sur l'étape d'écriture des lignes de code (implémentation) mais plutôt à un niveau beaucoup plus élevé qui est celui de l'architecture du logiciel. L'abstraction dans le développement de logiciels procure plusieurs avantages tels que la réutilisation, la réduction des coûts de développement de logiciels, la diminution du temps de développement ou l'augmentation de la fiabilité [4] [3].

L'architecture COSA décrit un système en termes de types et d'instances. Les composants, les connecteurs et les configurations sont des types qui peuvent être instanciés pour construire plusieurs architectures logicielles. Les concepts de base de l'architecture COSA sont les composants, les connecteurs, les configurations, les interfaces, les contraintes et les propriétés. Ces concepts partagent une base conceptuelle similaire qui peut être qualifiée d'ontologie [4]. La *figure* 1 décrit le méta-modèle de l'architecture COSA. Cette figure montre entre autres, que l'architecture COSA distingue deux types d'interfaces : l'interface d'un composant (appelée *port*) et l'interface de connecteur (appelée *rôle*) et que les interfaces fournissent des points de connexion entre les composants et les connecteurs.

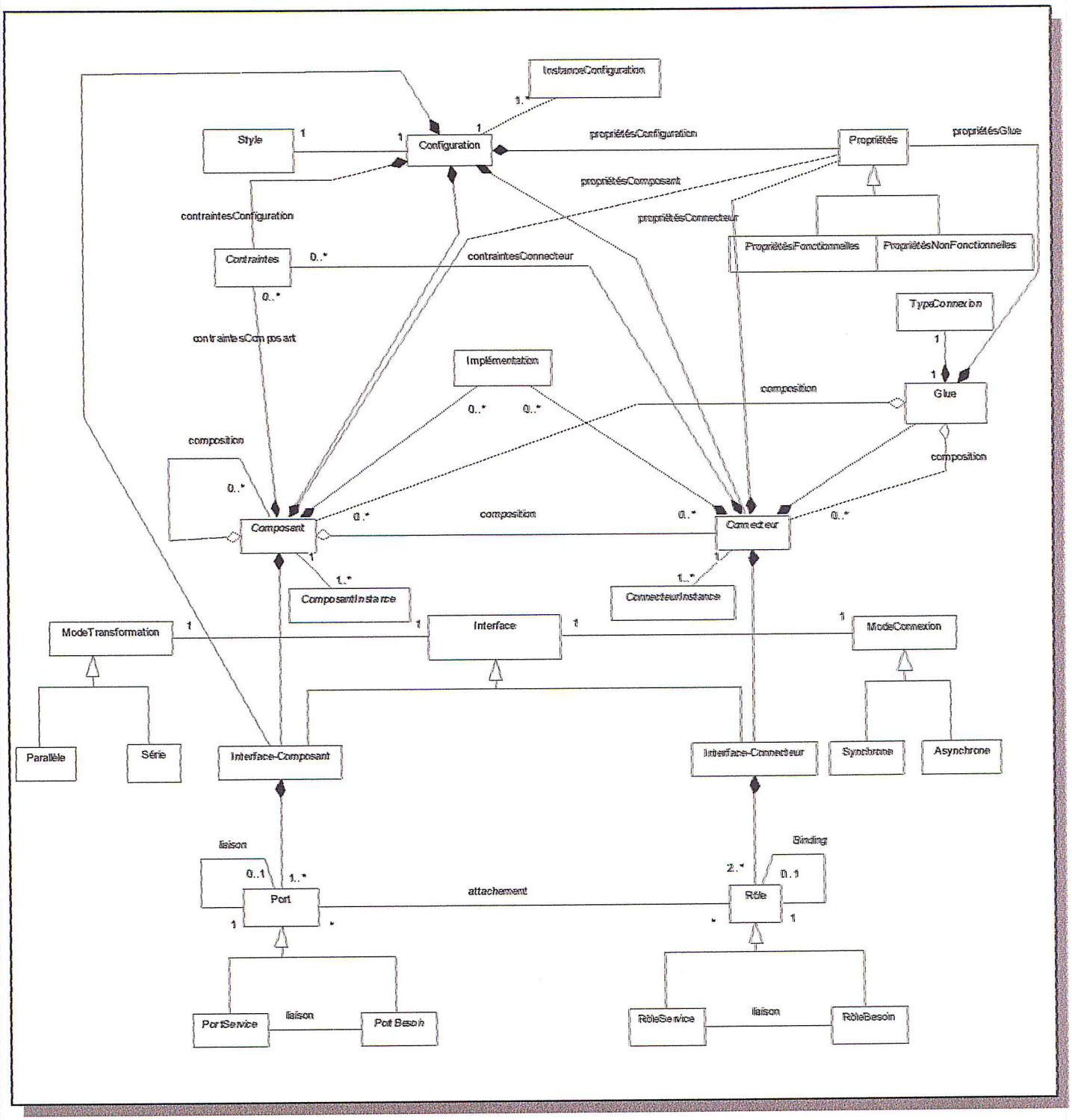


Figure VI.1 : Méta-modèle décrivant les concepts de base de l'architecture COSA

II.1. Le concept connecteur dans l'architecture COSA

Un connecteur est principalement définit par une *interface* et une *glu*, comme le montre le diagramme de classes de la *figure* 2. En principe, *l'interface* décrit les informations nécessaires du connecteur, y compris le nombre de rôles, le type de services fourni par le connecteur (communication, conversion, coordination et facilitation), le mode de connexion (synchrone et asynchrone), le mode de transfert (parallèle et série), etc. Les points d'interaction d'une *interface* sont appelés *rôles*. Un *rôle* est une interface d'un connecteur appelé à être relié à une interface d'un composant (un port de composants). Un *rôle* est soit de type « Besoin » ou de type « Service ». En principe, un *rôle* est une interface générique d'un connecteur qui sera relié à une interface d'un composant. Un *rôle* « Service » sert comme un point d'entrée dans l'interaction d'un composant représentée par une instance d'un type de connecteur. Il est appelé à être connecté à une interface « Besoin » d'un composant (ou a un *rôle* « Besoin » d'un autre connecteur).

De manière similaire, un *rôle* « « Besoin » sert comme point de sortie de l'interaction d'un composant représenté par une instance d'un type de connecteur et ce *rôle* a pour but de se connecter à une interface « Service » d'un composant (ou a un rôle « Service » d'un autre connecteur). Le nombre de *rôles* d'un connecteur représente le *degré* d'un type de connecteur. Par exemple, dans un type de connecteur Client-Serveur, la représentation de l'interaction d'appels de procédures entre les entités Client et Serveur est un connecteur de *degré* 2. Les interactions complexes, entre au moins trois composants, sont représentées par des types de connecteurs de *degré* supérieur à 2.

Certains langages de description d'architectures proposent des connecteurs avec uniquement deux *rôles*, et un connecteur ne peut être relié qu'à un autre connecteur (cas des langages C2 [5] et Acme [6]). Une *interface* peut avoir ses propres *propriétés* et ses propres *contraintes*.

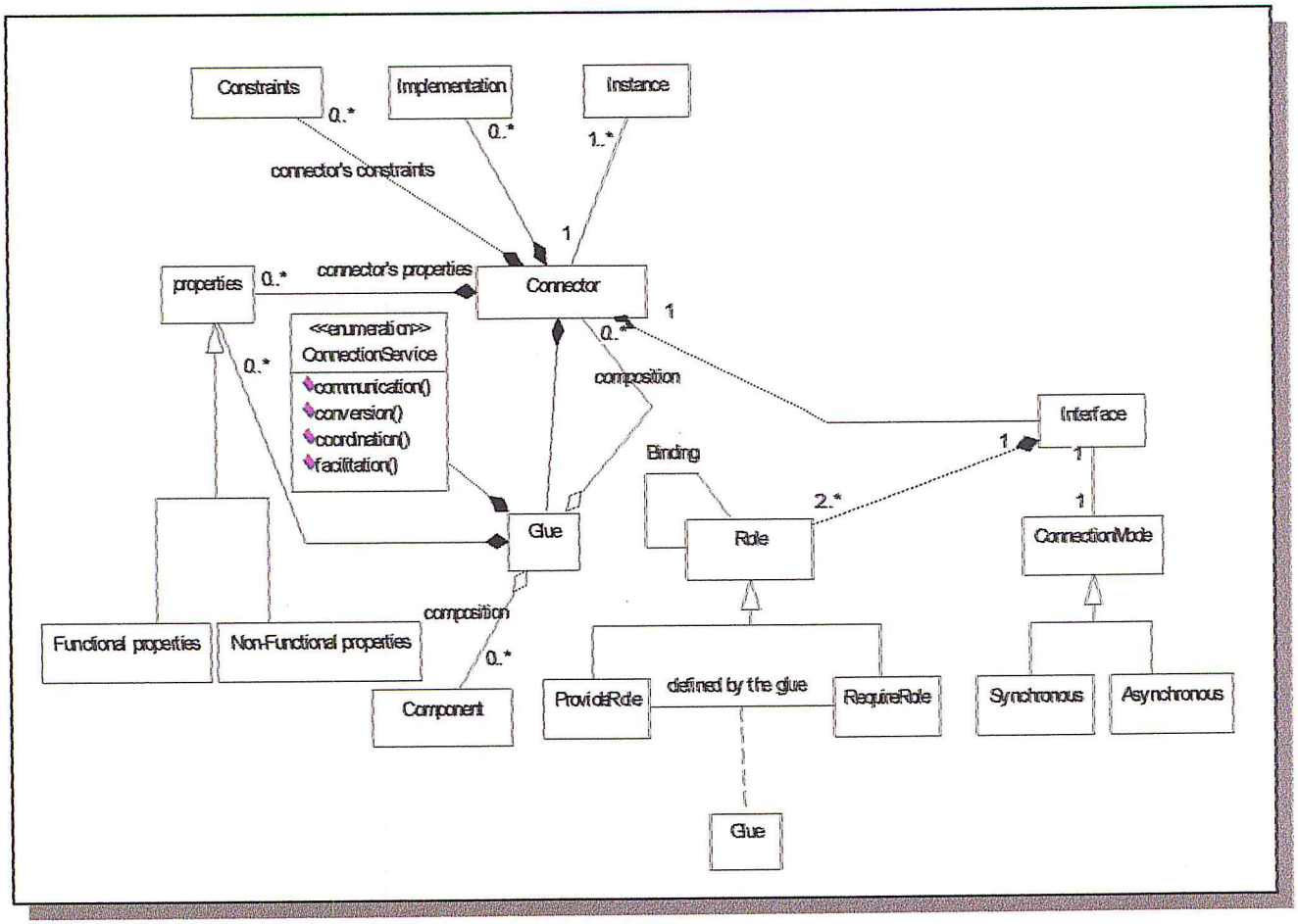


Figure VI.2 : Diagramme de classes du méta-modèle des connecteurs dans COSA

L'*interface* est la partie visible du connecteur. Ainsi, elle doit contenir assez d'information sur le service et le type de ce connecteur. De ce fait, en examinant seulement l'*interface*, nous pouvons décider si un connecteur donné peut être réutilisé ou non.
La *glu* décrit les fonctionnalités attendues d'un connecteur. Elle peut être un simple protocole reliant des *rôles* ou un protocole complexe ayant plusieurs opérations telles que le lien, la conversion de format de données, le transfert ou l'adaptation. En général, la *glu* d'un connecteur représente le type de connexions de ce connecteur. Les connecteurs peuvent avoir leur propre architecture interne qui contient des calculs et du stockage de données. Par exemple, un connecteur peut exécuter un algorithme de conversion de données d'un format à un autre. Ainsi, le service fourni par un connecteur est défini par sa *glu*. Les services d'un connecteur peuvent être de type communication, conversion, coordination ou facilitation.
Dans le cas d'un connecteur composite, les sous-connecteurs et les sous-composants de ce connecteur doivent être définis dans la *glu* ainsi que les liens entre les sous-connecteurs et les sous-composants. La *glu* peut aussi avoir ses propres *propriétés* et ses propres *contraintes* en plus des *propriétés* et des *contraintes* du connecteur. La *figure* 3 présente la définition explicite des connecteurs dans l'approche COSA.

```
Classe Connecteur nom-connecteur {
Interface {
        Rôles {……} // définition des rôles
        Propriétés {……} // propriétés de l'interface
        Contraintes {……} // contraintes de l'interface
        Type-Service {……} // communication, conversion, coordination ou facilitation
        Mode-Connexion = …… // synchrone ou asynchrone
        Mode-Transfert = …… // parallèle ou série
} // fin Interface
Glu {// définition de la glu
        Définition-Service {……} // communication, conversion, etc.
        Définition-Composition {
        Classes {……} // définit les sous-composants et les sous-connecteurs
        Liaison {……} // définit les liaisons sous-composants/sous-connecteurs
        }
        Propriétés {……} // propriétés de la glu
        Contraintes {……} // contraintes de la glu
} // fin Glu
Propriétés {……} // propriétés du connecteur
Contraintes {……} // contraintes du connecteur
}
```

Figure VI.3 : Structure d'un connecteur COSA

II.2. Le concept composant

Les composants représentent les éléments de calcul et de stockage des données d'un système. Chaque composant peut avoir plusieurs *interfaces* ayant plusieurs *ports*. Chaque interface est composée de plusieurs points d'interaction entre les composants et le monde extérieur qui permettent l'invocation des services. Un composant peut avoir plusieurs implémentations. La *figure* 4 décrit un composant d'une architecture COSA. Notons que cette description du composant est similaire à celle fournie par les autres approches d'architectures logicielles [7] [8] [9] [10] [11] [12].

```
Classe Composant Nom-Composant {
Interface {
Port {définition des ports}
…} // Autres services de l'interface
    Composition-Définition {…..}
            // Définit les sous-composants et leurs liaisons
Propriétés {Définition des propriétés du composant;}
Contraintes {Définition des contraintes du composant;}
}
```

Figure VI.4 : Structure d'un composant COSA

II.3. Le concept configuration

Les configurations représentent un graphe de composants et de connecteurs. Les configurations sont aussi appelées représentations ou systèmes [12]. Dans COSA, les configurations sont des classes qui peuvent être instanciées plusieurs fois et qui donneront plusieurs architectures d'un système. Une configuration peut avoir une ou plusieurs interfaces définissant des ports, qui a leur tour, seront reliés aux ports des composants internes. En général, les configurations sont structurées de manière hiérarchique : les composants et les connecteurs peuvent représenter des sous-configurations qui disposent de leurs propres architectures. La *figure* 5 définit les principales parties d'une configuration COSA.

```
Classe Configuration nom-configuration
{
        Interface {Port  {définition des ports;}
                  Liaison {définition  des liaisons entre les ports configurations
                  et les ports des connecteurs;}
        Classe Composant {définition du composant;}
        Classe Connecteur {définition du connecteur;}
        Propriétés {propriétés de la configuration;}
        Contraintes {contraintes de la configuration;}
}

        Instance Configuration nom-configuration nom-architecture {
                Instances {définition des instances de composants et de connecteurs}
                Attachements {description des connexions entre les ports et les rôles}}
                {..... // définition d'autres architectures si elles existent}
        }
```

Figure VI.5 : Structure d'une configuration COSA

II.4. Les autres concepts de COSA

Interfaces: Dans COSA, les interfaces sont des entités de première classe. Elles fournissent des points de connexion entre les composants et les connecteurs. Aussi, elles définissent comment s'effectue la communication entre deux composants. Un point de connexion d'une interface d'un composant est appelé port alors qu'un point de connexion d'une interface d'un connecteur est appelé rôle. En plus des ports et des rôles, les interfaces peuvent fournir d'autres services telles que la diffusion. Les interfaces peuvent avoir des propriétés et des contraintes. Enfin, une interface peut transférer des données de manière parallèle ou en série et la communication peut être synchrone ou asynchrone.

Propriétés : représentent des informations sémantiques d'un système, de ses composants et de ses connecteurs. Elles servent aussi à documenter les détails d'une architecture. Il existe deux types de propriétés : les propriétés fonctionnelles et les propriétés non fonctionnelles. Les propriétés fonctionnelles concernent la sémantique des fonctions d'un système alors que les propriétés non-fonctionnelles représentent d'autres besoins d'un système tels que la sécurité, la portabilité ou la performance.

Contraintes : définissent certaines règles sur l'utilisation des composants et des connecteurs. Elles sont considérées comme un type de propriétés spécifiques et elles permettent à un modèle d'architecture de rester valide même s'il évolue dans le temps. Enfin, elles peuvent inclure des restrictions sur les valeurs permises des propriétés.

III. LES MECANISMES OPERATIONNELS POUR LA REUTILISATION DES COMPOSANTS ET DES CONNECTEURS

La définition de connecteurs comme des entités de première classe au même titre que les composants permettent leur réutilisation à travers des mécanismes tels que l'instanciation, l'héritage, la composition, le raffinement ou la généricité. Les mécanismes que nous utilisons pour la réutilisation et l'évolution des connecteurs sont identiques aux mécanismes utilisés pour la réutilisation et l'évolution des composants. Dans la suite, nous détaillons uniquement les mécanismes d'instanciation, d'héritage et de composition des composants et des connecteurs.

III.1. Le mécanisme d'instanciation

La réutilisation de logiciels est l'un des premiers objectifs du génie logiciel. Comme la décomposition architecturale est réalisée à un niveau d'abstraction au-dessus du code, l'architecture logicielle doit supporter la réutilisation en modélisant les connecteurs comme des types. Les types de connecteurs peuvent être instanciés plusieurs fois et chaque instance peut correspondre à plusieurs implémentations. Chaque instance d'un type doit contenir la structure définie par le type et doit avoir le même comportement que celui du type. Si certaines propriétés sont définies sur le type, toute instance de ce type doit avoir ces propriétés.

L'architecture logicielle d'un système est définit par un modèle de composants et un modèle de connecteurs. Les types de composants sont des abstractions qui contiennent des services dans les blocs réutilisables alors que les types connecteurs sont des abstractions qui contiennent des décisions de médiation, coordination, communication entre composants [13] [14]. Dans COSA, nous utilisons le mécanisme d'instanciation classique de l'approche objet où les composants et les connecteurs sont définis comme des classes. La *figure* 6 décrit un exemple de construction d'une configuration Pipe-Filtre en utilisant les composants Filtre1, Filtre2 et le connecteur Pipe et leurs C1, S1, C1-S1.0

```
Classe Configuration PipeFiltre {
   Classe Composant Filtre1 {
       Interface {
                    Port Service {protocole Service;}}
              Propriétés {débit=10kb/s}
   }
   Classe Composant Filtre2 {
       Interface {
                    Port Besoin {protocole Besoin;}}
        Propriétés {débit=5kb/s}
    }
   Classe Connecteur Pipe {
        Interface {
            Rôles {entrée {entrée protocole rôle;}
                    sortie {sortie protocole rôle;}
                   }}
        Glue {
              Type-Connexion {
                 lire entrée;
                 écrire sortie;}
            }
        Propriétés {max-rôles =2;}
                     } }

Instance Pipe-Filtre exemple-d-instance {
        Instances {
              S1 : Filtre1; // S1 est une instance de Filtre1
              C1 : Filtre2; // C1 est une instance de Filtre2
              C1-S1 : Pipe; // C1-S1 est une instance de Pipe
              C1-S1.propriétés.max-rôles=2 ;} // Mettre le nombre maximum de rôles à 2

      Attachements {
          C1.Besoin to C1-S1.lire;
          S1.Service to C1-S1.écrire;}
   }
```

Figure VI.6 : Exemple d'instanciation dans COSA.

III.2. Le mécanisme d'héritage

L'héritage fournit une classification naturelle de types de classes. C'est un puissant outil pour l'évolution et la réutilisation. La relation d'héritage telle qu'elle est définit au niveau des langages orientés objets est très intéressante pour les systèmes à base de composants. Pour modéliser des systèmes complexes, l'héritage dans les systèmes à base de composants doit être sélectif et dynamique. L'héritage sélectif permet de s'assurer qu'un connecteur peut avoir plusieurs types de combinaisons (interface et propriétés, interface et comportement, ...) en utilisant, par exemple, des mots clés tels que «*with*», « *without* » et « *extend* ». L'héritage dynamique, quant à lui, consiste à permettre aux sous-connecteurs et aux composants de ne modifier que ce qu'ils héritent. Aussi, l'héritage peut être de type multiple où un connecteur ou un composant peut hériter des attributs ou des méthodes de plusieurs super-composants ou super-

connecteurs. Dans notre modèle, l'héritage est sélectif, dynamique et de type multiple. Dans l'architecture COSA, l'héritage ne concerne pas uniquement les composants et les connecteurs mais aussi les configurations. La *figure* 7 décrit un exemple d'héritage de configuration (*Client-serveur-2*) et de composants (*Client-2 et Serveur-2*) dans l'architecture COSA.

```
Classe Configuration Client-Serveur-2 extend client-serveur {
        // La configuration Client-Serveur-2 hérite de la configuration client-serveur
        // et ajoute un nouveau connecteur RPC-2
    Classe Composant Serveur-2 extend Serveur whithout Interface {
        // Serveur-2 hérite de serveur et définit une nouvelle interface
        Interface {
                Port Service {protocole Service}
            Autres-interfaces {Diffusion ;}
    }
    Classe Composant Client-2 extend Client whithout Propriétés {
        Propriétés {type-données= format-1;}}
            // Client-2 hérite de la classe client et définit de nouvelles propriétés
    }
    Classe Connecteur RPC-2 {// définition d'un nouveau connecteur
        Interface {
            Rôles {entrée, sortie}}
            Glue {
            Type-Connexion {
                Lire entrée ;
                Ecrire sortie ;}
            Propriétés {bidirectionnelle ;}
            }}
  Instance Configuration example-héritage {// description d'une nouvelle architecture
//avec un serveur connecté à deux clients en utilisant deux connecteurs différents
    Instances {
        S1 : Serveur-2 ;
        C1 : Client ;
        C2 : Client-2 ;
        C1-S1 : RPC ; // utilisation de RPC quant la conversion est nécessaire
        C2-S2 : RPC-2 ; // utilisation de RPC-2 quant la conversion n'est pas
nécessaire
        }
    Attachements {
        S1.service à C1-S1.participateur-1;
        C1.besoin à C1-S1.participateur-2;
        S1.service à C2-S2.entrée;
        C2.besoin à C2-S2.sortie;
        }}}
```

Figure VI.7 : Exemple d'héritage dans COSA.

III.3. Le mécanisme de composition

La composition permet de construire un système en utilisant des modèles déjà existants. Elle permet à l'architecte logiciel de décrire un système à différents niveaux de détails [12]. Cependant ce mécanisme nécessite que les modèles aient des interfaces bien définies puisque leurs structures internes sont cachées. Comme ces modèles sont traités comme des boîtes noires (la structure interne du modèle est cachée, seule son interface est visible), ce type de réutilisation est dit *« réutilisation boîte noire »* [15]. Une entité composite partage la relation de composition de ses composants qui sont des entités composites comme au niveau de la relation d'héritage. Les propriétés partagées dans la composition des entités ont souvent une sémantique d'héritage. Ce type d'héritage est, dans certains cas, appelé *héritage horizontal* pour le distinguer de l'héritage de la relation entre une entité et ses représentants.

```
Classe Connecteur pipe-composite {
        Interface {
                Rôles {lire;
                       écrire;}
                }
    Glue {
            Composition-Définition {
                Composants {
                    Connecteur-1;
                    Connecteur-2;
                    }
                Liaison {
                    lire à connecteur-1.Besoin;
                    Connecteur-1.Service à Connecteur-2.Besoin;
                    Connecteur-2.Service à écrire;
                }

        }
    }
```

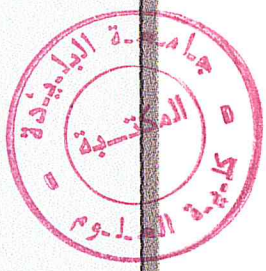


Figure VI.8 : Exemple de composition dans COSA.

Bibliographie

[1] A. Smeda, T. Khammaci, M. Oussalah, "A Multi-paradigm Approach to Describe Software Systems", *Proceedings of the 3rd WSEAS International Conference on Software Engineering, Parallel and Distributed Systems (SEPADS'04)*, Salzburg, Austria, February 2004.

[2] A. Smeda, M. Oussalah, T. Khammaci, "Software Connectors Reuse in Component-Based Systems", *Proceedings of the IEEE International Conference on Information Reuse and Integration (IRI'03)*, Las Vegas, Nevada, October, 2003.

[3] Ofta, " Architecture de logiciels et réutilisation de composants", Collection ARAGO, No 24, Editions TEC & DOC, Paris, octobre 2000.

[4] T. Bures, F. Plasil, "Scalable Element-Based Connectors", *Proceedings of the 1st International Conference on Software Engineering Research and Applications*, San Francisco, USA, Jun 2003.

[5] N. Medvidovic, D. Rosenblum, R. Taylor, "A Language and Environment for Architecture-Based Software Development and Evolution", *Proceedings of the 21st ICSE*, Los Angeles, 1999, pp. 44-53.

[6] D. Garlan, R. Monroe, and D. Wile, "ACME: An Architecture Description Interchange Language", *Proceedings of CASCON'97*, Toronto, Ontario, November 1997, pp. 169-183.

[7] R. Allen, D. Garlan, "A formal Basis for Architectural Connection", *ACM Transaction on Software Engineering and Methodology*, Volume 6, Issue 3, July 1997, pp. 213-249.

[8] D. Garlan, R. Monroe, D. Wile, "ACME: An Architecture Description Interchange Language", *Proceedings of CASCON'97*, Toronto, Ontario, November 1997, pp. 169-183.

[9] D. Garlan., S. Cheng, J. Kompanek, "Reconciling the Needs of Architectural Description with Object-Modeling Notations", *Proceedings of the Third International Conference on the Unified Modeling Language*, York, UK, October 2000.

[10] D.C. Luckham, L. M. Augustin, J. J Kenny, J. Vera, D. Bryan, W. Mann, "Specification and analysis of system architecture using Rapide", *IEEE Transactions on Software Engineering*, Vol. 21, no. 4, April 1995, pp. 336-355.

[11] N. Medvidovic, D. Rosenblum, R. Taylor, "A Language and Environment for Architecture-Based Software Development and Evolution", *Proceeding of the 21st International Conference on Software Engineering*, Los Angeles, CA., May 1999, pp. 44-53.

[12] N. Medvidovic, R. N. Taylor, "A Classification and Comparison Framework for Software Architecture Description Languages", *IEEE Transactions on Software Engineering*, Vol. 26, 2000, pp. 70-39.

[13] A. Lopes, M. Wermelinger, J. L. Fiadeiro, "A compositional Approach to Connector Construction", *Proceedings of the 15th workshop on Algebraic Development Techniques*, Genova, Italy, 1-3 April 2001.

[14] D. Perry and A. Wolf, "Foundations for the Study of Software Architectures", ACM SIGSOFT Software Engineering Notes, 17(4), (1992), pp. 40-52

[15] K. Ostermann, M. Mezini, "Object-Oriented Composition Untangled", *Proceedings of OOPSLA 2001*, Tampa Bay, Florida, November 2001.

Chapitre VII

Modèle de connecteur en couches dans le contexte COSA

Le connecteur doit fournir les mécanismes permettant de supporter tous les aspects d'un protocole de communication. La mise au point de modèles prédéfinis pour les ADL est d'une grande importance pour rendre plus efficace le travail des architectes logiciels. Pour les connecteurs dédiés au transport d'interaction, un modèle architectural en couches semble le plus adéquat. Un travail intense de recensement des différentes techniques d'interconnexions et de connecteurs est nécessaire pour dégager un tel modèle, et aussi pour déterminer les domaines de conception associés. [1]

I. INTRODUCTION

Aujourd'hui, il est largement reconnu que la conception d'un système logiciel complexe doit commencer par la définition d'une architecture de système. On l'identifie également que la conception des futurs systèmes complexes sera accomplie d'une voie semblable à ceux trouvées dans l'architecture d'ordinateur, où le système entier est visualisé comme ensemble de composants joints par des connecteurs, donnant une topologie qui répond à la fonctionnalité et à l'exigence du système désiré. Cette façon est loin de l'approche conventionnelle où le logiciel a été principalement considéré comme structure d'algorithme et de données.

Tout en élaborant les systèmes logiciels, les architectes focalisent toute leur compétence principalement en recherchant, en concevant et en décidant quel composant est le plus approprié à réaliser des buts du système. Une fois trouvés, et librement de leur niveau d'abstraction, ces composants sont joints par des connecteurs. Dans l'esprit d'un architecte, un connecteur joue simplement le rôle des concepts portants résultant de l'interaction des composants joints.

Récemment, plusieurs concepts et outils sont unanimement acceptés et considérés comme éléments fondamentaux de l'architecture logicielle, tels que les notions de composants, connecteurs, configuration, style architectural ou langage de description d'architecture [2], il reste que certains concepts, notamment les connecteurs, nécessitent une plus grande compréhension et une plus grande maîtrise.

II. NOTIONS DE CONNEXION ET DE CONNECTEUR

Dans la section précédente, nous avons insisté sur une distinction entre la notion de connexion (ou interconnexion) et celle de connecteur. Nous pensons qu'il est nécessaire de donner à chaque terme son sens et son espace de validité. Nous remarquons à ce niveau, qu'il est nécessaire que l'architecture logicielle, comme une discipline du génie logiciel, se dote d'un vocabulaire précis concis et sans aucune ambiguïté.

La distinction entre la notion de connexion et connecteur permet de tracer la première frontière permettant de distinguer clairement le rôle des composants et le rôle des connecteurs dans une architecture logicielle. Une connexion est un lien direct entre deux ou plusieurs composants. Malgré sa forte relation avec le connecteur, une connexion décrit une relation entre composants. Elle définit les rôles que joue chaque composant dans le contexte de la relation. La sémantique de la relation est perceptible au niveau de l'architecture et a un impact sur le fonctionnement global de l'architecture. Elle représente un aspect fonctionnel de l'architecture. A titre d'exemple, une connexion représentant une interaction décrit les concepts échangés, l'ordre des interactions ou les décisions à prendre suite à une interaction.

La notion de connecteur, quant à elle, décrit l'infrastructure permettant de véhiculer les différents concepts d'une connexion. Un connecteur n'a aucune connaissance du sens associé au concept de connexion qu'il véhicule. A titre d'exemple, un connecteur véhiculant une interaction ne comprend pas le sens des éléments échangés dans le contexte de l'interaction et ne peut pas intervenir sur le déroulement de l'interaction.

L'interconnexion peut avoir un sens bien précis, mais dans cet article, nous l'assimilons à une connexion. Une interconnexion peut être très simple, mais sa prise en charge peut nécessiter un connecteur simple comme elle peut nécessiter une infrastructure de communication composée d'éléments dédiés à la réalisation efficace de l'interconnexion. Cette distinction entre les deux concepts n'est pas précisée de manière explicite dans la littérature sur l'architecture logicielle. Seule la bibliographie traitant des connecteurs bien particuliers (connecteurs d'interaction) distingue les deux

concepts. A titre d'exemple, les connecteurs décrits dans COSA [3] et WRIGHT [4] [5] correspondent plutôt à la notion de connexion.
Par ailleurs, les connecteurs transportent les concepts définis dans une connexion entre composants. Ce transport se fera dans le contexte d'un protocole décrivant le comportement d'une connexion. Le protocole permet l'exploitation convenable du connecteur et le contrôle de l'adhérence des composants en connexion aux règles définies dans le protocole. Un protocole peut être très simple ou complexe. A titre d'exemple, si nous prenons comme connecteur une variable scalaire, une des actions du protocole serait de ne soumettre à ce connecteur que les valeurs appartenant au type de la variable. Un protocole plus simple ne réalise pas le contrôle de typage des valeurs mises sur ce connecteur. Le protocole veille à ne soumettre aux connecteurs que les concepts pouvant être transportés par le connecteur. Le mode de transport de concepts (sens unique, dans les deux sens avec blocage, dans les deux sens mais sans blocage, etc.) sont des éléments définis au niveau du protocole.

```
System SimpleExample
  component Server =
      port provide [provide protocol]
      spec [Server specification]
  component Client =
     port request [request protocol]
     spec [Client specification]
  connector C-S-connector =
     role client [client protocol]
     role server [server protocol]
     glue [glue protocol]
  Instances
    s: Server
    c: Client
    cs: C-S-connector
  Attachments
    s.provide as cs.server;
    c.request as cs.client
end SimpleExample.
```

Figure VII.1 : Description WRIGHT d'une communication Client-Serveur.

Dans le langage WRIGHT [4] [5] et dans le modèle COSA [3], il est possible de définir la fonctionnalité d'un connecteur (une connexion à notre sens), appelée protocole du connecteur. Cette définition est réalisée au niveau de la clause appelée *glue* (Figure 1). Le protocole du connecteur WRIGHT a pour objectif de contrôler le bon déroulement du protocole entre les rôles d'un connecteur Dans cette description, le protocole associé à un connecteur est décomposé entre le connecteur et les composants en interaction. Dans l'exemple de la figure 1, nous distinguons un protocole du serveur, celui du client et la *glue* qui aura la charge de spécifier les règles correctes du déroulement du protocole. La vue de WRIGHT sur les connecteurs ne correspond pas à la notre, dans laquelle nous essayons de réduire à néant un quelconque rôle fonctionnel des connecteurs dans l'architecture.
Dans COSA [3], le connecteur offre quatre catégories de services (communication, conversion, coordination ou facilitation). Si la communication, la conversion et la

facilitation peuvent être des catégories de services d'une infrastructure de communication, la coordination doit être prise avec beaucoup d'attention. Si la coordination indiquée correspond à une forme de synchronisation des fonctionnalités des différents composants d'une architecture, alors elle est plutôt un élément du comportement de toute l'architecture et non pas d'un connecteur. La coordination pourrait être remplie dans le contexte d'un protocole bien précis et peut être dans le contexte d'un style architectural. Dans d'autres approches [6] [7], la coordination n'est pas un service de connecteur mais un qualificatif fait par un observateur externe au connecteur, sur le travail de ce dernier. A titre d'exemple, un appel de procédure est un connecteur qui transporte des valeurs et le contrôle. Le passage de contrôle qui est observable entre les différentes procédures est vu comme une opération de coordination. Celle- ci est implicite ou gérée par une fonction bien précise, telle que la fonction principale.

Un connecteur peut avoir une structure interne plus ou moins complexe, dépendant des concepts qu'il est capable de transporter et l'environnement dans lequel il opère. Le modèle proposé par COSA [3] permet de spécifier l'architecture de connecteurs à base de sous- connecteurs et sous-composants. Cette composition peut correspondre à une infrastructure de communication, qui se charge de réaliser les objectifs du connecteur en terme de transport de concept d'un composant vers un autre, mais aussi en terme de caractéristiques telles que la sûreté des transferts de données ou la performance des acheminements.

Un connecteur complexe est composé de composants et de connecteurs. Le rôle des composants dans un connecteur se concentre sur l'amélioration des services de transfert et la fourniture de services de communications complémentaires. Les composants internes à un connecteur sont interconnectés par des connecteurs et des protocoles moins complexes.

Un connecteur, qu'il soit simple ou complexe, peut offrir un certain nombre de services permettant d'améliorer les communications ou nécessaires pour un protocole bien particulier. A titre d'exemple, un connecteur peut être doté d'une mémoire tampon dans laquelle les concepts à transporter sont mis en file d'attente.

Les travaux de Bures [8] se sont concentrés sur la mise au point de styles d'architecture pour quatre types de connecteurs d'interaction, représentant chacun un style de communication : appel de procédure (locale ou distante), message, flot de données et tableau noir. Les modèles d'architectures de connecteurs proposés permettent de spécifier un style de communication et de fixer les propriétés non fonctionnelles que doit prendre en charge un connecteur. Le modèle prend ses concepts de base des infrastructures de communication dite commerciale (Middleware), exploite ces infrastructures et peut être pris en charge par des outils d'automatisation de la génération de code des connecteurs.

III. NECESSITE DE MODELE DE CONNECTEURS COMME ELEMENT DE BASE DES ADL

La disponibilité au niveau des ADL de modèle de connexion et de connecteurs est d'une importance capitale pour les architectes de logiciels. L'analyse du raisonnement d'un architecte dans le processus d'élaboration d'une architecture permet de déceler ce besoin au niveau des outils de spécification.

Dans un domaine de conception bien précis, notamment le domaine structurel (ou architectural), un architecte commence toujours par porter de manière intense son attention sur la détermination des composants de bases qu'il combine ou configure selon une topologie bien précise permettant d'atteindre les objectifs de son système.

Lors de la définition des connecteurs, l'architecte se contente souvent d'affecter à ces derniers un rôle parfois vague, parfois précis et souvent il indique, même à un niveau très abstrait de l'architecture, une technologie d'interconnexion bien précise Dans ce dernier cas bien précis, l'architecte a résolu dès les premières heures de l'existence d'un logiciel, le problème de raffinement du connecteur.

IV.DEMARCHES DE MISE EN EVIDENCE DES MODELES DE CONNECTEUR

La détermination de modèle de connecteurs ne peut être réalisée sans une compréhension très fine du concept de connecteurs dans les architectures logicielles. La compréhension des connecteurs mène certainement dans un premier temps à les organiser. Elle permet de déterminer et de comprendre les liens pouvant exister entre les connecteurs dans les différents domaines de conception. Pour atteindre ce but, nous préconisons une démarche qui se base sur les trois actions suivantes :

1- Le recensement des différents domaines de conception des connecteurs
2- La détermination de l'espace de mise en œuvre des connecteurs
3- Le recensement des principales techniques d'interconnexion et de connecteurs

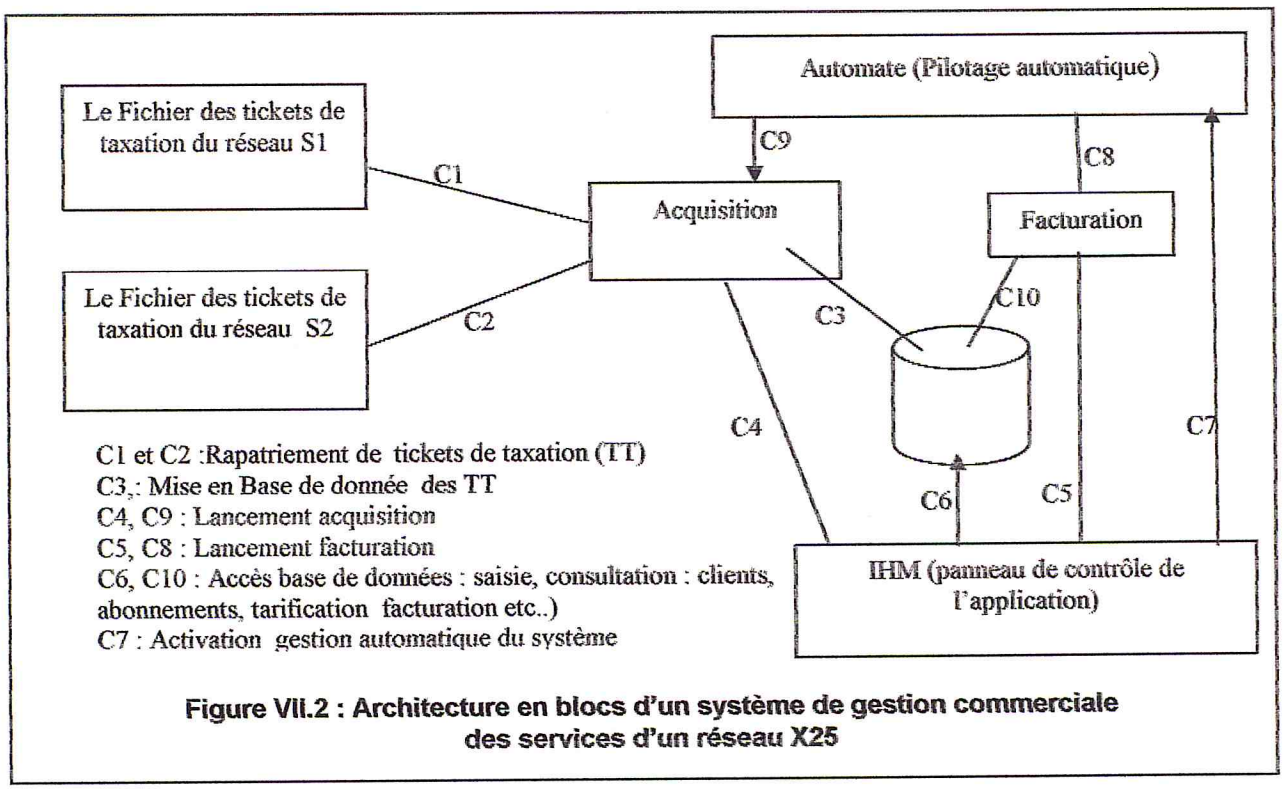


Figure VII.2 : Architecture en blocs d'un système de gestion commerciale des services d'un réseau X25

IV.1. Les domaines de conception des connecteurs

En architecture logicielle, tous les connecteurs ne sont pas des connecteurs d'interaction où il y a nécessairement échanges d'informations. A titre d'exemple, il y a des connecteurs de contrôle de l'implémentation, des connecteurs montrant une dépendance entre un composant et un autre. Ces connecteurs apparaissent dans un domaine bien précis de conception. Ainsi, les connecteurs d'interaction sont exploités dans un domaine où il est question de faire ressortir les blocs fonctionnels avec un flux de contrôle ou un flux de données. Les connecteurs d'implémentation comme les liens à résoudre, ou les concepts de déclaration/utilisation sont exploités dans l'organisation du code source, du code objet et des librairies.

Le travail réalisé par [7] représente une avancée certaine dans la voie de la compréhension des connecteurs en architecture logicielle. Nous pensons que la taxonomie définie dans ce travail devrait être renforcée par l'indication du domaine de conception dans lequel sera mis tel ou tel connecteur.

L'organisation des connecteurs par domaine de conception, nécessite au préalable la définition précise de ces domaines. Plusieurs travaux ont essayé de déterminer certains domaines de conception, mais à ce jour, il n'existe pas un consensus sur leur nombre ni l'objectif assigné à chaque domaine de conception. A cet effet, nous nous préconisons de recenser tous les domaines de conception cités en architecture logicielle.

IV.2 La détermination de l'espace de mise en œuvre des connecteurs

La compréhension des connecteurs nécessite la prise en considération de la position des composants dans une connexion : sont-ils très proches (dans un même exécutable), sont-ils un peu loin (même tâche), loin (même système d'exploitation), très loin (sur des machines différentes, au niveau des systèmes d'exploitation différents). Cet aspect a un impact certain sur le processus de choix d'alternatives de réalisation de connecteurs. Ce processus débute par le concept de connexion et se termine par une technologie bien précise de connecteurs. A titre d'exemple, vers quelle technologie sera résolu le connecteur C1 de la figure 2. Si le fichier à rapatrier est assez loin (sur un hôte distinct de l'hôte sur lequel sera déployé le composant d'acquisition), le connecteur correspond dans le domaine d'implantation à une des technologies suivantes : FTP, HTTP, RCP, etc. Si le fichier n'est pas très loin, sur une machine distincte mais dans le contexte d'un système de fichier distribué (NFS, SMB, etc.) par exemple, alors le connecteur peut correspondre à un accès aux données simples ou aux technologies précédentes.

Le choix de l'une des alternatives de réalisation d'un connecteur abstrait dépend certainement de l'aspect position des composants et bien sur d'autres objectifs imposés par l'environnement A titre d'exemple, dans le projet décrit par la figure 2, le connecteur C1 doit aussi être capable d'indiquer à la station de supervision de créer un nouveau fichier dans lequel elle met les nouvelles consommations, transfère le fichier de consommation courant, vérifie que le transfert s'est déroulé correctement et élimine le fichier transféré de la station de supervision. Ces actions ne pourront se faire que si le connecteur est de type HTTP ou RCP.

IV.3 Le recensement des principales techniques de connexion et de connecteur

Une compréhension de la connectivité en architecture logicielle permet le recensement des principales techniques d'interconnexion (ou liaison), la détermination des types de connecteurs capables de supporter ces interconnexions et la détermination des liens entre ces différents connecteurs. Ce recensement permet d'arriver à élaborer l'architecture des différents connecteurs.

Cette démarche nous mène certainement à déterminer qu'un connecteur complexe, disons « intelligent», sera certainement bâti sur les services de connecteurs moins complexes (moins « intelligents »). Une telle vue de l'architecture des connecteurs est nécessairement du style architecture d'interconnexion en couches. Ce type de vue nous fait rappeler les architectures en couches dans les réseaux de communication.

V. ELABORATION D'UN MODELE EN COUCHE POUR LES CONNECTEURS

L'élaboration de notre modèle de connecteur a été guidée par la nécessité d'atteindre entre autres les objectifs principaux suivants :

- Le modèle doit être totalement orienté vers une méthodologie de construction de systèmes par assemblage de composants et devra se doter de mécanisme permettant la prise en charge directe des modèles mentaux des architectes de logiciels
- Il doit être Fonctionnellement neutre vis-à-vis de toute architecture
- Il doit être totalement sure en ne fournissant, de par sa structure, aucun mécanisme implicite permettant à des composants d'accéder aux aspects internes d'autres composants. Le partage de propriétés et fonctionnalités entre composants devra se faire de manière explicite et sous la responsabilité totale des composants. Le connecteur devra fournir les mécanismes permettant de mettre en œuvre cet aspect de partage.
- Il devra être doté de capacités et facultés lui permettant d'associer une vue abstraite de connecteurs à diverses vues d'implémentations. Dans ce contexte le raffinement de la vue abstraite vers la vue concrète se réalisera selon des critères dictés par l'architecte à travers des propriétés non fonctionnelles (performance, sécurité, charge, déploiement etc.)
- Il ne doit pas être spécifique à style architectural précis
- Il doit reconnaître et prendre en charge directe les connecteurs standard (i.e. FTP, Bus CORBA, RMI etc..) et les infrastructures à objets distribués (EJB, CCM etc....)
  Le modèle d'interconnexion que nous sommes en trains d'élaborer et d'affiner est composé d'un niveau abstrait et de divers niveaux d'implémentation. Le nombre de niveaux d'implémentations qui seront mis en œuvre dépendra du type de spécification du connecteur au niveau abstrait.

C'est au niveau abstrait que seront spécifiées les caractéristiques qui auront une influence sur le choix du (ou des) connecteurs les plus adéquats pour le transports des divers concepts liés à des interactions entre composants. Parmi ces caractéristiques nous retrouvons diverses propriétés non fonctionnelles, l'organisation des interconnexions, les modes d'interaction, ainsi que les divers besoins en terme de services de communication tel que la résolution de nom et la localisation de composant.

Au niveau implémentation, le connecteur décrit le moyen d'acheminement des interactions. Une interaction peut être acheminé par un ou plusieurs connecteurs selon des modes de communication bien précis, dans des environnements de déploiement différents. Un connecteur agit sans aucun impact fonctionnel sur l'architecture.

```
Component Comp_01 {
Ports {
  p1 {required synchronous control in,
    provided integer out1, out2;
    peer float inout:; }
  } // end of External Component Ports
  Structure ; // Primary, Not Available
  Behaviour; // Not Yet Implemented
  Distribution {
    "src/c1.cc"; }
  Runtime ;// Not Available
}
```

```
Component Comp_02{
 Ports {
    p1 {provided synchronous control out,
        required integer in1, in2;
          peer float inout:; }
    p2 { provided float out
          required char in;  }
    p3  { peer integer inout;}
   } // end of Component ports
 Structure ; // Primary, Not Available
 Behavior; // Not Yet Implementerd
 Distribution {
       "src/c2.Comp_02"  }
  Runtime. ::Not Available
}
```

```
Component Comp_03 {
  Ports; //
 structure {
  use Comp_01, Comp_02;
  Comp_01 c1, c11, c12;
   Comp_02 c2n c21, c22,
   connect (c1.p1, c2.p1) with local
   connect (c11.p1, c21.p1) with unix_pipe;
   connect (c1.p12, c22.p1) with sun_rpc;  }
   Distribution {
         "src/Comp_03.cc"; }
   Runtime ; // Not Available
}
```

Figure VII.3 : Description des composants des figures 9 et 10

Le niveau d'abstraction peut influer sur l'ampleur du domaine de choix des connecteurs Figure 3. Ainsi pour une interaction trop abstraite (i.e. aucune indication sur le protocole de communication favoris, la proximité des composant etc..), le nombre de connecteurs pouvant assurer le transport pourrait être très important, ce qui n'est pas le cas lorsque au niveau abstrait nous retrouvons des informations spécifiant clairement des choix bien précis tels la nécessité d'utiliser un protocole de communication. Pour ce dernier cas le nombre de connecteurs capable de supporter l'interaction pourrait être très réduit.

Pour l'élaboration de notre modèle de connecteurs, deux grands aspects doivent trouver une solution, Figure 4.

- **Au niveau abstrait**, il est nécessaire de définir un langage de spécification (c'est l'application dans notre cas) des divers concepts guidant la mise en place d'un connecteur adéquat. Ce langage étant destinés aux praticiens, nous pensons qu'une forme de spécification à la CSP [9] représente une solution qui manquerait d'efficacité [10]. Dans le contexte de travail actuel, pour des raisons de validation nous utilisons un langage simple permettant de décrire les composants d'une configuration et les connexions.
- **Au niveau implémentation**, il est nécessaire de définir une architecture qui soit capable de donner à partir du niveau abstrait les connecteurs les plus adéquats pour supporter correctement les diverses interactions.

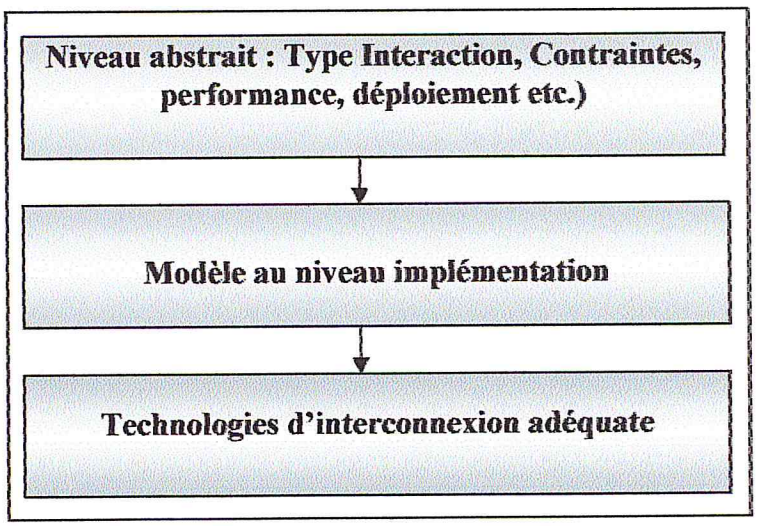


Figure VII.4 : Aspects pour l'élaboration du modèle de connecteurs

VI. UN MODELE EN COUCHES POUR LES CONNECTEURS LOGICIELS

L'idée de base sur laquelle repose le modèle de connecteurs que nous recherchons est la suivante : Dans n'importe quelle interaction entre composants nous distinguons les aspects suivants :

- Deux éléments fondamentaux sont échangée à travers les ports: *Le flux de contrôle* et *les données*. Les données représentent le cœur de toute interaction. Le flux de contrôle définira les composant en activité. Ainsi dans une définitions architecturale deux concepts seront cités : la connexion de port et l'activation/désactivation des composant (séquence de l'intervention des composants). C'est ainsi que le modèle de composant [11] sur lequel se base une

partie de la définition de notre modèle de connecteur, offre trois catégories de ports: Les ports de données (i.e. port de données à transférer, ports d'états, port log etc..), les port de services (i.e. nom méthode) et les ports de contrôle (activation/désactivation d'un service, d'un composant etc..).

- Pour l'échange des données ou le transfert de contrôle, les ports peuvent exiger de manière implicite ou explicite divers services du connecteur (mode d'interaction synchrone, asynchrone, fiable, localisation de services etc....)

Dépendant de la position des composants en interaction, les connecteurs à mettre en œuvre peuvent être plus ou moins complexes. Ainsi, pour acheminer une donnée vers un ou plusieurs destinataires, un certain niveau d'intelligence et un certain nombre de services devront être mis à la disposition de la donnée. A titre d'exemple, une communication entre deux composants très proches pourrait être réalisée à l'aide d'une variable commune située dans un espace de nomination commun. Elle pourrait par contre nécessiter beaucoup d'intelligence et plusieurs services si un composant voulant se connecter à un autre, ne connaît pas au préalable l'endroit de cet autre composant

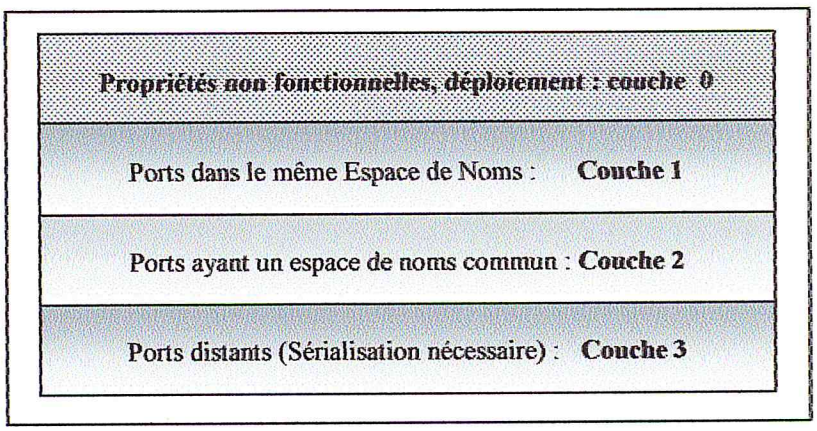


Figure VII.5 : Les couches de modèle de connecteurs

La figure 5 schématise une approche en couche de l'architecture des connecteurs. Chaque couche correspond en fait à un espace de nomination qui situerait la position des ports de composants et à partir duquel une classe de connecteurs adéquate pourrait être définie. Ainsi chaque niveau correspond à un type de distance séparant les ports de composants. Les couches se distinguent l'une de l'autre par le type de ressources et fonctionnalités qu'elles offrent [12]. Un service de couche supérieure est bâtie en utilisant les services et ressources des couches inférieures.

VII. ELEMENTS D'ARCHITECTURE POUR LE MODELE DE CONNECTEURS EN COUCHES

Dans le modèle en couche, un connecteur est constitué d'au moins deux adaptateurs et d'un conduit ou canal. En plus des adaptateurs, un connecteur peut disposer ou accéder à diverses facilités permettant de gérer le transport des interactions selon divers style d'interconnexion et selon les exigences spécifiées dans les propriétés non fonctionnelles. Le rôle principal des adaptateurs est de cacher aux composants l'aspect éloignement. Les figures 6 et 7 présentent deux topologies d'interconnexion de

base : un connecteur simple et un autre partagé. Dans ces deux types de connecteurs, ou les adaptateurs sont liés directement, les services complémentaires de gestion de l'interconnexion se situent au niveau adaptateur et canal.

La figure 8 montre une situation dans laquelle le connecteur est représenté par une infrastructure. Cette infrastructure peut disposer de services très complexes qui pourraient être mis en œuvre pour assurer l'interaction entre composants. Les composants d'un tel connecteur sont dédiés aux opérations de transport de l'interaction.

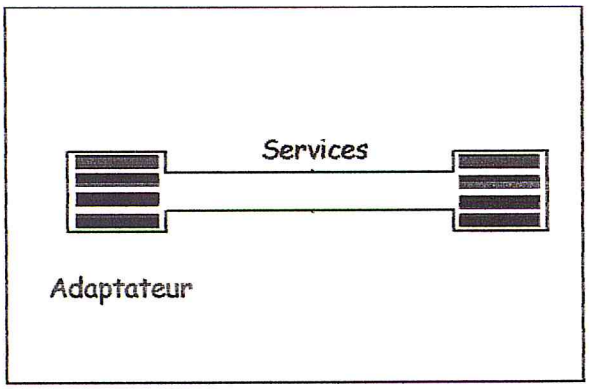


Figure VII.6 : Connecteur point à point

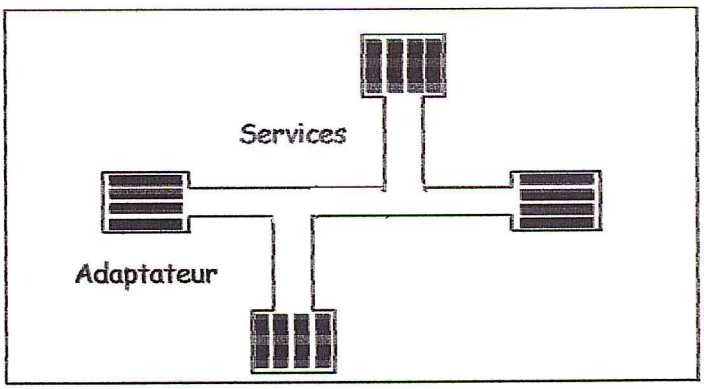


Figure VII.7 : Connecteur multipoint

Dans la logique d'interconnexion de composant avec le modèle en couches, un ou plusieurs ports à connecter sont des ports dit de niveaux 1. Le processus d'interconnexion doit les pourvoir d'adaptateurs adéquats pour qu'ils puissent s'interconnecter. Pour gérer des situations de déploiements plus ou moins complexes, plusieurs niveaux d'adaptateurs seront utilisé en commentant par le niveau 1.

Le modèle de connecteur en couche s'adapte aisément pour l'interconnexion de port de composant non conçu dans l'esprit du connecteur en couche. Le processus d'affectation des adaptateurs à ces ports débutera par une opération de mise à niveau qui consiste à mettre les ports au niveau le plus proche soit en les rétrogradant à un niveau inférieure soit en les haussant à un niveau supérieure [12]. Ce concept de mise à niveau de port vient du fait que les ports de composants défins dans un autre contexte que le contexte actuel, renferment déjà certains aspects d'un niveau bien précis.

Si à partir de notre modèle, il serait possible de définir toute une infrastructure, en pratique de telles infrastructures existent et il n'est pas judicieux d'en construire d'autre, sauf nécessité. Une infrastructure existante serait intégrée et exploitée par le biais des *composants d'interconnexion* dont le rôle principal est de cacher les spécificités de l'infrastructure (Figure 8).

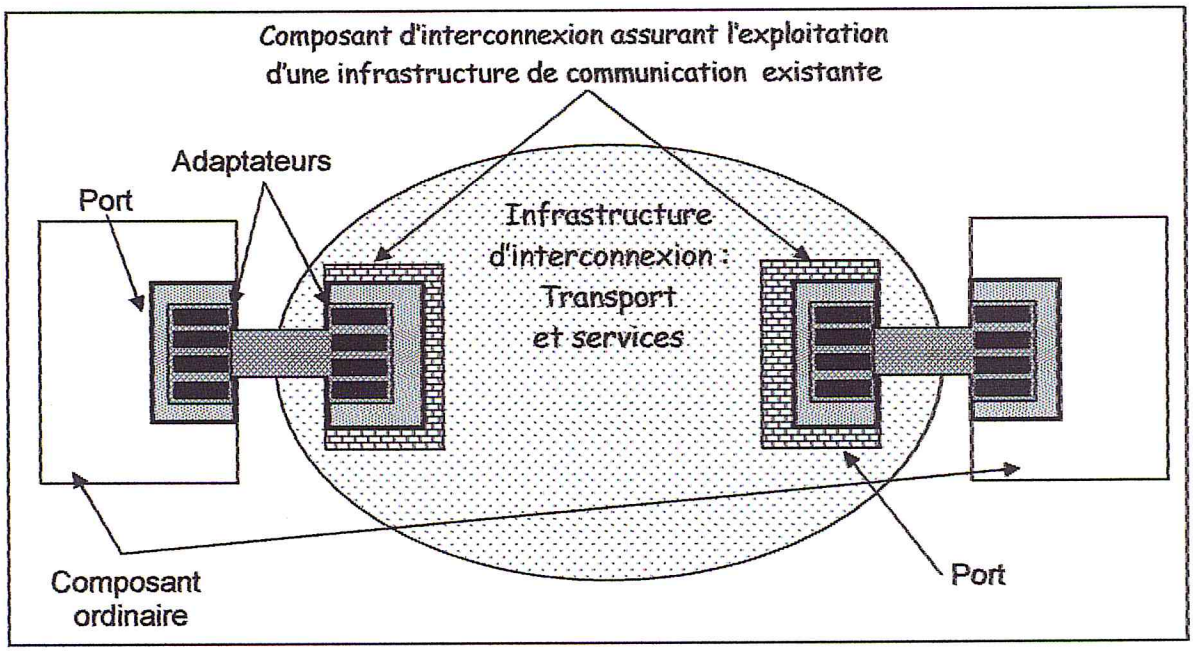


Figure VII.8 : Connecteur exploitant une infrastructure d'interconnexion

Les composants en connexion opèrent comme s'ils étaient interconnectés selon les modèles des figures 6 et 7. La connexion composant ordinaire, composant d'interconnexion est une connexion qui obéit totalement au modèle de connecteur en couches.

VIII. PROPRIETES FONDAMENTALES DU MODELE DE CONNECTEURS

Un ensemble de propriétés et règles de construction et d'exploitation du modèle de connecteur en couches **(Figure)**, ont été définis. Une présentation complète est donnée en [12]. Dans ce qui suit, nous en présentons un échantillon.

- **Connecteur**: Un connecteur est formé par la liaison d'au moins deux adaptateurs. Un connecteur à deux adaptateurs est appelé connecteur point à point et celui mettant en œuvre plusieurs adaptateurs sera appelé un connecteur multipoint.
- **Point de connexion :** Un adaptateur (de même pour un port) est composé d'un ou plusieurs points de connexion. Nous distinguons trois catégories de points de connexions ; les points de données, les points de services, les points d'activation. Les points de données regroupent les points de données échangées, les points d'états de composants et les points log. Dans une définition architecturale un point de données peut être lié à un autre point de données selon les règles qui seront énoncé par la suite ou peut être mis à une constante.
- **Type du point de données :** Un point de connexion est d'un type bien défini. Le type indiquera la nature de l'information qui passera par ce point.
- **Nature du point de connexion :** Un point de connexion peut être une entrée (require), une sortie (provide) ou une entrée sortie (ProvideRequire ou peer).
- **Port (similaire pour adaptateur) :** Un port (adaptateur) est un ensemble de point de connexions agissant ensemble dans le contexte d'une même interaction.
- **Statut de ports (adaptateur) :** Un port peut avoir l'un des statuts suivant : server, client, peer. A chaque statut correspond une forme d'interaction bien précise et des règles de connexion. Ainsi ; à titre d'exemple, un port serveur peut être raccordé à un ou plusieurs clients.

- **Processus d'interconnexion de composants :** Le processus d'interconnexion de composant se fera en deux grandes étapes successives : Affectation des adaptateurs aux ports, Liaisons des adaptateurs.
- **Sens de l'interaction** Un connecteur peut autoriser le transfert de données dans un seul sens (directionnel) ou dans les deux sens (bidirectionnel).
- **Point de connexion compatible (connectable) :** Ce sont deux points ayant le même type de données et de nature compatible. Un point provide est compatible avec un point require, et un point peer est compatible avec un point peer
- **Connecteur élémentaire**: Il est dédié à connecter deux point de connexion de même type et compatible.
- **Association point de connexion et niveau de connecteur:** De par sa constitution et les règles imposées sur leur définition, les point de connexion et les ports peuvent être d'un niveau de connecteur bien précis. A titre d'exemple un port dont les points de connexion ne sont que des références est un port de niveau. Un point de connexion représentée par une variable est un point de niveau 2
- **Connecteur élémentaire Multipoint**: Tous les points sont de mêmes types. Quand à leur statut, nous avons les possibilités suivantes :
  - Tous les points de connexions sont des points peer (qui fournit),
  - Un seul point est peer et tous les autres des require (qui demande),
  - Un seul point provide et tous les autres des require.
- **Composition des connecteurs :** Un connecteur peut être composé de 1 ou plusieurs connecteurs élémentaires. Un connecteur composé de plusieurs connecteurs élémentaire peut acheminer en parallèle plusieurs valeurs.
- **Adaptateurs compatibles :** Deux adaptateurs ou plusieurs adaptateurs sont compatibles s'ils peuvent être liés pour former un et un seul connecteur
- **Règle de construction de connecteur :** les règles de construction de connecteur indiquerons pour un connecteur d'un certain niveau quels seraient les types de composants et connecteurs qui pourraient être utilisé dans l'architecture d'un connecteur complexe.
- **Boucle :** Une boucle est un connecteur qui relie les points de ports d'un même composant

Organisation des interactions : À la base des concepts précédents, diverses organisations des interactions peuvent être imaginées. A Titre d'exemple, dans une interaction synchronisée par un port (donc par un composant), ce dernier pourra jouer le rôle d'un port maître et les autres seront des ports esclave. Dans une autre organisation totalement asynchrone, un port peut être source de données à n'importe quel moment. Dans ce cas le canal deviendra une ressource critique dont l'exploitation devra être gérée efficacement. Entre ces deux situations extrêmes diverses organisations des interactions entre composant pourraient être imaginées.

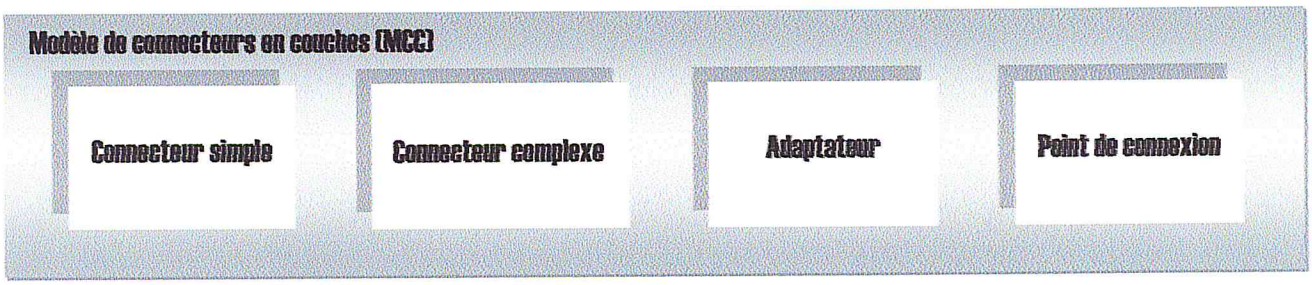


Figure VII.9 : Eléments fondamentaux du modèle de connecteurs en couches.

IX. ETUDE PAR L'EXEMPLE DU MODELE EN COUCHE

Dans cet exemple nous présentons des cas de figures qui vont illustrer la mise en place de connecteurs plus ou moins complexes.

Dans la figure 10, nous présentons deux composant C1 et C2. C1 possède un port de communication doté de 4 points de connexion dont seule la nature est citées sur le schéma. Le composant c2 possède 3 ports, l'un possédant 4 points de connexions, un autre en possède 2 et le troisième possède un seul point de connexion. La description de C1 et C2 est donnée en figure 3.

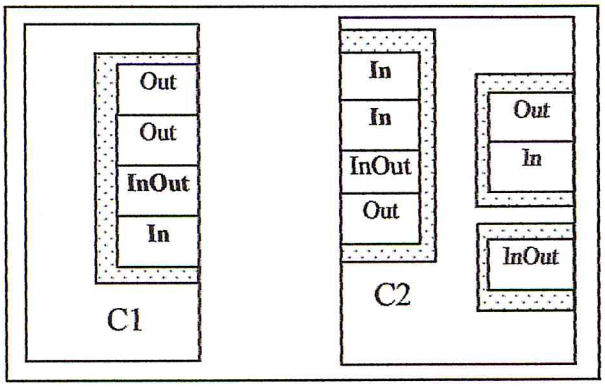


Figure VII.10: Représentation de deux composants

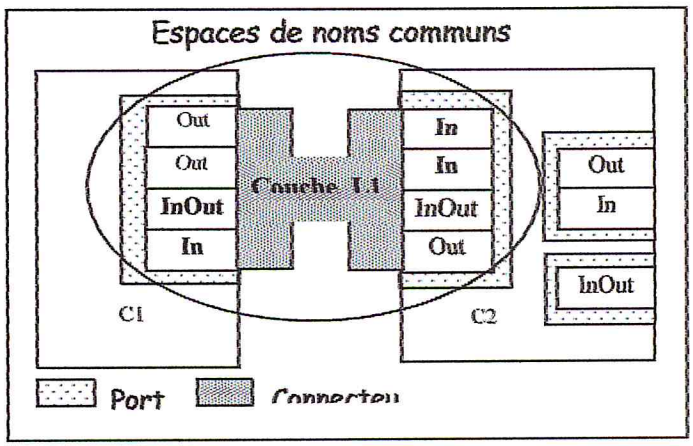


Figure VII.11 : Etablissement d'une connexion directe de niveau 1

Dans les cas de figures suivantes nous considérons que les ports de C1 et C2 sont des ports de niveau 1 pouvant être pris encharge directement par des adaptateur de la couche 1.. Les ports de niveau 1 ne doivent disposer d'aucune capacité de mémorisation au niveau de leurs points de connexion [12]

En figure 11, nous établissons une connexion directe de niveau 1 Le déploiement dans ce cas indique que les deux composants (plus précisément les deux ports) doivent évoluer dans un espace de nomination commun. Une forme assez simple d'implémentation consiste à utiliser un ensemble de variables partagées. Dans le cas de point de connexion bidirectionnels le connecteur pourra imposer certaines règles contrôlant les accès aux variables partagées représentant un connecteur bidirectionnel.

Le premier niveau offre des connecteurs dits directs entre composants très proches qui seront déployés pour évoluer dans une même classe d'objet, une tache (thread) ou un processus.

Pour des architectures point à point simples, dans lesquelles le transfert de contrôle est dans sa plus simple forme, l'accès au connecteur ne pose apparemment pas de problèmes cruciaux. Ceci n'est malheureusement pas le cas pou des connecteurs point à point ou multipoint établis dans des contextes de communication complexes ou le schéma global de transfert de contrôle permettrait des accès concurrents au connecteur. C'est conjointement au niveau adaptateur et canal que les accès concurrents aux connecteurs sont gérés.

Les connecteurs directs dans le contexte d'architecture simple point à point, offrent un nombre réduit de services, les plus important étant :

- La vérification que l'interaction n'enfreint pas les règles imposés par les connecteurs (compatibilité des points de connexions)
- La résolution de nom : les ports à connecter peuvent utiliser des identificateurs différents pour leurs points de connexion et pour les port. Ceci indique qu'un composant lors de sa conception n'est pas

obligatoirement dirigé pour s »interconnecter avec un composant bien précis

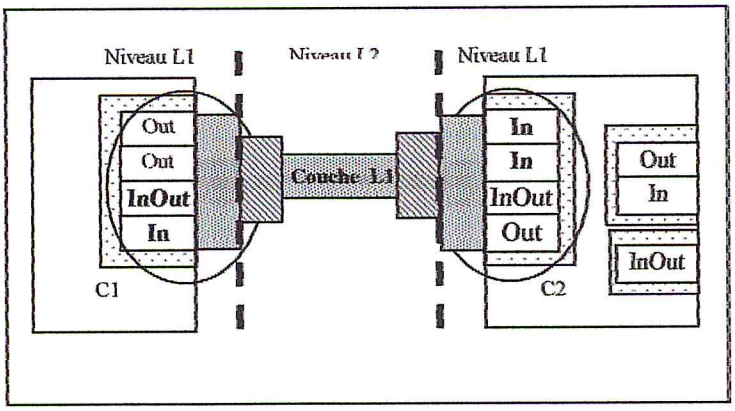


Figure VII.12 : Un cas simple d'interaction avec deux niveaux

La figure 12, illustre un cas simple d'interaction mettant en œuvre 2 niveaux. Les deux ports sont situés dans des espaces de noms différents et aucun espace de noms commun n'est défini.
Pour qu'une donnée issue d'un port arrive à destination, l'adaptateur de la couche 1, n'ayant pas de capacité pour faire arriver la donnée à destination, demandera les services de la couche 2. Il fera franchir à la donnée un niveau et la donnera au niveau 2. Le port doit être obligatoirement doté d'adaptateur de la couche 1 et de la couche 2. Au niveau 2 la valeur est véhiculée par des composant et connecteurs inaccessibles aux composants. Théoriquement, selon les services et les caractéristiques du niveau 2, la valeur peut prendre une forme différente de celle qu'elle avait au départ au niveau du port. Elle ne reprendra la forme de départ (ou une forme adéquate, que nous pourrons appeler forme de niveau 1) qu'à l'arrivée au niveau du port de destination. Le passage au niveau 2 peut être représentée à titre d'exemple par une opération simple de copie dans les variables internes du connecteur.
Dans le niveau deux les composant sont proches et peuvent mettre en œuvre des services plus élaborés que le niveau 1 Le niveau deux force le déploiement de composants pour qu'ils évoluent dans un même processus et dans un même thread ou des threads différents, mais dans des espaces de noms différents sans mise en œuvre d'un espace commun.

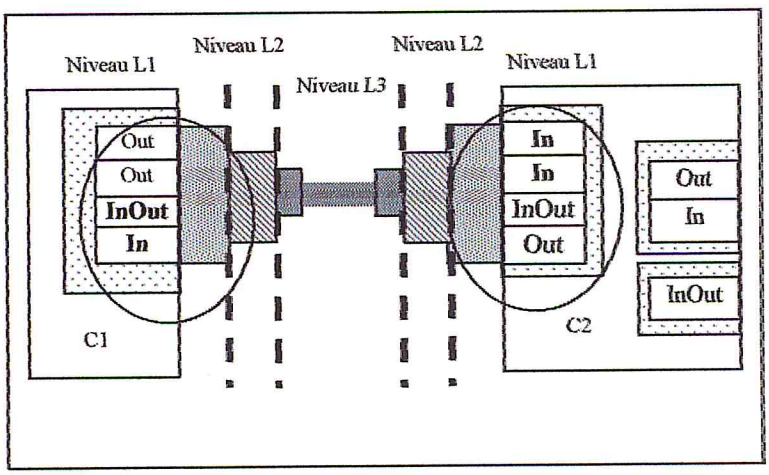


Figure VII.13 : Un cas de connexion en trois niveaux (même machine ou distante)

La figure 13, montre un cas, dans lequel, les composants sont distants. Le déploiement forcera les composants à évoluer dans des environnements totalement indépendants (processus différents sur même machine, composant évoluant dans des serveurs d'application différents, etc..).Les caractéristiques du niveau 2 ne seront plus suffisantes, vu que ce dernier n'est valable que pour des composants évoluant dans un même environnement. Un troisième niveau capable de lier des composants évoluant dans des environnements différents est nécessaire. Ce troisième niveau est composé des divers services fournis par les systèmes d'exploitation pour la communication interprocessus Parmi ces techniques, nous citerons les techniques de communication par mémoire partagée, les pipes, et les socket. En pratique, une grande partie des techniques de communication inter composant est basé sur les socket. Comme illustré sur la figure 13 les adaptateurs d'un connecteur de niveau 3 disposent des 3 niveaux de connexion.

Dans le niveau 3, les valeurs sont transformées selon les besoins de la technologie utilisée et les services complémentaires nécessités (compression, codage etc..). Le niveau 3 peut être représentée par une infrastructure complète de communication tels CORBA ou les JAVA RMI, comme il peut être représentée par un système de communication propriétaire basée directement sur les socket ou autre technique offerte par les systèmes d'exploitation.

X. CONCLUSION

Le modèle de connecteur en couches que nous venons de présenter assure une totale neutralité vis-à-vis de toute fonctionnalité et permet de gérer des schémas de déploiement complexe. Il s'inscrit dans une logique de construction totale de système par assemblage de composant et a été réfléchi pour qu'il puisse prendre en charge directement les spécification d'un modèle mental .Même si le modèle s'adapte facilement avec une modèle de composant spécifique, notamment au niveau de la définitions des ports (data, contrôle etc..).

Ces connecteurs se distinguent des autres connecteurs par deux aspects fondamentaux :

1- Bien que les autres solutions ne s'intéressent qu'a des connexions de données, notre modèle prend en charge l'aspect échange de données et l'aspect transfert de contrôle Cette approche ouvre grande la voie vers la définition d'architectures simples ou concurrentes selon divers schémas.

2- Le modèle intègre un aspect abstrait de connecteur et un abstrait implémentation. L'aspect abstrait peut être résolu selon le modèle en couche en plusieurs variantes d'implémentation et de déploiement (utilisation d'une interface graphique).

Bibliographie

[1] Dj. Bennouar: « Vers la définition d'un modèle de connecteurs logiciels pour la description de systèmes complexes », internal report LDRSI Lab, Computer Science Department, Saad Dahlab University at Blida, Algeria.

[2] N. Medvidovic, R. N. Taylor, A Classification and Comparison Framework for Software Architecture Description Languages, IEEE Transactions on Software Engineering, Vol. 26, no1, pp. 70-93, January 2000.

[3] M. Oussalah, T . Khammaci et A. Smeda, Expliciter les connecteurs dans les architectures logicielles, Journée CNRS-GDR-ALP-OCM, Langages et modèles à Objets (LMO'04), Lille, mars 2004.

[4] R. J. Allen, A Formal Approach to Software Architecture, PHD Thesis, May 1997, CMU-CS-97-144.

[5] R. Allen, D. Garlan, A Formal Basis for Architectural Connection, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA 15213, September 1998.

[6] N.R. Mehta, N. Medvidovic, S. Phadke, Towards a Taxonomy of Software Connectors, Proceedings of ICSE2000, Limerick, Ireland, May 2000.

[7] N.R. Mehta and N. Medvidovic, Understanding Software Connector Compatibilities Using A Connector Taxonomy, Computer Science Department University of Southern California, Los Angeles, CA 90089-0781, USA.

[8] T. Bures, F. Plasil, Scalable Element-Based Connectors, Proceedings of SERA 2003, SF, USA, June 2003.

[4] R. Allen, D. Garlan, A Formal Basis for Architectural Connection, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA 15213, September 1998.

[10] O. Barais, L. Duchien, TranSAT: Controlling Software Architecture Evolution, Langages et Modèles à Objets, Lille, march 2004 (In french).

[11] Dj. Bennouar, "fundamentals concepts of the layered connector model", Internal Report, LDRSI Lab, Computer Science Departement, Saad Dahlab University at Blida,; Algeria, Feb. 2005.

[12] Dj. Bennouar A Component Model for the Design of Software System by Assembling Components, internal report LDRSI Lab, Computer Science Departement, Saad Dahlab University at Blida; Algeria, Dec. 2004

Partie III

Conception et Réalisation

Chapitre VIII La conception

Notre démarche de conception consiste à définir l'intérêt et l'objectif du modèle où on découvrir les concepts du modèle et présenter les grands axes du projet, qui se présentent en : Un cahier des charges contient le contexte, prise de connaissance, compréhension du modèle, les consignes de développements et le choix de l'environnement de développement et d'implémentation.

Et une méthode de travail qui explique les étapes : Mise en place de l'interface homme machine, Intégration des fonctionnalités, L'utilisation et la sauvegarde d'information.

Enfin, une conception et modélisation UML de l'application fait par un ensemble de diagrammes de classes.

I. INTRODUCTION

Après une étude théorique sur les différentes notions associées au modèle en couches de connecteurs, on arrive à la deuxième étape qui est la conception de l'application **MCC-CASIC**.

L'application que nous somme entraîne d'étudier et de concevoir, est un outil graphique permet la validation des architectures logicielles.

Dans le cadre de ce projet, le travail sollicité a fait appel à des acquis liés au développement en java, et notamment aux classes graphiques (Swing, AWT, et personnelles) ainsi qu'à la compréhension des principes fondamentaux du modèle en couches de connecteurs.

En effet, c'est en partant de l'explication de Intérêt et objectif de ce modèle, puis les grands axes du projet, et on arrive à la conception et modélisation UML de l'application.

L'idée principale étudiée est la connectique en architecture logicielle qui à un importance capital. C'est grâce à ces connecteurs qu'il est possible de spécifier et définir une architecture dépend de la manière dont les interconnections sont spécifiés.

L'architecture aura un sens lorsque les connecteurs seront définis (figure).

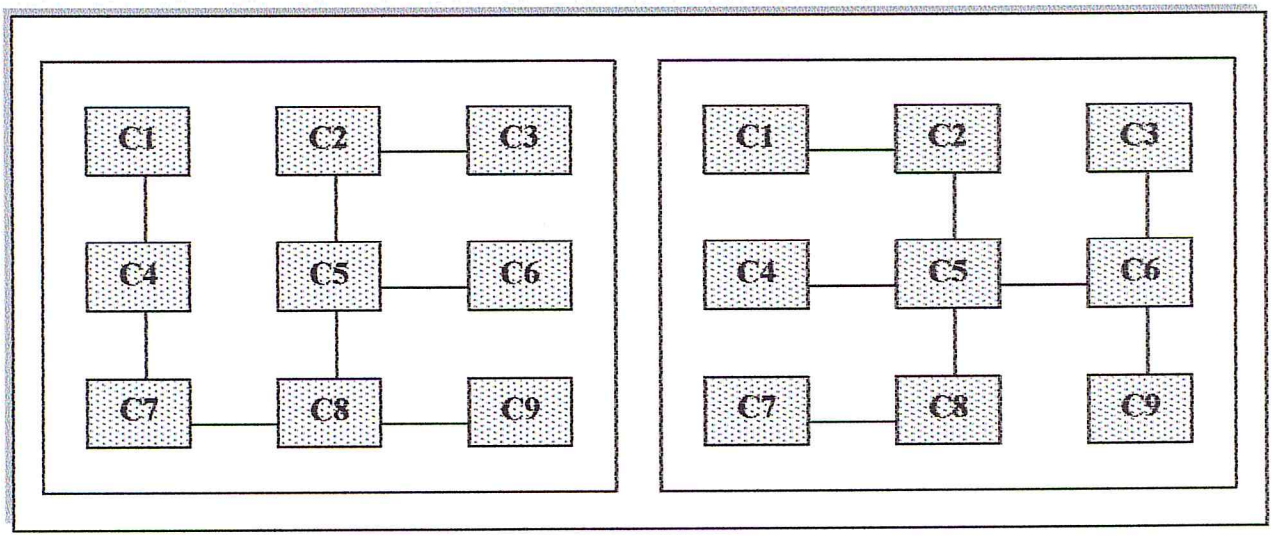


Figure VIII.1: la différence entre ces deux architectures réside dont la façon avec laquelle les composants sont interconnectés.

L'application que nous avons à concevoir, présente une forte interactivité avec l'utilisateur, et orienté vers le traitement graphique. En plus, c'est une application qui met en oeuvre une interface de haut niveau dont la conception fait appel à des standards d'environnement. Le développement de ce type d'application nécessite l'utilisation d'une méthodologie de conception apte à prendre en compte ces spécificités. L'approche orientée objets apporte les concepts nécessaires et les outils de modélisation appropriés.

II. INTERET ET OBJECTIF DU MODELE REALISE

II.1 Présentation générale du modèle

L'ensemble de bas des informations relatives au modèle en couches de connecteurs provient principalement de document « *un modèle de connecteur en couches pour la conception par assemblage des systèmes informatique complexes* » présentée par **M.Djamel Bennouar**, responsable de notre projet.

Le modèle en couches de connecteurs s'intéresse à la définition d'une architecture de système Pour la description de la structure et du comportement d'un système logiciel complexe.
Dans une architecture logicielle, un système est représenté par un ensemble de composants logiciel, d'un ensemble de connecteurs simples ou complexes, et d'un ensemble d'interactions comportementales.
L'architecture du modèle en couches de connecteurs décrit un système en termes de types et d'instances. Les composants, les connecteurs et les configurations sont des types qui peuvent être instanciers pour construire plusieurs architectures logicielles. Les concepts de base de l'architecture du modèle en couches de connecteurs sont les composants, les connecteurs, les configurations, les interfaces, les contraintes et les propriétés.
Pour élaborer notre modèle, deux grands aspects doivent trouver une solution :

- Au niveau abstrait, il est nécessaire de définir un langage de spécification des divers concepts guidant la mise en place d'un connecteur adéquat, et en prendre en considération qu'il est destiné aux praticiens.
- Au niveau implémentation, il est nécessaire de définir une architecture qui soit capable de donner à partir du niveau abstrait les connecteurs les plus adéquats pour supporter correctement les diverses interactions.

Si à partir de ce modèle, il serait possible de définir toute une infrastructure. Une infrastructure de communication existante telle que CORBA serait intégrée et exploitée par le biais des *composants d'interconnexion,* et dont le rôle principal est de cacher les spécificités de celle-ci.

II.2 Découverte des concepts du modèle

II.2.1 Présentation du concept connecteur

Le connecteur est constitué d'au moins deux adaptateurs. Le cas de deux adaptateurs, il est appelé connecteur point à point, alors dans les autre cas il est appelé connecteur multipoint.
Il existe deux catégories d'interconnexion de base (Figure) :

- *Le Connecteur simple*
  C'est un connecteur élémentaire point à point ou multipoint dédié à connecter deux ou plusieurs points de connexion de même type.
- *Le Connecteur complexe*
  Il possède une architecture propre, elle est spécifie à base de sous-connecteurs et sous-composants.
  Cette composition peut correspondre à une infrastructure de communication bien précise.

➢ **Composition des connecteurs :**
Un connecteur peut être composé de un ou plusieurs connecteurs élémentaires. Un connecteur composé de plusieurs connecteurs élémentaires peut acheminer en parallèle plusieurs valeurs.
Un connecteur simple est composé de deux ou plusieurs adaptateurs ayant chacun un point de connexion.

➢ **Capacités des connecteurs :**
C'est un triplet indiquant le nombre de connecteur dans un premier sens, le nombre dans un autre sens, et le nombre de bidirectionnel.

➢ **Cardinalité d'un connecteur :**
C'est le nombre d'adaptateurs.

➢ Validité des connecteurs :

Pour l'instant, nous pouvons énoncer qu'un connecteur est valide si tous ses connecteurs élémentaires sont valides.

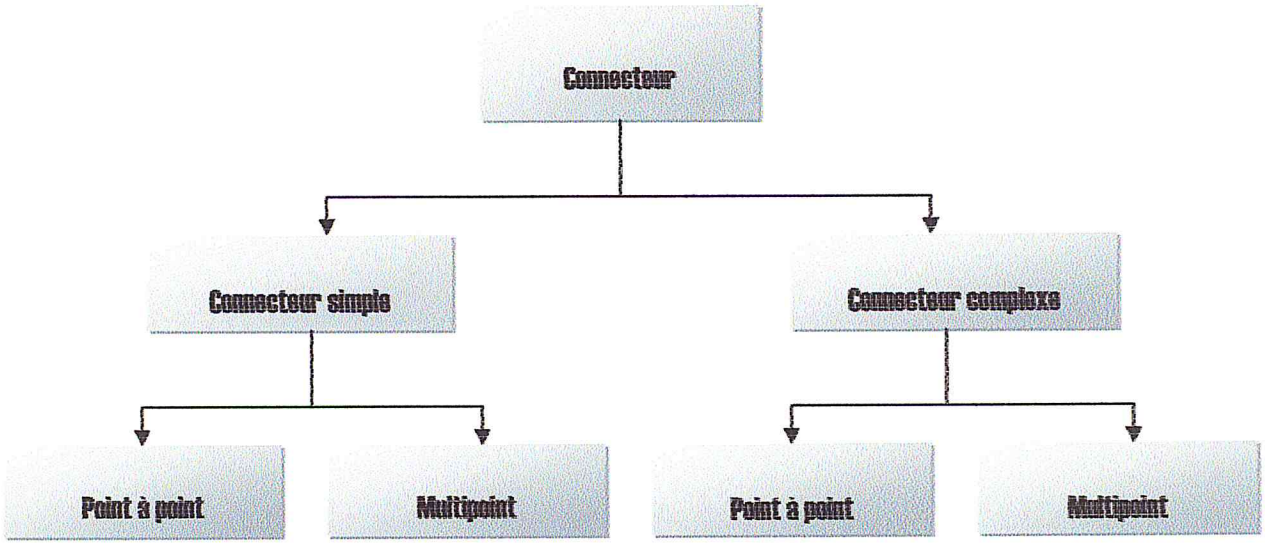


Figure VIII.2 : Hiérarchie de concept connecteur.

II.2.2 Présentation du concept adaptateur

L'adaptateur est la partie de connecteur qui se loge dans le port d'un composant. On dit qu'un deux ou plusieurs adaptateurs sont compatibles, s'ils peuvent être liés pour former un et un seul connecteur.

L'adaptateur est figuré en deux catégories :

- Adaptateur point à point.

Il est destiné à la topologie point à point

- Adaptateur multipoint.

Il est destiné à la topologie multipoint

➢ Validité d'adaptateur :

Un adaptateur sera valide s'il respecte les règles de regroupement de point de connexion

II.2.3 Présentation du concept point de connexion

C'est la partie principale du port, même l'adaptateur. Elle apparue en trois catégories : Les points de données, les points de services, et les points d'activation.

- **Type du point de connexion :**

Chaque point de connexion à un type bien défini. Qu'il indiquera la nature de l'information qui traversera ce point.

- **Nature du point de connexion :**

Elle peut être une entrée, une sortie, ou une entrée sortie.

- **Points de connexion compatible :**

Ce sont deux points ayant le même type de données et de nature compatible (un point «entrée» est compatible avec un point «sortie», et un point «entrée-sortie» est compatible avec un point «entrée-sortie»).

III. LES GRANDS AXES DU PROJET

III.1 Cahier des charges

III.1.1 Contexte

Afin de donner la possibilité à tout un chacun de pouvoir utiliser le modèle en couches de connecteurs au sein d'un environnement de développement des systèmes complexes, le développement de l'application a été lancé dans le cadre des Projets de Fin d'Etude des étudiants 5ème année au niveau de laboratoire LDRSI, département d'informatique, université Saad Dahlab à blida. L'outil MCC-CASIC est l'un de suite de logiciels suivis par M.Djamel Bennouar et ce rapport va donc en présenter les différentes spécifications.

III.1.2 Prise de connaissance

- *Connaissance du sujet* :

  La première phase du projet est la prise de connaissance avec le modèle en couches de connecteurs.

  Pour cela, une réunion a eu lieu le mois d'octobre 2004 avec M.bennouar, promoteur du projet, afin d'expliquer le thème du projet.

- *Définition des objectifs* :

  Les objectifs ont été définis lors de la première réunion, à savoir réaliser une application nommée MCC-CASIC qui peuvent exploiter et représenter toutes les notions du modèle étudié.

  Celui-ci doit permettre de créer, modifier ou supprimer graphiquement des éléments entrant dans la construction des configurations des architectures logicielles.

  En d'autres termes, elle offre la possibilité de décrire la sémantique opérationnelle d'une architecture qui sera utilisé pour la conception des systèmes plus ou moins complexes.

- *Limites du sujet*

  On ne s'intéresse pas à l'échange de flux de contrôle.

  On ne s'intéresse pas à l'étude des propriétés non fonctionnelles tels que la performance, plan de déploiement, la sécurité.

III.1.3. Compréhension du modèle

Le développement de L'outil MCC-CASIC nécessite de bien cerner le sujet et ses limites.

➢ Architecture du modèle en couches de connecteurs.

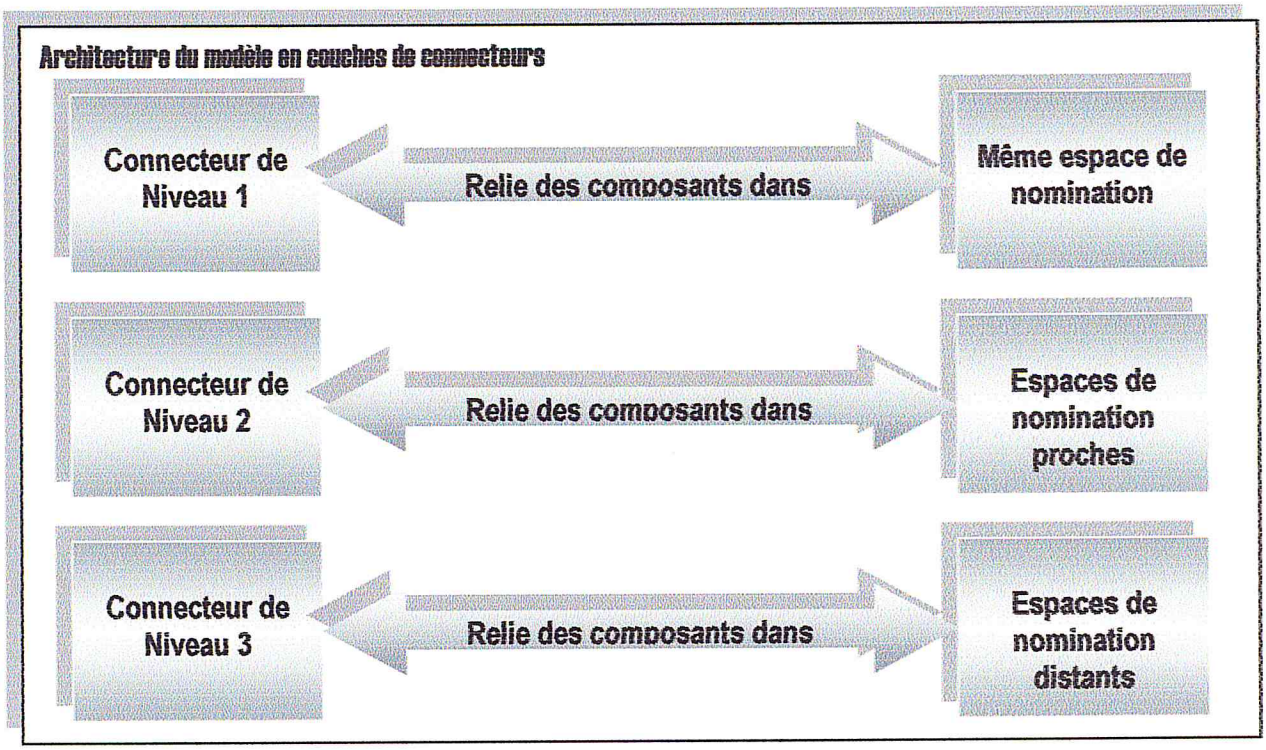


Figure VIII.3 : Architecture du modèle étudié.

➢ Principe du modèle en couche de connecteurs.

Avec la manipulation de l'application MCC-CASIC un utilisateur (Architecte) utilise une bibliothèque de composants réutilisables pour avoir une configuration validée d'un système (Figure).

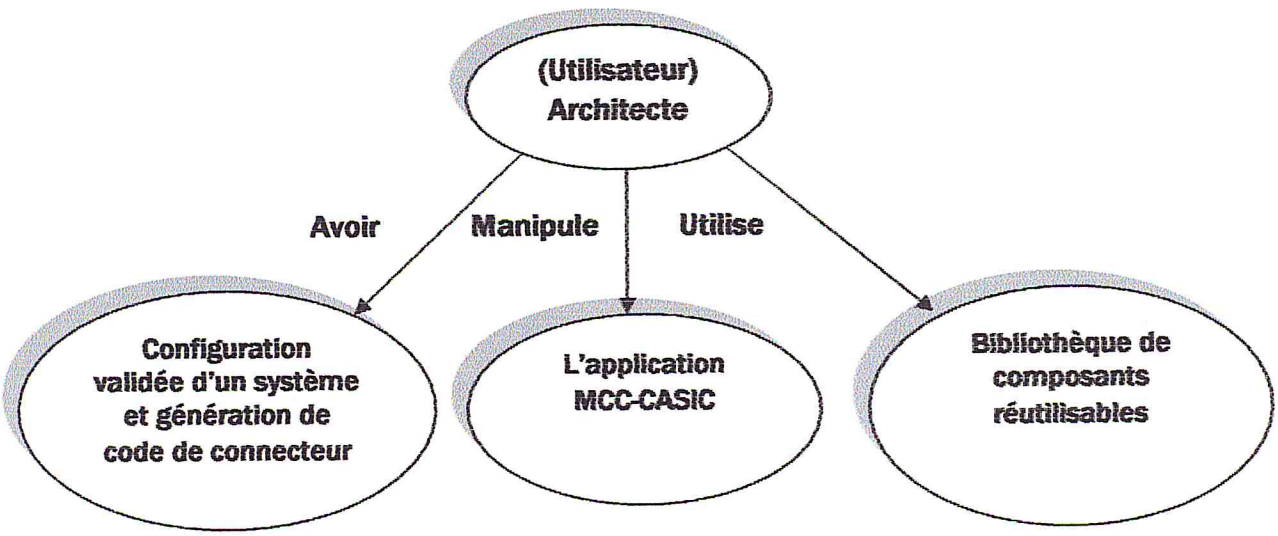


Figure VIII.4 : Principe du modèle étudié.

III.1.4. Les consignes de développements

En ce qui concerne le travail requis dans le cadre de notre projet, nous avons dû, durant le développement, suivre deux grands axes :

- Respecter les besoins précédemment décrits et

- Fournir un code source clair, commenté et réutilisable.

Il est toutefois nécessaire de préciser que l'objectif du projet n'était en aucun cas de terminer l'application. Nous devions simplement avancer le plus possible notre implémentation qui sera par la suite complétée et raffinée aux prochaines versions de l'application.

III.1.5. Choix de l'environnement de développement, implémentation

- **Conception**

La norme UML étant reconnue et standardisée par l'OMG, nous avons retenu celle-ci pour la partie modélisation de l'application MCC-CASIC, où on s'intéresse globalement sur les diagrammes de classes.

Pourquoi UML ?

UML fournit les fondements pour spécifier, construire, visualiser et décrire les artefacts d'un système logiciel, elle permet de modéliser de manière claire et précise la structure et le comportement d'un système indépendamment de toute méthodes ou de tout langage de programmation.

Les diagrammes qui seront réalisés dans le cadre du projet seront des diagrammes de classes.

Le support de modélisation est ArgoUML V0.16.1 [1].

- **Langage de programmation**

Le choix d'un langage orientés objets était quasiment inévitable, car les avantages de l'approche objet nous ramènent à l'adopter et l'utiliser, parmi ces avantages : la stabilité de la modélisation par rapport au monde réel, la construction itérative facilitée par le couplage faible entre composants et la possibilité de réutiliser des éléments d'un développement à un autre.

D'où, et d'un commun accord avec le promoteur du projet, le langage que nous avons choisi comme support de développement de l'application MCC-CASIC est Java.

Ce choix se justifie pour plusieurs raisons :

- **Distribué**

Java possède une importante bibliothèque de routines permettant de gérer les protocoles TCP/IP tels que HTTP et FTP, et Le mécanisme d'invocation de méthode à distance (RMI) autorise la communication entre objets distribués.

- **Portabilité**

L'application doit pouvoir tourner aussi bien sous MS Windows que sur stations type Unix ou linux.

- **Fiabilité**

Java a été conçu pour que les programmes qui l'utilisent soient fiables sous différents aspects. Sa conception encourage le programmeur à traquer préventivement les éventuels problèmes, à lancer des vérifications dynamiques en cours d'exécution et à éliminer les situations génératrices d'erreurs.

- **Orienté objet**

Pour rester simples, disons que la conception orientée objet est une technique de programmation qui se concentre sur les données (les objets) et sur les interfaces avec ces objets.

- **Simple**

  Nous avoirs voulu créer un système qui puisse être programmé simplement sans nécessiter un apprentissage ésotérique, et qui tire parti de l'expérience standard actuelle. En conséquence, même si nous pensions que C++ ne convenait pas, Java a été conçu de façon relativement proche de ce langage dans le dessein de faciliter la compréhension du système. De nombreuses fonctions compliquées, mal comprises, rarement utilisées de C++, alors java est Facile à programmer par rapport à un langage comme C++.

- **Sécurité**

  Java a été conçu pour être exploité dans des environnements serveur et distribués.

  Dans ce but, la sécurité n'a pas été négligée. Java permet la construction des systèmes inaltérables et sans virus.

- **Architecture neutre**

  Le compilateur génère un format de fichier objet dont l'architecture est neutre – le code compilé est exécutable sur de nombreux processeurs, à partir du moment où le système d'exécution de Java est présent.

- **Interprété**

  L'interpréteur Java peut exécuter les bytecode directement sur n'importe quelle machine sur laquelle il a été porté. Dans la mesure où la liaison est un processus plus incrémentiel et léger, le processus de développement peut se révéler plus rapide et exploratoire.

- **Performances élevées**

  En général, les performances des bytecodes interprétés sont tout à fait suffisantes, il existe toutefois des situations dans lesquelles des performances plus élevées sont nécessaires. Les bytecodes peuvent être traduits à la volée en code machine pour l'unité centrale destinée à accueillir l'application.

- **Multithread**

  Les avantages du multithread sont une meilleure inter-réactivité et un meilleur comportement en temps réel.

Enfin, nous avons choisi d'utiliser le logiciel **eclipse** (version 3.0.1) d'IBM [2]. Ce logiciel ne fournissant pas, à la bas un assistant de développement, nous avons codé l'ensemble des instructions manuellement. Pour compiler et exécuter l'application MCC-CASIC, nous avons utilisé le JDK 1.5.0.

- **Package graphique**

  La librairie graphique que nous avons choisie pour réaliser la partie graphique du logiciel est Swing. Cette librairie est en en effet très répandue, son utilisation est aisée et la documentation qui s'y réfère abonde.

- **Documentation**

  Les classes écrites sont commentées suivant la convention définie dans le Javadoc-tool disponible sur le site de Sun :
  http://java.sun.com/j2se/javadoc/writingdoccomments/index.html

III.2 Méthode de travail

Aux vues des instructions précédemment exposées, nous avons extrait trois grandes phases de développement pour mener à bien notre programme de travail, et c'est en respectant cette méthodologie que nous avons pu progresser :

III.2.1 Mise en place de l'interface homme-machine

Cette étape qui est la première chronologiquement consiste à mettre en place les classes graphiques et leurs écouteurs en prévision de l'insertion des fonctionnalités. Par rapport aux consignes données, c'est au cours de cette phase de travail que sera décrite la structure définitive de toutes les parties de l'application.

III.2.2 Intégration des fonctionnalités

Il faut noter que nous considérons comme fonctionnalités l'ensemble des opérations qui doit être mis à la disposition de l'utilisateur pour qu'il puisse faire évoluer la configuration d'un système dont il est en train de spécifier le comportement, par rapport à ses besoins. C'est donc durant cette étape que seront ajoutées toutes les fonctions de connectivité et de génération du code associé à cette configuration comme expliqué précédemment.

III.2.3 L'utilisation et la sauvegarde d'information

Cette dernière phase, qui devra constituer la fin du développement du logiciel, nous permettra d'insérer les fonctionnalités de sauvegarde des configurations créées. D'un point de vue graphique, c'est au cours de cette étape que l'ensemble des possibilités offertes à l'utilisateur dans le menu «Fichier» sera implémenté.
Une fois cette méthodologie établie, nous allons pouvoir nous attacher à présenter notre travail et l'implémentation que nous avons réalisée de l'application MCC-CASIC.

IV. LA MODELISATION UML DE L'APPLICATION (DIAGRAMMES DE CLASSES)

IV.1 Etude de Contexte initial

Avant de commencer à développer l'application MCC-CASIC, nous avons réalisé une modélisation de celle-ci selon la norme UML. Cette modélisation a été faite dès le début du projet de manière à ne pas perdre de temps et avoir une première maquette navigable de l'application MCC-CASIC. C'est la modélisation qui avait été initialement présentée dans le dossier de spécifications et qui nous a permis d'évoluer à partir d'une première maquette MCC-CASIC.

IV.2. Diagrammes de classes réalisés

Les différents diagrammes de classes que nous avons conçu puis fait évoluer au cours du projet, sont présentés ci-après.

IV.2.1. Diagrammes de classes associés au graphique de l'application

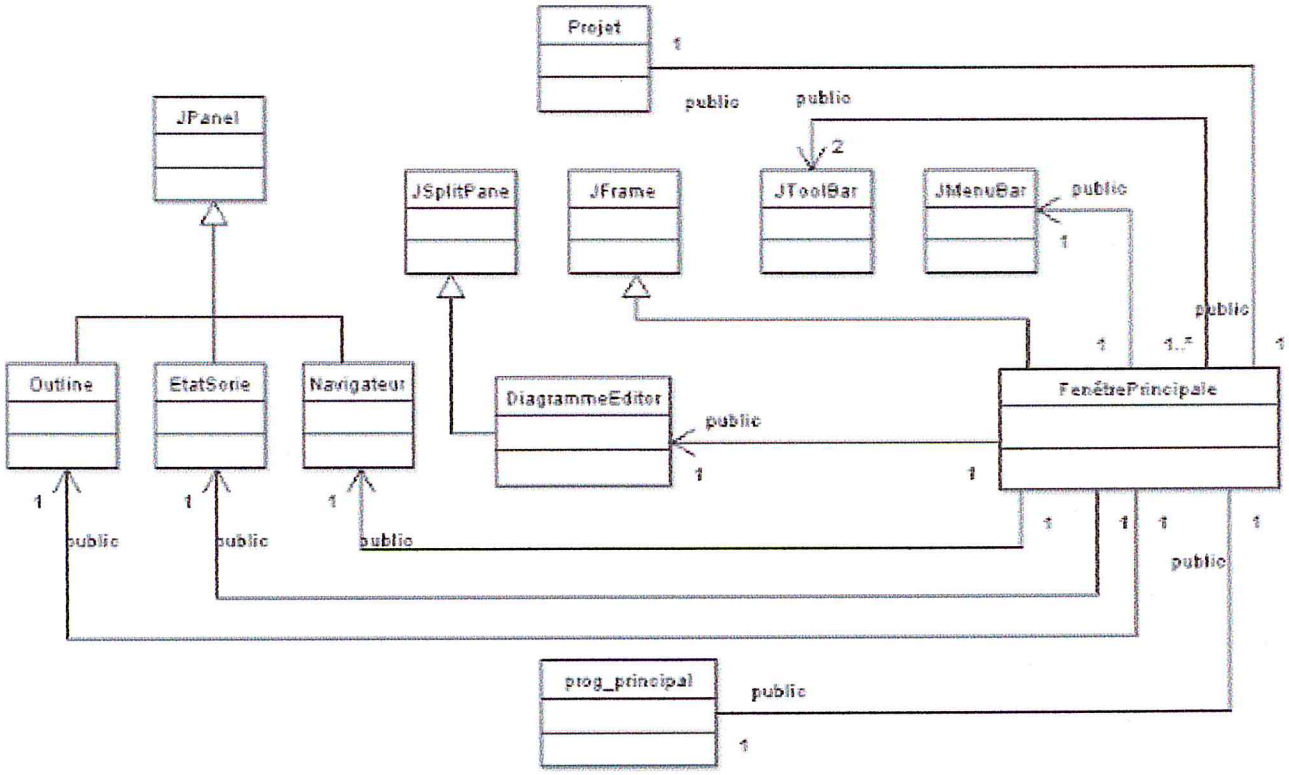


Figure VIII.5 : Diagramme de classes général

Description brève :

- **Prog_Principal :** Classe principale du l'application, contient le « main ».
- **FenêtrePrincipale :** Classe dérivant de javax.swing.JFrame, fenêtre de l'application.
- **Outline :** Classe de la représentation d'un arbre supportant les éléments d'une configuration.
- **Navigateur :** Classe de représentation d'un arbre supportant les projets lancés.
- **EtatSorie :** Classe chargée de la représentation de sortie, propriétés, et problèmes après validation.
- **DiagrammeEditor :** Classe représente la zone de dessin des configurations.
- **Projet :** Classe représentante d'un projet qui regroupe le panneau Navigateur.

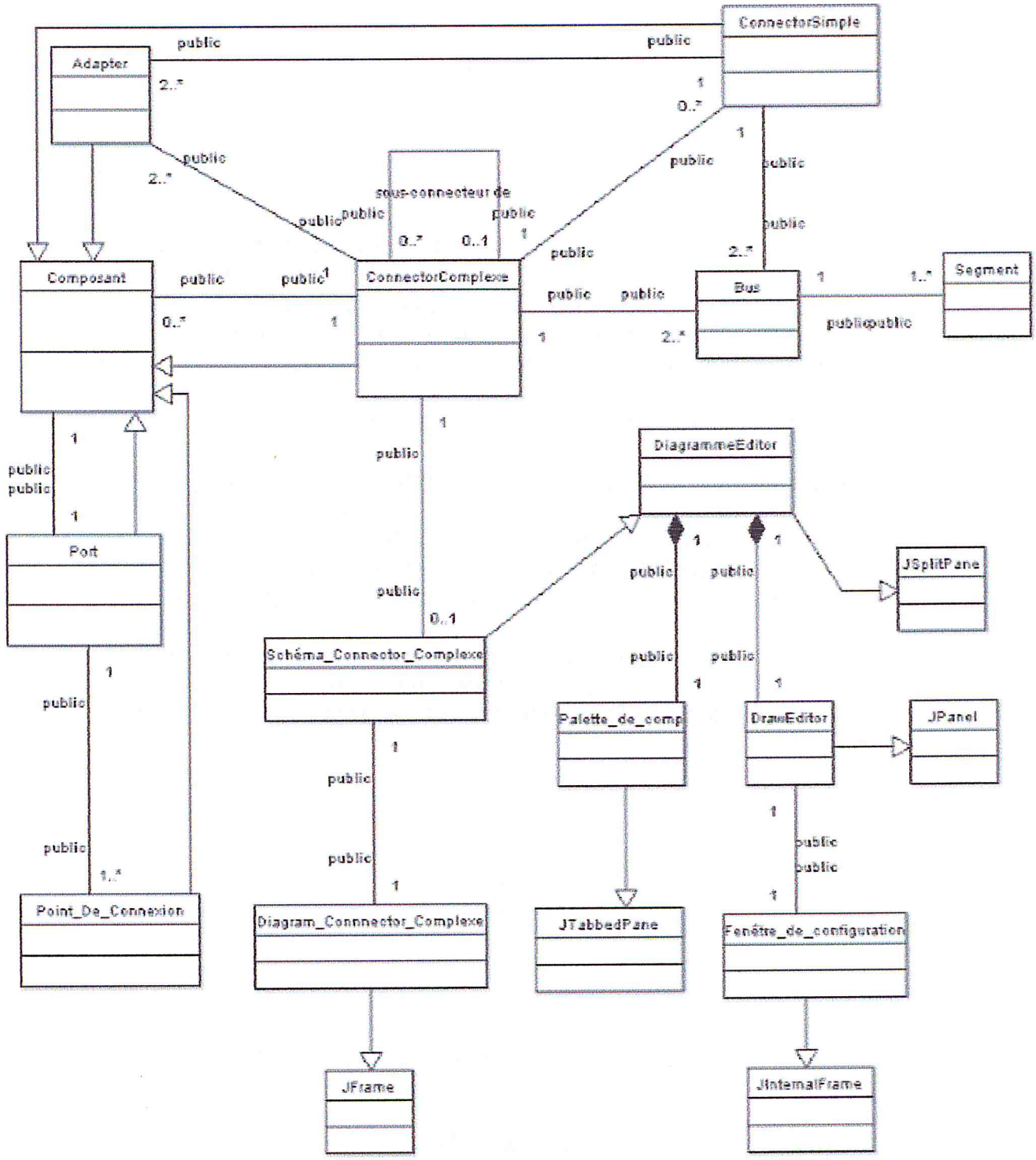


Figure VIII.6 : Diagramme de classes de l'éditeur de diagrammes (configurations).

Description brève :

- **DiagrammeEditor :** Classe représente la zone de dessin des configurations.
- **DrawEditor :** Classe représente le panneau qui supporte la Fenêtre de configuration.
- **Palette_de_comp :** Classe représente les outils de manipulation graphique des configurations.

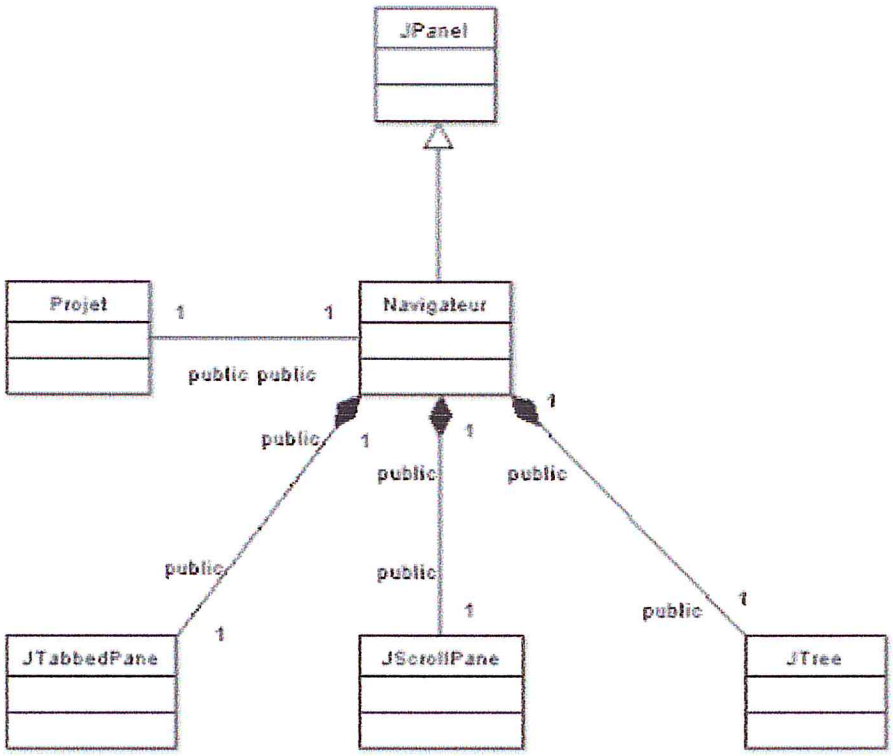


Figure VIII.8 : Diagramme de classes de la vue Navigateur.

Description brève :

- **Navigateur :** Classe de représentation d'un arbre supportant les projets lancés.
- **Projet :** Classe représentante d'un projet qui regroupe le panneau Navigateur.

- **Fenêtre_de_configuration :** Classe représente la fenêtre qui supporte les configurations.
- **Composant :** Classe chargée de la représentation graphique d'un composant.
- **Port :** elle est chargée de la représentation graphique d'un Port.
- **Point_De_Connexion :** Classe chargée de la représentation graphique d'un point de connexion.
- **ConnectorComplexe :** Classe chargée de la représentation graphique d'un connecteur complexe.
- **Schéma_Connector_Complexe :** Classe la zone de dessin des Schémas de Connecteurs Complexes.
- **Diagram_Connnector_Complexe :** Classe représente le Schéma d'un Connecteur Complexe, ou d'un sous-Connecteur Complexe.
- **ConnectorSimple :** Classe chargée de la représentation graphique d'un connecteur simple.
- **Adapter :** elle est chargée de la représentation graphique d'un Adaptateur.
- **Bus :** Classe chargée de la représentation graphique d'un Bus.
- **Segment :** Classe chargée de la représentation graphique d'un Segment.

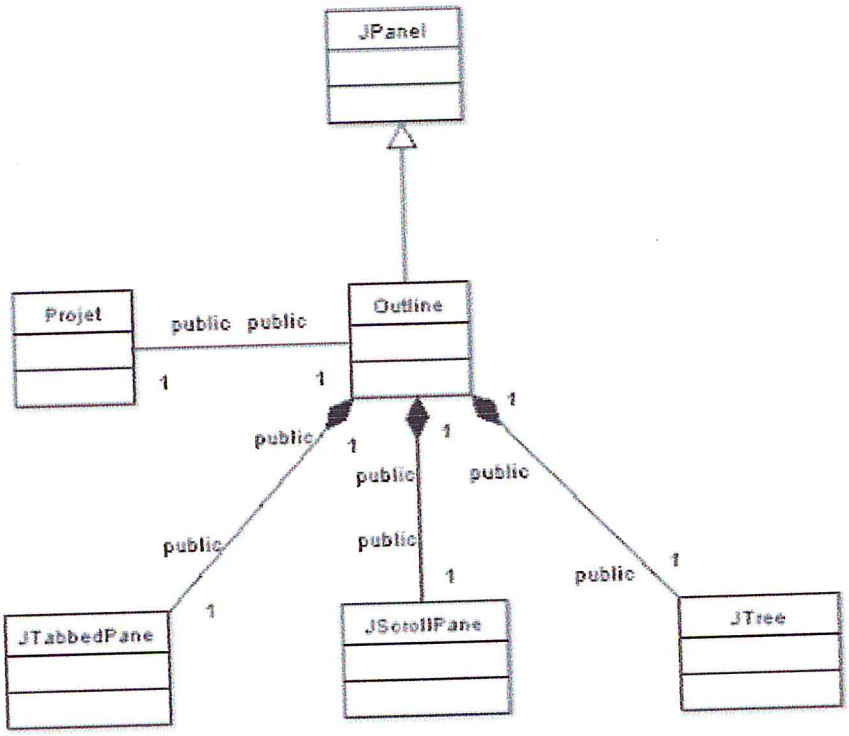


Figure VIII.7 : Diagramme de classes de la vue Outline.

Description brève :

- **Outline :** classe de la représentation d'un arbre supportant les éléments d'une configuration.
- **Projet :** classe représentante d'un projet qui regroupe le panneau Outline.

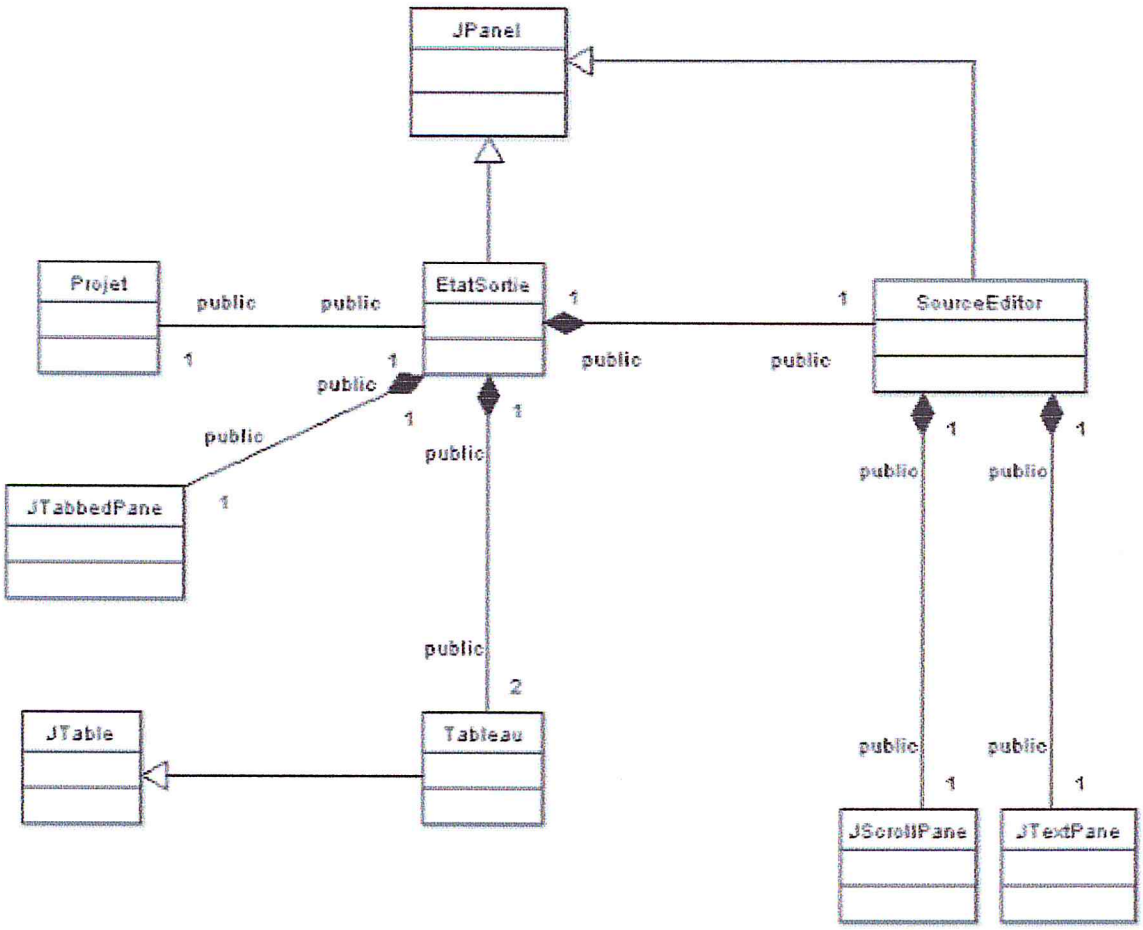


Figure VIII.9 : Diagramme de classes de la vue EtatSortie.

Description brève :

- **EtatSortie :** Classe chargée de la représentation des panneaux SortieGénérale, propriétés d'un élément, et problèmes après validation.
- **SourceEditor :** Classe représentante la zone d'affichage après validation.
- **Projet :** Classe représentante d'un projet qui regroupe le panneau EtatSortie.
- **Tableau :** Classe chargée de la représentation d'un tableau de propriétés.

IV.2.2 Diagrammes de classes associés aux concepts de base du modèle

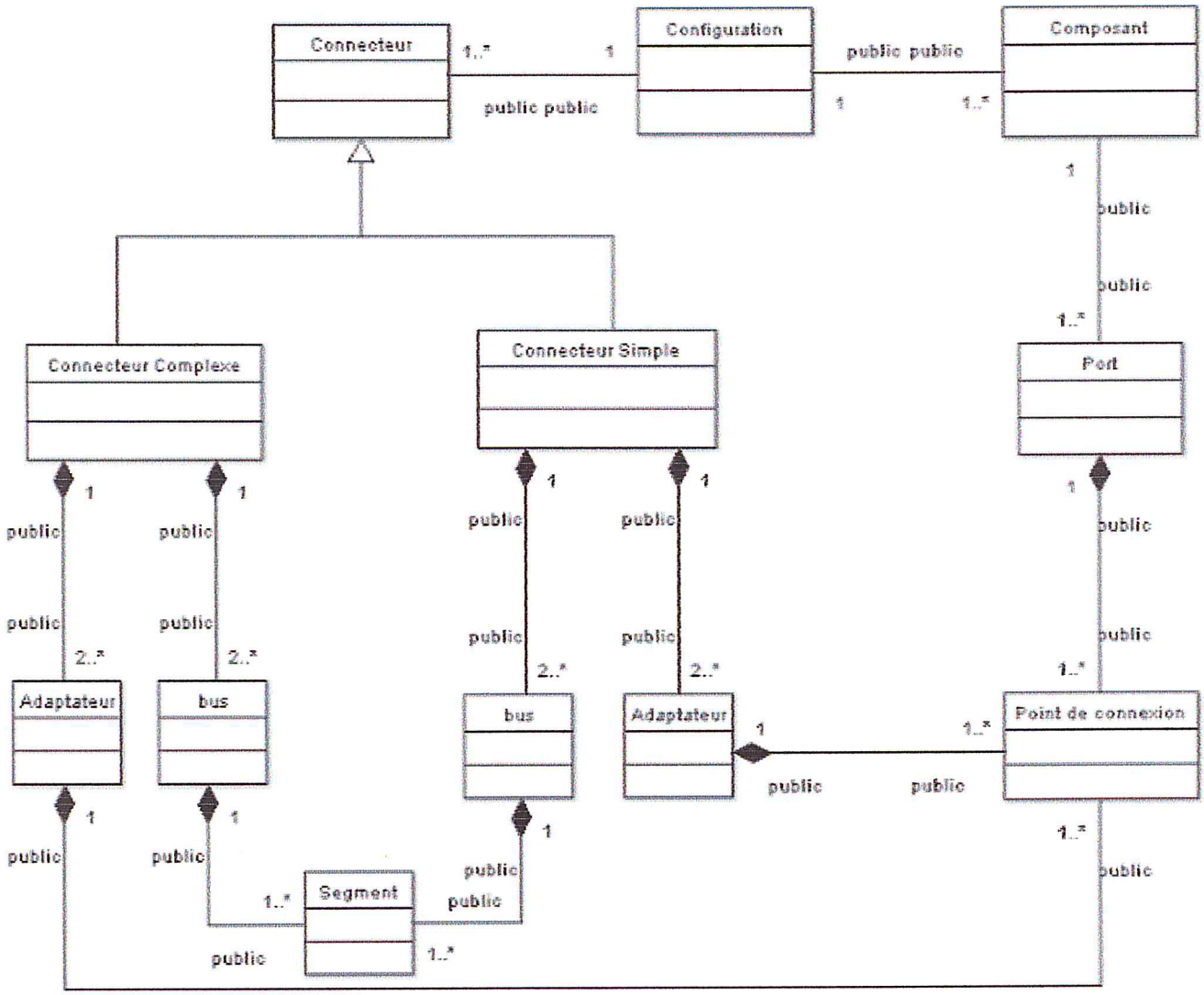


Figure VIII.10 : Diagramme de classes du concept configuration.

Description brève :

- **Configuration :** Classe représente le concept Configuration.
- **Connecteur :** Elle représente le concept Connecteur.
- **Connecteur simple :** Elle représente le concept Connecteur simple.
- **Connecteur complexe :** Classe représente le concept Connecteur complexe.
- **Adaptateur :** Elle représente le concept Adaptateur.
- **Bus :** Classe représente le concept Bus.
- **Segment :** Elle représente le concept Segment.
- **Point de connexion :** Classe représente le concept Point de connexion.
- **Composant :** Classe représente le concept Composant.
- **Port :** Elle représente le concept Port.

Bibliographie

[1] The University of California: ArgoUML V0.16.1 est un environnement de modélisation UML.
Site : http://argouml.tigris.org
[2] IBM : eclipse 3.0.1 est un environnement de développement a base de plugins (Annexe).
Site : http://www.eclipse.org/platform

Chapitre IX

La réalisation

On va entamer la réalisation de notre application avec le processus suivant : Une présentation générale de notre application, puis une description de l'implémentation, une présentation de fonctionnalités et enfin les outils et documentation utilisés.

I. INTRODUCTION

Globalement, l'application de conception par assemblage des systèmes informatiques complexes MCC-CASIC a été réalisée dans le but précis d'analyser et de créer des configurations permettent de visualiser les architectures logicielles et mise en place de la connectivité entre ses composants logiciels.
Une fonctionnalité intéressante de l'application est la capacité à schématiser des configurations des systèmes de plus simple au plus complexe, et joue le rôle d'un outil d'aide au architecte logiciel.
Nous avons tenté de respecter au mieux les délais et contraintes imposées par le promoteur, mais après la rencontre de quelques problèmes de développement et l'ajout de quelques contributions à notre initiative, le planning prévu a dû évoluer.
Ainsi, nous allons vous présenter le travail que nous avons réalisé, en détaillant notre organisation, en précisant les choix que nous avons effectués et en fournissant le résultat voulu.
L'outil se présente sous forme d'interface graphique. En créant une nouvelle configuration, qu'elle est construite avec le regroupement d'un ensemble de composants, en utilisant un ensemble de connecteurs avec ses diverses topologies, pour l'établissement de la connectivité, afin d'aboutir à une architecture logicielle bien précis, qui réalise les besoins d'un architecte logiciel.

II. PRESENTATION GENERALE DE NOTRE APPLICATION

Dans cette partie, nous allons présenter L'application MCC-CASIC, dans ces grandes lignes, et tel qu'il devrait être a la fin de son développement.
L'application MCC-CASIC est un outil aide aux utilisateurs (architectes logiciel) pour la construction totale des systèmes par assemblage de composants logiciels, sa structure apparue comme deux vues complémentaires :

- un éditeur graphique permettant de définir des configurations de logiciels, et tous ces éléments internes (composants, connecteurs simples, connecteurs complexes, ports, adaptateurs, point de connexion), comme expliqué durant la présentation du modèle de connecteur.
- Et une vue interne qui permet la génération de code associé à la connectivité réalisée entre les composants de la configuration.

III. DESCRIPTION DE L'IMPLEMENTATION

III.1. Description de travail

Nous avons distinguer deux types de travail : ce qui suivent la méthodologie prédéfinie et nous ont amené à rédiger nos différentes classes ; ceux qui résultent des consignes générales au niveau de développement et qui s'appliquent à l'ensemble de nos classes.
Le premier travail a comme méthodologie la présentation des interfaces graphiques, ou bien L'IHM (Interface Homme Machine), en utilisant un API (Application Programming Interface), qui est le Swing de Java.
Cependant Le deuxième travail a comme méthodologie l'utilisation des API aide au génération de code interne de la configuration, parmi ces API : les flux d'E/S (entrés/sorties), les fichiers, les threads, la gestion réseaux et le package CORBA.

III.2. L'éditeur graphique

III.2.1. Présentation générale

La capture d'écran suivante représente une vue globale de notre application.

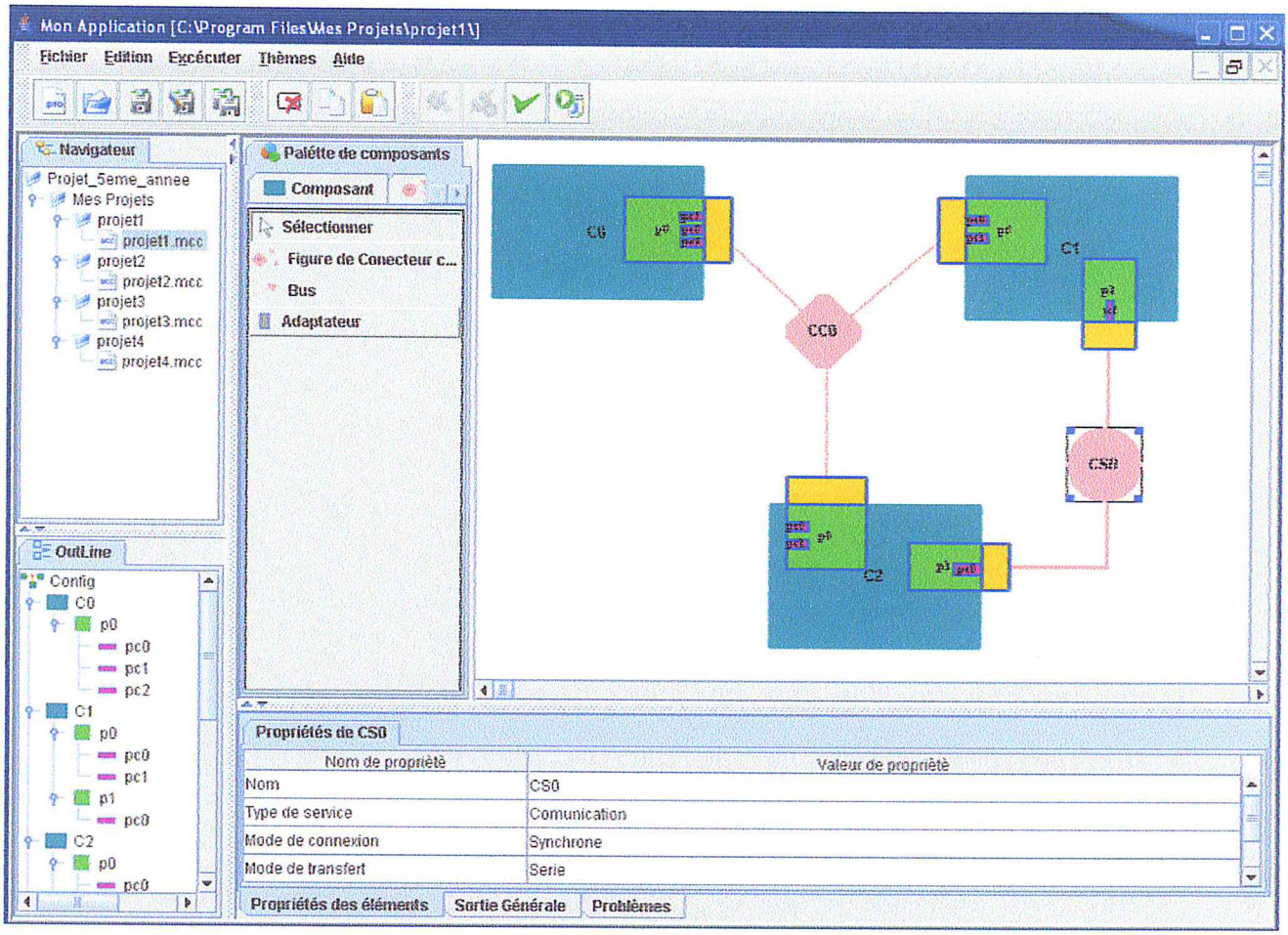


Figure IX.1 : Vue globale de l'application.

III.2.2. Description détaillée des classes

Notre travail a consisté, dans son intégralité, à développer en Java [1] l'application MCC-CASIC tout en respectant les consignes définies auparavant (chapitre conception). C'est en suivant la méthodologie présentée dans le paragraphe 3.2 du chapitre précédant, que nous sommes parvenus à rédiger 48 classes dans les principales sont :

- Prog_Principal.
- FenêtrePrincipale.
- Fenêtre_de_configuration.
- Composant.
- ConnectorSimple.
- ConnectorComplexe.
- Port.
- Adapter.
- ListProjets.
- ListBus.
- Listcomposant.
- ListSegment.

- DiagrammeEditor.
- DrawEditor.
- Palette_de_comp.
- Diagramme_Connector_Complexe.
- Diagramme2_Connector_Complexe.
- Bus.
- Segment.
- Schéma_Connector_Complexe.
- Schéma1_ConnC.
- Point_De_Connexion.
- Projet.
- SourceEditor.
- Navigateur.
- Outline.
- Tableau.
- TablePropriété.
- EtatSortie.
- SourceEditor.
- InfoComposant.

Pour décrire le travail, nous commencerons par détailler les classes qui correspondent à la mise en place de l'interface homme-machine.

Il convient de préciser que nous avons utilisé la plupart du temps des classes en provenance du package « javax.swing » de «Sun Microsystems» [2], car ces classes nous offrent un plus grand nombre de fonctionnalités sur les objets graphiques.

- La classe Prog_Principal :

  La classe Prog_Principal est la classe principale de l'application MCC-CASIC. C'est elle qui comporte la méthode main qui permet le lancement de l'application, avec l'instanciation de la classe FenêtrePrincipale, qui est décrite ci-après.

- La classe FenêtrePrincipale :

  Elle hérite de la classe « JFrame », et c'est dans celle-ci que sont créés les panneaux nous permettant de définir l'aspect général de l'application.

- La classe Composant :

  Une classe permet la représentation graphique de Composant. Elle hérite de la classe «JLabel ».

- La classe ConnectorSimple :

  Une classe permet la représentation graphique d'un Connecteur Simple. Elle hérite de la classe « Composant ».

- La classe ConnectorComplexe :

  Une classe permet la représentation graphique d'un Connecteur Complexe. Elle hérite de la classe « Composant ».

- La classe Port :

  Une classe permet la représentation graphique d'un Port. Elle hérite de la classe « Composant ».

- La classe Adapter :
  Une classe permet la représentation graphique de l'adaptateur. Elle hérite de la classe «Composant ».

- La classe Bus :
  Une classe permet la représentation graphique de Bus. Où le Bus est constitué d'un ou plusieurs segments.

- La classe Segment :
  Une classe permet la représentation graphique d'un Segment, qui représente une partie de Bus d'un connecteur.

- La classe listcomposant :
  C'est une Structure de données (liste linéaire chaînée), utilisée pour stocker les composants.
  Avec le sous casting sur cette classe on peut hériter :
  - Liste de connecteurs simples.
  - Liste de connecteurs complexes.
  - Liste d'adaptateurs.
  - Liste de points de connexion.
  - Liste de Ports.

- La classe listBus :
  Structure de données (liste linéaire chaînée), utilisée pour stocker les bus des connecteurs en cours de l'utilisation du l'application

- La classe ListSegment :
  Structure de données (liste linéaire chaînée), utilisée pour stocker les bus des connecteurs en cours de l'utilisation du l'application

- La classe ListProjets :
  Structure de données (liste linéaire chaînée), utilisée pour stocker les projets de l'application.

- La classe DiagrameEditor :
  Elle permet la représentation de la configuration. Elle hérite de la classe «JSplitPane », et rassemble la palette de composants et l'éditeur de dessin DrawEditor.

- La classe Diagramme_Connector_Complexe :
  La fenêtre qui contient l'instance du Schéma_Connector_Complexe. Elle hérite de la classe « JFrame ».

- La classe Diagrame2_Connnector_Complexe :
  La fenêtre qui contient l'instance de Schéma 1_ConnC de sous-connecteur complexe. Elle hérite de la classe « JFrame ».

- La classe DrawEditor :
  Elle hérite de la classe « JPanel », et supporte un JDesktop qui à son tour supporte des fenêtres internes qui contiennent les configurations. Elle contient le dessin de l'ensemble de composants afin de construire le schéma de configuration.

- La classe Palette_de_comp :
  Elle hérite de la classe « JTabbedPane », et permet de visualiser la zone d'outils de dessin. Avec la technique glisser-déplacer on peut dessiner les éléments de configuration.

- La classe Fenêtre_de_configuration :
  C'est la fenêtre de zone de dessin où on place les différents éléments de configuration. Elle hérite de la classe « JinternalFrame ».

- La classe Schéma_Connector_Complexe :
  Permet de définir l'architecture du connecteur complexe d'une configuration. Elle hérite de la classe «DiagrameEditor ».

- La classe Schéma 1_ConnC :
  Permet de définir l'architecture de sous-connecteur complexe, qui appartient au schéma d'un connecteur complexe d'une configuration. Elle hérite de la classe «Schéma_Connector_Complexe ».

- La classe Point_De_Connexion :
  Une classe permet la représentation graphique d'un Point de Connexion Elle hérite de la classe « Port ».

- La classe Projet :

Classe qui contient les vues d'un projet.

- La classe SourceEditor :
  Une classe qui crée un éditeur de texte. Elle hérite de la classe « JPanel ».

- La classe Navigateur :
  Une classe qui permet de crée l'arbre qui visualise l'ensemble de projets. Elle hérite de la classe « JPanel ».
  Il faut noter que la création de ces nœuds est réalisée par la classe «DefaultMutableTreeNode».

- La classe Outline :
  Cette classe contient l'arbre qui visualise tous les éléments d'une configuration, alors elle permet la sélection des éléments de la fenêtre de configuration lorsque un nœud est sélectionné. Elle hérite de la classe « JPanel ».
  Il faut noter que la création de ces nœuds est réalisée par la classe «DefaultMutableTreeNode»

- La classe Tableau:
  Elle hérite de la classe « JTable ». Elle permet de crée un tableau supportant toutes les propriétés fonctionnelles du modèle, ou un tableau contient les problèmes rencontrés.

- La classe SourceEditor :
  Une classe qui crée un éditeur de texte. Elle hérite de la classe « JPanel ».

- La classe EtatSorie:
  Elle hérite de la classe « JPanel », leur rôle est de visualiser la sortie générale, Propriétés d'un élément, et le tableau des problèmes reçus.

- La classe InfoComposant :
  Elle est chargée de la sauvegarde des projets de l'application.

III.2.3. Schéma global de l'application

On peut schématiser les choix d'implémentation généraux de l'application de la façon suivante :

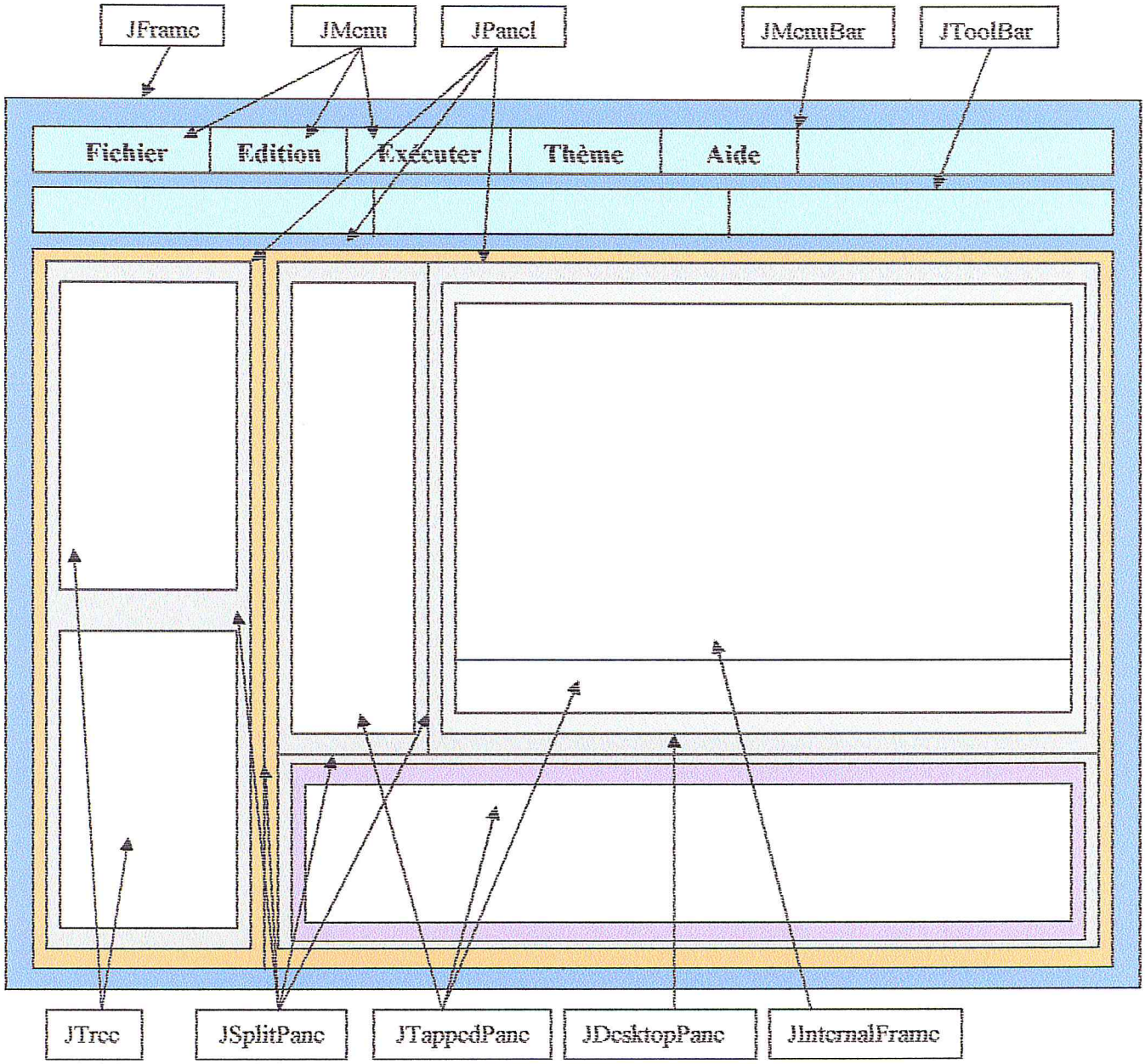


Figure IX.2 : Structure de l'application MCC-CASIC.

La fenêtre principale du l'application MCC-CASIC est une JFrame. Celle-ci joue le rôle de « Container » et accueille donc d'autres composants JavaSwing.
Elle est ainsi composée de trois entités : un JMenuBar, un JToolBar et un JPanel.

- Le composant **JMenuBar** est lui-même composé de cinq **JMenu** : « Fichier », « Edition », « Exécuter », « Thème » et «Aide». Ces cinq JMenu étant composés eux-mêmes de plusieurs sous menus de type **JMenuItem** :

« Fichier » ⇨ «Nouveau projet», «Ouvrir un projet», «Enregistrer le projet», «Enregistrer le projet sous», « Enregistrer tout», «Fermer le projet» et «Quitter».
« Edition » ⇨ «Supprimer», «Copier», «Coller» et «Sélectionner tout».

« Exécuter » ⇨ «Valider» et «Générer».
«Thème » ⇨ «Windows», «Motif» et «Java».
«Aide » ⇨ «Rubrique d'aide» et «A propos».

- Le composant **JToolBar1** est composé de cinq **JToogleButton** associés aux Sous-menu de Menu « Fichier » (« Nouveau projet », «Ouvrir un projet», «Enregistrer le projet», « Enregistrer le projet sous » et « Enregistrer tout »).

- Le composant **JToolBar2** est composé de trois **JToogleButton** associés aux Sous-menu de Menu « Edition » (« Supprimer », « Copier », « Coller »).

- Le composant **JToolBar3** est composé de deux **JToogleButton** associés aux Sous-menu de Menu « Exécuter » («Valider », « Générer »).

- Le composant **JPanel** est composé d'un **JSplitPane** à son tour composé de deux **JPanel.**
  - Le composant **JPanel** gauche est composé d'un **JSplitPane** à son tour composé de deux **JPanel** contenais des JTree.
  - Le composant **JPanel** droit est composé d'un **JSplitPane** à son tour composé de deux **JPanel.** Le **JPanel** de Haut support un **JSplitPane.**
    - Le composant **JSplitPane** haut est composé d'un **JTappedPane** dans sa partie gauche, et d'un **JDesktopPane** dans sa partie droite.
      - Le composant **JDesktopPane** est composé d'un **JInternalFrame.**

    - Le composant **JPanel** bas est composé d'un **JTappedPane** qui regroupe deux **JTable** et un **JTextPane.**

III.2.4. Technique de sauvegarde

Dans un chemin de sauvegarde identifié par l'utilisateur de l'application, et après un clic sur le sous-menu « Nouveau projet », un répertoire nommé *MesProjets* sera crée. Le contenue de ce répertoire est l'ensemble des répertoires associés à chaque projet, où chacun d'eux supporte le fichier représentatif d'un projet, qui à comme extension « .mcc », et comme icône «  ».

Le sauvegarde des configurations nécessite une opération de choix des critères à sauvegardés (nom de l'élément, leur position…etc.).
La classe chargée de cette opération « **InfoComposant** », représente un Technique bien particulier de sauvegarde qui est le suivant :

1. Avoir des informations (critères de sauvegarde) sur chaque élément de la configuration puis les enregistrer au niveau de disque dur.
2. la restauration des ces informations et reprendre l'affichage de ces éléments de nouveau.

III.3. Validation et génération du code de connectivité

Il représente la validation des configurations et les schémas des connecteurs et sous-connecteurs associés, puis la génération du code de connectivité (connecteurs) associés.

La validation :

S'effectue en deux étapes :

Test de validation d'un connecteur

Pour tester si un connecteur est validé, en va suivre le processus qui est rédigé sous forme une fonction.

La fonction de validité d'un connecteur :

```
Algorithme valider_connecteur (Connecteur) : Booléen
Si le connecteur est élémentaire ALORS
    SI le connecteur est connecté  ALORS
        SI  connecteur. Mode de transfert = unidirectionnel ALORS
            Si  il existe un seul point «Sortie» et les autre sont des «entrée»ALORS
                SI le point «Sortie» et les «entrée» sont de même type ou compatibles ALORS
                    Connecteur valide
                FIN-SI
                SINON
                    Connecteur non valide
                FIN-SINON
            FINSI
            SINON
                Connecteur non valide
            FIN-SINON
        FINSI
        SINON
            SI les points sont tous « entrée-sortie » ALORS
                SI les points sont de sont de même type ou compatibles ALORS
                    Connecteur valide
                FINSI
                SINON
                    Connecteur non valide
                FIN-SINON

            FINSI
            SINON
                Connecteur non valide
            FIN-SINON
        FIN-SINON

    FINSI
FINSI
```

Test de validation d'une configuration

Pour tester si une configuration est validée, en va suivre le processus qui est rédigé sous forme une procédure d'une seule entrée et deux sorties

La procédure de validité d'une configuration:

```
Procédure valider_configuration (liste_connecteur :Liste, Var  valider_config :Booléen,
                                 liste_connecteur_valide :Liste) ;
valider_config ← vrai
TANT QUE  liste_connecteur <> null     FAIRE
   SI valider_connecteur (liste_connecteur. get(connecteur) ) = vrai ALORS
         liste_connecteur_valide. Inserer (connecteur)
   FIN-SI
   SINON
         valider_config ← faux
   FINSINON
FIN-TANT QUE
```

La génération du code de connectivité (pour chaque connecteur) :
Il y a trois cas de figure
Niveau1
Le déploiement indique que les composants (les ports) évoluent dans un espace de nomination commun (même processus, même thread, même classe pour des objets de classes internes, même module etc...).
Les adaptateurs ne dispose aucune capacité de mémorisation au niveau de leurs points de connexion, alors on établisse une connexion directe.
Cas de figure :

- Connecteur de connexion directe

Soit l'exemple suivant : simple interaction entre deux composants

```
// la class de premier composant ----------------------------------------------------------
public class composant1{
        int entrée1;
        char sortie1;
        public composant1(){   // le constructeur

        }
        public int getsortie(){
    return entrée1;
    }
    public Void setentrée (char x){
        sortie1=x;
    }
}

// la class de deuxième  composant ----------------------------------------------------------
public class composant2{
        int entrée2;
        char sortie2;
        public composant2(){   // le constructeur

        }
        public int getsortie(){
    return entrée2;
    }
```

```
        public Void setentrée(char x){
            sortie2=x;
        }
    }
    // gestion de transfert de données au niveau configuration ----------
    import fichier1 // le fichier1 contient la classe composant1
    import fichier2 // le fichier2 contient la classe composant2

  public class configuration{

    composant1 c1 = new composant1();
    composant2 c2 = new composant2();

    public class connector extends Thread{
        void run(){
            c2.setentrée(c1.getsortie());
        }
    }
    public void main(){
        new connector().start();
    }
}
```

Niveau2

Les composants évoluent dans des espaces de nomination différents.
Les connecteurs doivent avoir une capacité de mémorisation au niveau de leurs adaptateurs plus précisément au niveau de points de connexion.
Cas de figure :

- Connecteur support des variables internes permet une opération simple de copie dans ces variables.

Niveau3

Les composants sont distants. Le déploiement indique que les composants évoluent dans des environnements totalement indépendants.
Il est composé de divers services fournis par le système d'exploitation pour la communication interprocessus (exemple : techniques de communication par mémoire partagé, les pipe, les sockets).
Les valeurs sont transformées selon les besoins de la technologie utilisée,
Cas de figure :

- Il peut être représenté par une infrastructure complète de communication (Exemple : CORBA, JAVA RMI)
- Ou bien par un système de communication propriétaire basée sur les sockets ou autre technique offerte par les systèmes d'exploitation.

IV. FONCTIONNALITES

Cette partie permet de prendre connaissance avec l'application MCC-CASIC, en découvrant ses possibilités par le biais de nombreuses captures d'écran. Ce "manuel", nous l'espérons, permettra au lecteur de se familiariser rapidement avec l'application et d'apprécier son intuitivité et son intérêt.

IV.1. Fonctionnement de base

L'application est apparue en quatre vues coopèrent entre eux pour réaliser le *fonctionnement attendu.*

IV.1.1. Initialisation (Lancement de l'application)

L'outil est initialisé à son ouverture avec plusieurs panneaux:
Une partie gauche contient deux arbres, la première pour l'exploration de la zone de travail (les différents projets avec ses contenus), et la deuxième est dédiée à la visualisation de différents éléments de la configuration réalisée au niveau éditeur de schéma de configuration.
Et une partie droite contient en haut un éditeur de schéma de configuration, et en bas un panneaux dédié à la présentation des différentes propriétés associés à chaque élément, et une zone de sortie qui affiche les résultats et les problèmes rencontrés après la validation de l'architecture réalisée. (Voir ci-dessus «Figure IX.1»).

IV.1.2. Barre de Menus

C'est une barre qui regroupe cinq menus.

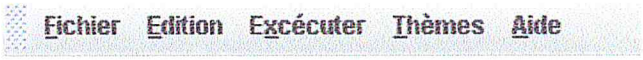


Figure IX.3 : La barre de menus

- Menu « Fichier »
  - Sous-menu « Nouveau projet »

    Permet la création d'un nouveau projet.

Cette fonctionnalité est réalisée au moyen d'une fenêtre de type « JFrame » qui permet à l'utilisateur de saisir le nom d'un projet puis créer fichier associé si le chemin est par défaut, sinon il aura la possibilité de choisir un autre chemin à partir d'une fenêtre de type « JFileChooser » (Figure IX.4).
Enfin, une fenêtre de boite de message apparue pour valider le choix de nom de projet donné (Figure IX.6).

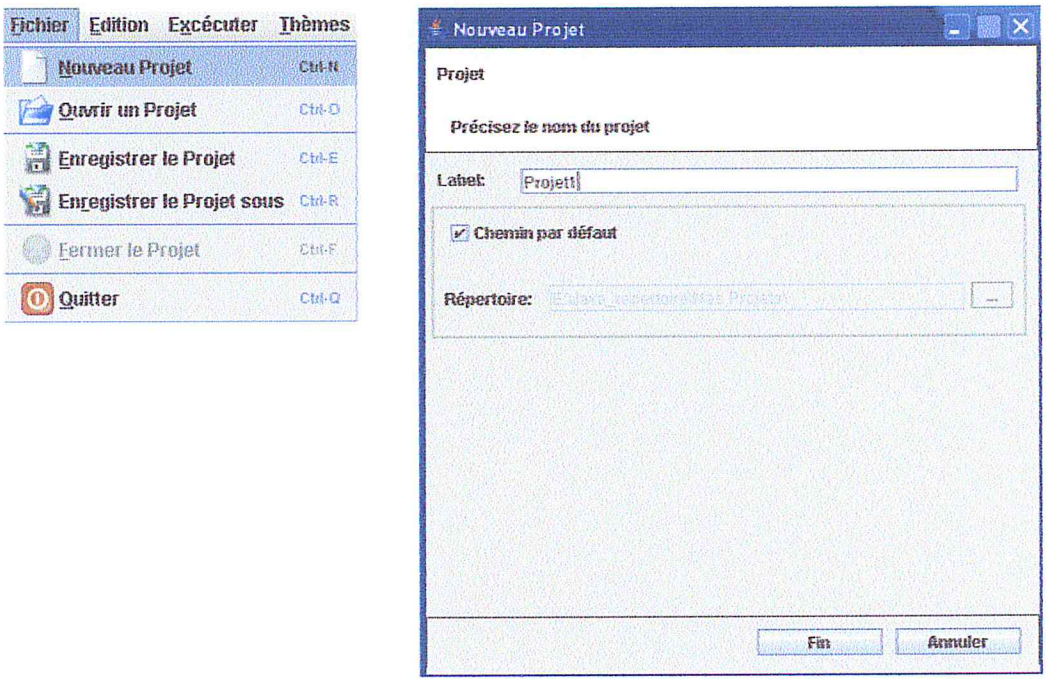


Figure IX.4 : Gestion de sous-menu « Nouveau projet ».

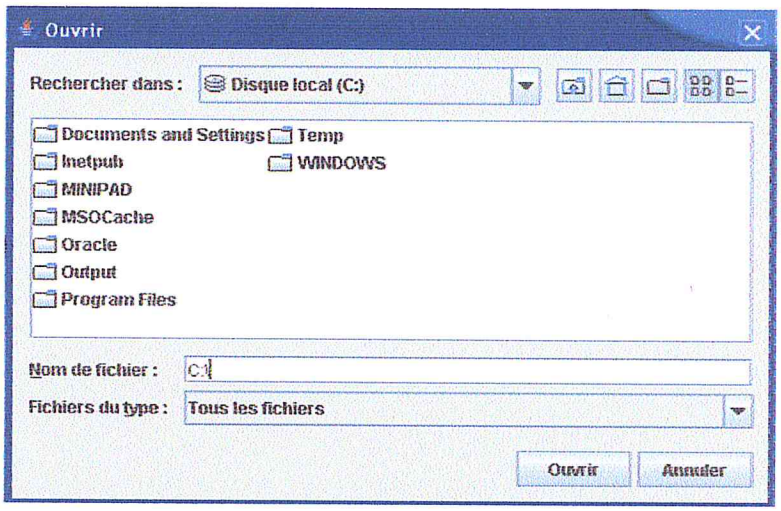


Figure IX.5 : Choix de chemin de sauvegarde

Figure IX.6 : Validation de choix de nom de projet donné

- Sous-menu « Ouvrir un projet »

Permet l'ouverture d'un projet existe dans la zone de travail (WorkSpace), et apparaître le fichier du projet qui a comme extension '.mcc'.
Cette fonctionnalité est réalisée au moyen d'une fenêtre de type « JFileChooser » qui permet à l'utilisateur de naviguer dans l'arborescence et de choisir le fichier avec l'extension voulue (Figure IX.7).

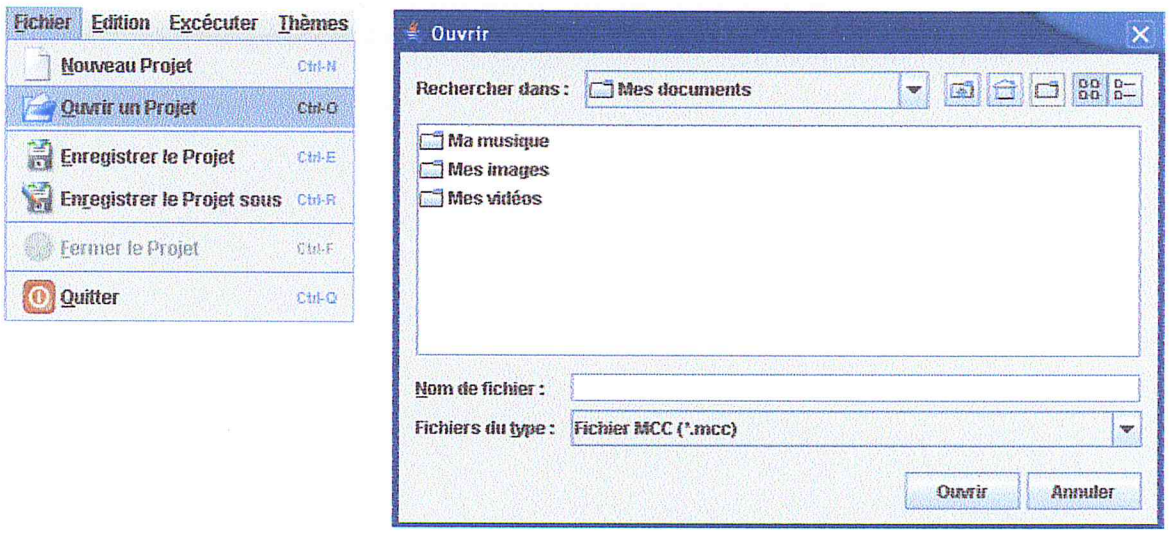


Figure IX.7 : Gestion de Sous-menu « Ouvrir un projet »

- Sous-menu « Enregistrer Le projet » Permet la sauvegarde d'un fichier de projet au fur et à mesure des modifications effectuées sur le projet en cours de la réalisation. Et pour la première fois de sauvegarde, la fenêtre enregistrer apparaître.

Figure IX.8 : Gestion de Sous-menu « Enregistrer le projet »

- Sous-menu « Enregistrer le projet sous »
  Il permet de sauvegarder un projet, ou le renommer.

Cette fonctionnalité est réalisée au moyen d'une fenêtre de type « JFileChooser » qui permet à l'utilisateur de naviguer dans l'arborescence et de nommer le fichier qu'il souhaite sauvegarder (Figure IX.9).

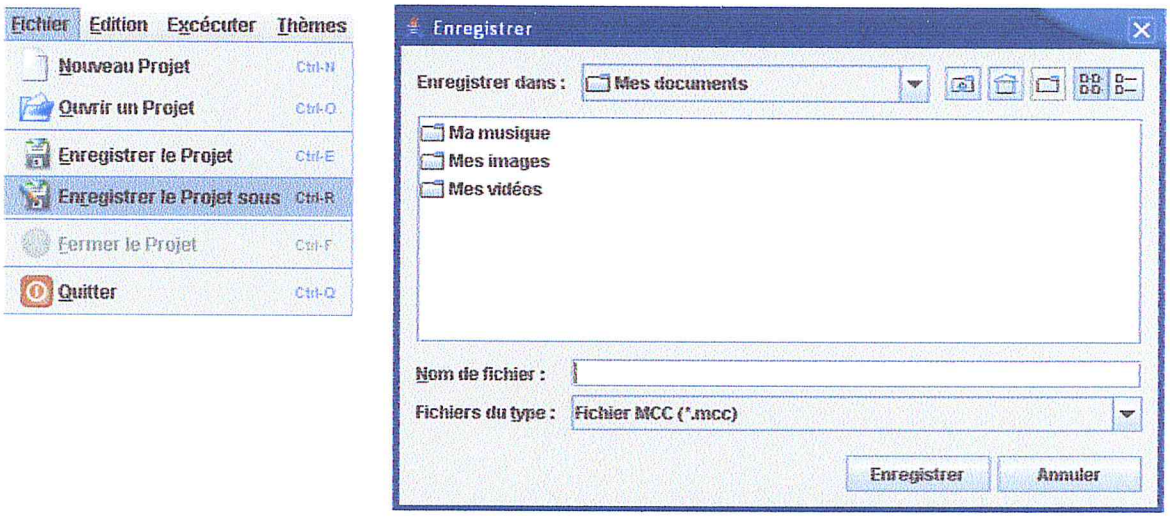


Figure IX.9 : Gestion de Sous-menu « Enregistrer le projet sous ».

- Sous-menu «Enregistrer tout »Il permet de sauvegarder touts les projets ouverts (Figure IX.10).

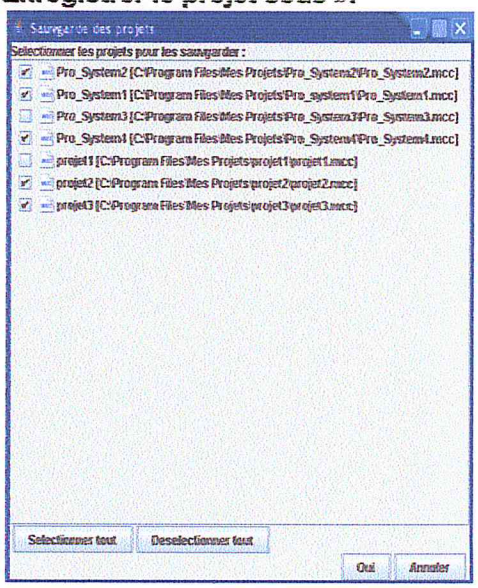


Figure IX.10 : Gestion de sous-menu «Enregistrer tout».

- Sous-menu « Fermer le projet »
  Il permet la fermeture d'un projet ouvert, et passe la main à un autre projet existe dans l'arbre Navigateur (selon l'ordre de lancement).

Figure IX.11 : Gestion de sous-menu « Fermer le projet ».

- Sous-menu « Quitter »
  Permet la fermeture de l'application où une fenêtre apparaître sous le titre sauvegarde des projets (Figure IX.), qui permettre la sélection-désélection des projets de l'arbre de vue *Navigateur* pour les enregistrer (Figure IX.12).

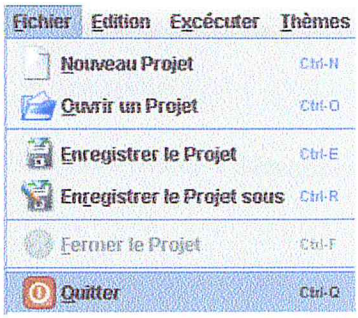


Figure IX.12 : Gestion de sous-menu «Quitter».

- Menu « Edition »
- Sous-menu « Supprimer »
  Permet de supprimer un élément, une partie, ou la configuration elle-même.

Figure IX.13 : Gestion de sous-menu « Supprimer »

- Sous-menu « Copier »
  Il permet de lancer l'opération de copier d'un élément après la sélection de celui-ci.

Figure IX.14 : Gestion de sous-menu « Copier »

- *Barre d'outils 1*

Il est composé de cinq boutons reflètent les sous-menus de menu « Fichier » (Figure IX.23).

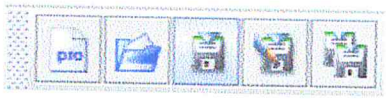


Figure IX.23 : Barre d'outils 1

- *Barre d'outils 2*

Il est composé de trois boutons reflètent les sous-menus de menu « Edition » (Figure IX.24).

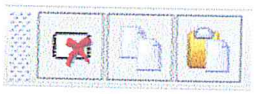


Figure IX.24 : Barre d'outils 2

- *Barre d'outils 3*

Il est composé de quatre boutons reflètent les sous-menus de menu « Exécuter » (Figure IX.25).

Figure IX.25 : Barre d'outils 3

Les boutons de ces barres permettent l'accès au contenue de l'application.

IV.1.4. La partie gauche

- *Haut*

Cette partie de l'interface affiche exactement les fichiers des projets dans un arbre. A chaque création d'un nouveau projet il sera affecter dans cet arbre avec un icône représentatif. Comme nous avons la main de lancer l'un de ces projets le moment où on veut (Figure IX.26).

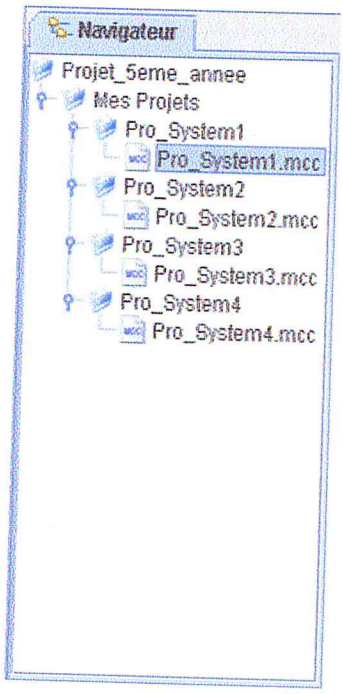


Figure IX.26 : Visualisation des projets dans un arbre.

- Sous-menu « Coller »
  Permet de coller l'élément (partie ou une configuration) copier dans le même projet ou dans un autre.

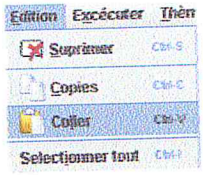


Figure IX.15 : Gestion de sous-menu « Coller »

- Sous-menu « Sélectionner tout »
  Il permet la sélection totale de touts les éléments de la configuration.

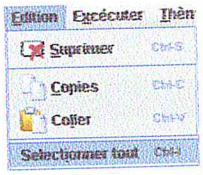


Figure IX.16 : Gestion de sous-menu « Sélectionner tout »

- Menu « Exécuter »
  - Sous-menu «Valider »
    Permet la validation des configurations des schémas des connecteurs complexes.
  - Sous-menu « Générer »
    Il lance l'opération de génération de code de connectivité (code de connecteur).

- Menu « Thème »
  - Sous-menu « Windows »
    Il donne le thème (LookAndFeel) Windows à l'application (Figure IX.17).

Figure IX.17 : Le Sous-menu « Windows ».

  - Sous-menu « Motif »
    Il donne le thème (LookAndFeel) Motif à l'application (Figure IX.18).

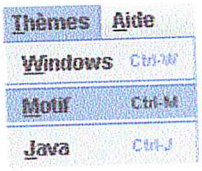


Figure IX.18 : Le Sous-menu « Motif ».

  - Sous-menu « Java »
    Il donne le thème (LookAndFeel) Java à l'application (Figure IX.19).

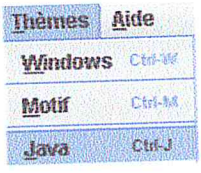


Figure IX.19 : Le Sous-menu « Java ».

- Un connecteur complexe aura une architecture à base des sous-connecteurs et sous-composant (l'explication aura lieu aux prochains points).

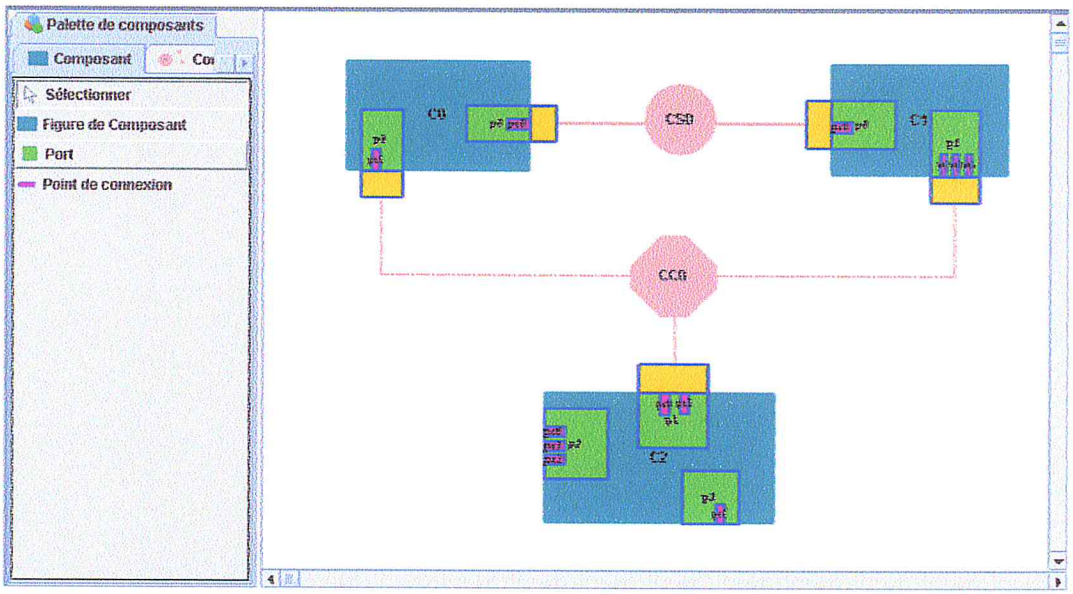


Figure IX.28 : Editeur de configurations regroupe la palette de composants et la zone de dessin

❖ **Comment placer et configurer les éléments de la palette dans la zone de dessin**

1. la sélection d'un élément

Avec la gestion presser relâcher de souris on crée le rectangle de sélection qu'il doit entourer cet élément (Figure IX.29).

(1) Composant Avant sélection (2) Composant sélectionné

Figure IX.29 : La sélection d'un élément (ex : composant).

2. La connexion graphique des adaptateurs aux ports

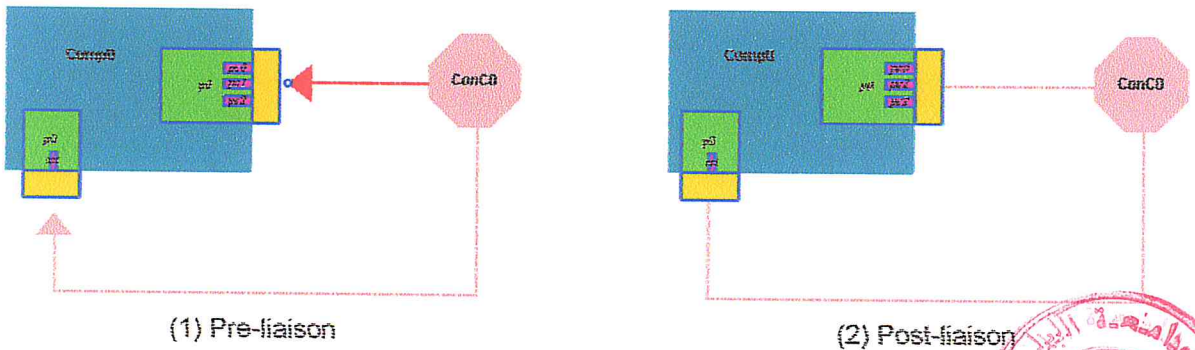
Il passe par deux étapes :

2.1. L'affectation des adaptateurs aux ports

(1) Pre-affectation (2) Menu contextuelle d'affectation (3) Post-affectation

Figure IX.30 : Affectation des adaptateurs aux ports.

2.2. La liaison des adaptateurs

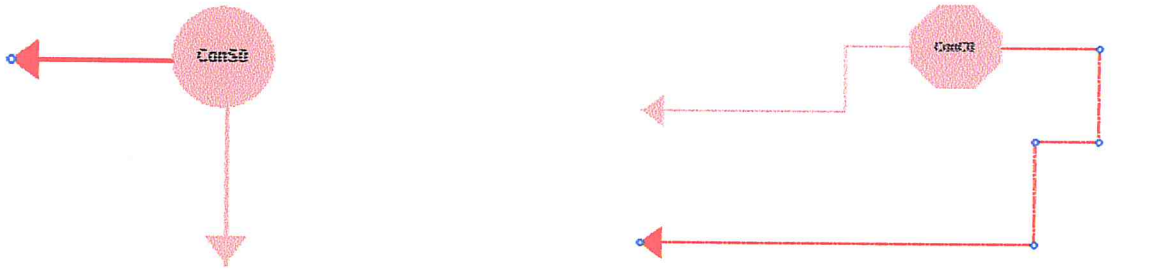


(1) Pre-liaison (2) Post-liaison

Figure IX.31 : Liaison des adaptateurs.

3. Les bus d'un connecteur

Un bus prend dans sa sélection la couleur rouge et une petite cercle bleu dans sa tête. Il peut être cassé en plusieurs tranches appelées segments en double cliquant sur un point de celui-ci.

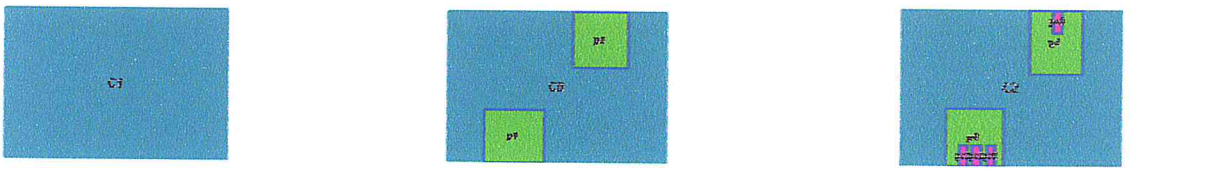


(1) Bus sélectionné d'un connecteur simple. (2) des bus cassés d'un connecteur complexe.

Figure IX.32 : Les bus d'un connecteur

4. L'ajout des ports et points de connexion

Avec la gestion glisser déplacer on peut ajouter des ports au composant, comme on peut ajouter des points de connexion au port.



(1) composant (2) l'ajout des ports (3) l'ajout des points de connexion

Figure IX.33 : L'ajout des ports et des points de connexion.

5. La gestion Copier Coller et supprimer

La suppression de touts les éléments même une configuration ou une partie de celle-ci se fait à travers le Menu et la barre d'outils.

La gestion Copier Coller permet de copier une configuration ou une partie puis la coller.

- Menu « Aide »
  - Sous-menu « Rubrique d'aide »
    Lance une fenêtre contient l'aide sur l'utilisation et la manipulation de l'application (Figure IX.21).

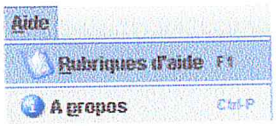


Figure IX.20 : Le Sous-menu « Rubrique d'aide ».

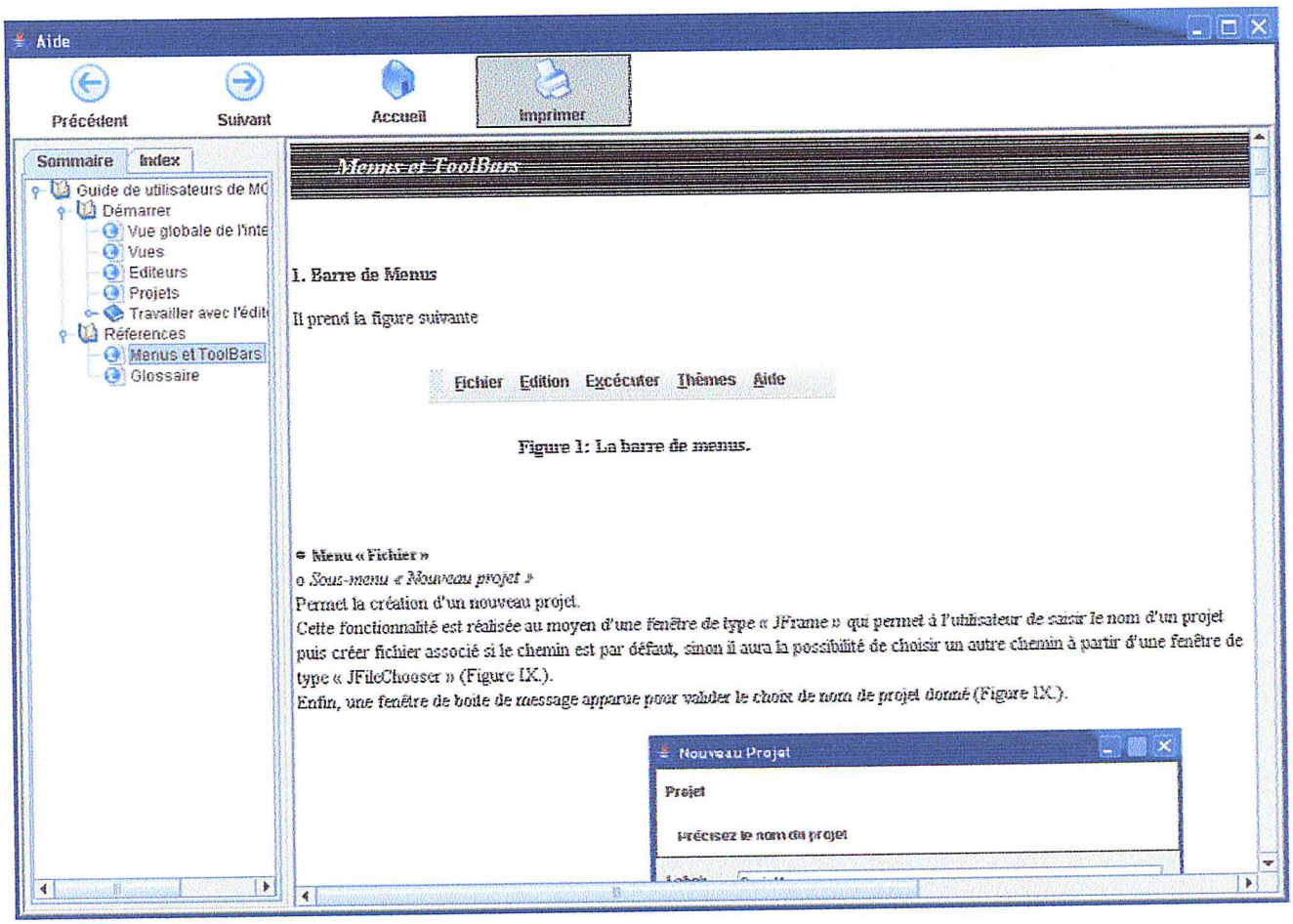


Figure IX.21 : Fenêtre de la Rubrique d'aide.

  - Sous-menu « A propos »
    Donne des informations sur la version du logiciel et les auteurs (Figure IX.22).

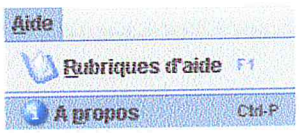


Figure IX.22 : Le Sous-menu « A propos ».

IV.1.3. Barres d'outils

L'application MCC-CASIC possède trois barres d'outils, qui sont organisées comme :

Remarque : autre suppressions sont possibles depuis l'arbre hiérarchique de la vue Outline, en utilisant le clavier.

- *Bas* (Vue EtatSortie (Figure IX.34))

Cette partie de l'application affiche la vue EtatSortie qui regroupe les panneaux suivants :

- Propriétés des éléments :

Ce panneau représente les propriétés de chaque élément sélectionné, il contient un tableau de deux colonnes et plusieurs lignes. La première cellule affiche le nom de la propriété et la deuxième sa valeur.

Et voici les Propriétés exploitées des chaque élément

- Connecteur élémentaire :
  - Nom.
  - Type de service.
  - Mode de connexion.
  - Mode de transfert.
  - Sens de l'interaction.
  - Disposition des composants.
- Composant :
  - Nom.
  - Chemin.
- Port :
  - Nom.
  - Type de port.
- Point de connexion :
  - Nom.
  - Nature de point de connexion.
  - Type de point de connexion.

- Sortie Générale :

Représente les messages de validation s'il n'y a aucun problème, sinon des messages d'erreurs apparaissent.

- Problèmes :

Il supporte la structure d'un tableau où chaque ligne affiche un problème de validation rencontré avec précision des éléments concernés.

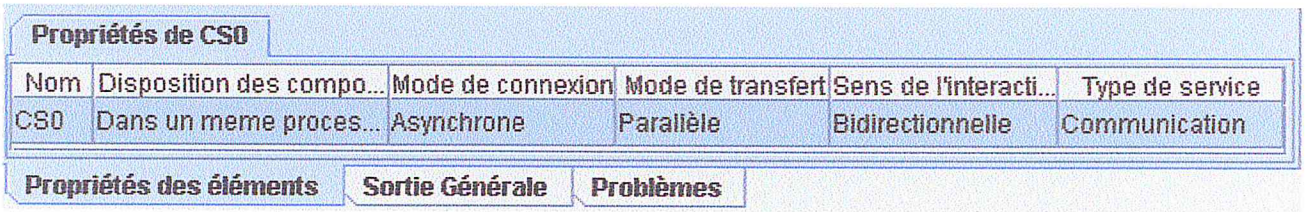


Figure IX.34 : La vue EtatSortie regroupe les propriétés, la sortie générale et les problèmes.

IV.1.6. Fenêtre de schéma détaillé d'un connecteur complexe

C'est une zone dédiée à la schématisation d'un connecteur complexe. Elle est partagée en deux parties (Figure IX.35):

- Editeur de diagramme de connecteur complexe
  - La palette de composants

Elle est de type JTappedPane à deux panneaux (connecteur simple, connecteur complexe).
Chacun de ces boutons représente un élément graphique du schéma de connecteur complexe en cours de conception (connecteur simple, connecteur complexe, bus, adaptateur).

- La zone de dessin

Avec la gestion glisser déplacer, on peut dessiner touts ces éléments sous les règles et les contraintes d'exploitation suivantes :

- Un port est présenté avec un ou plusieurs points de connexion.
- Un adaptateur d'un connecteur simple (complexe) est connecté à un point de connexion.
- Un connecteur simple (même complexe) a au moins deux bus.
- Un connecteur complexe aura une architecture à base des sous-connecteurs et sous-composants.

➢ Vue EtatSortie (Propriétés des éléments, Sortie Générale, Problèmes)
Même description que la vue EtatSortie de la fenêtre principale.

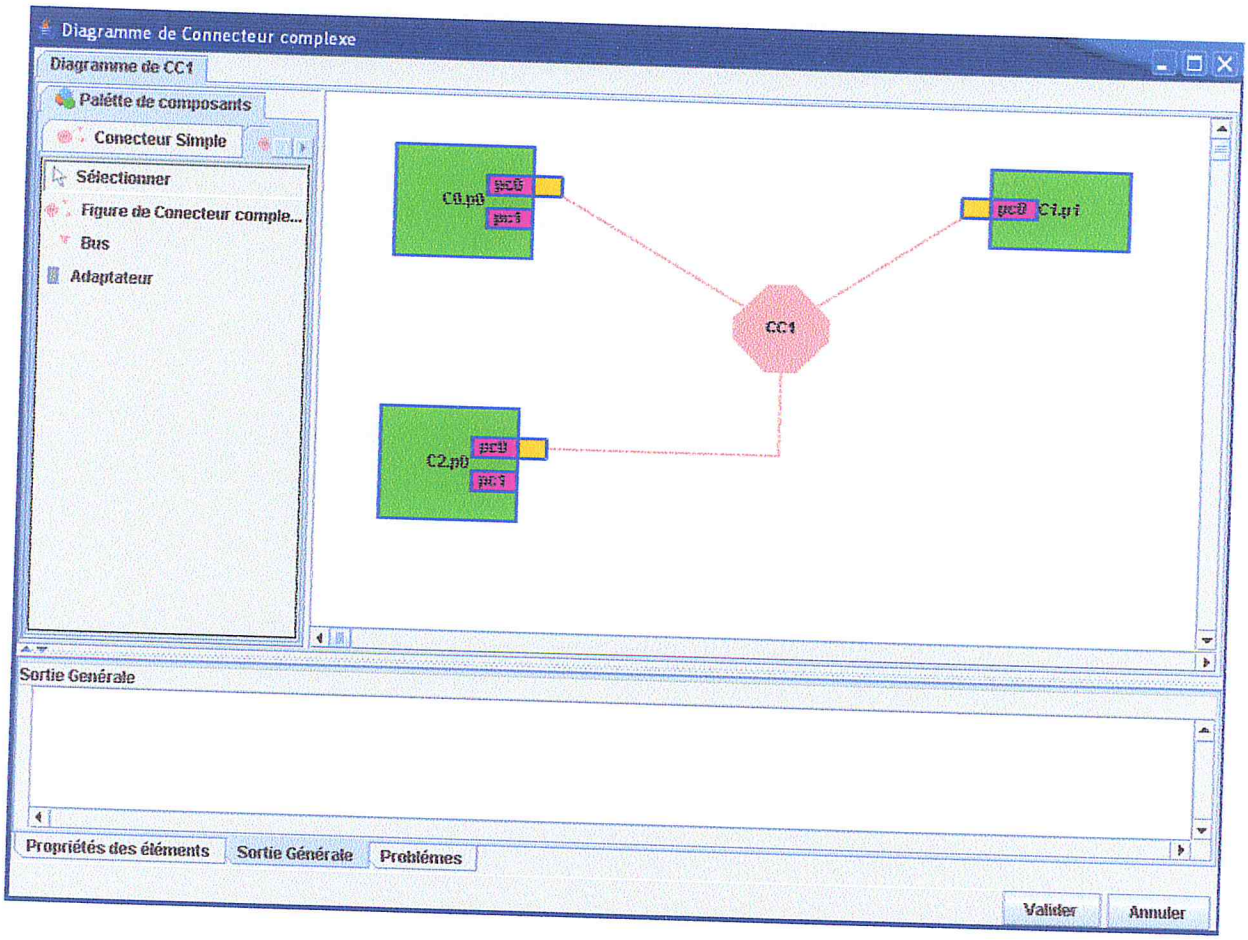


Figure IX.35 : Fenêtre de schéma détaillé d'un connecteur complexe.

IV.1.7. Fenêtre de schéma détaillé d'un sous-connecteur complexe

C'est une zone dédiée à la schématisation d'un sous-connecteur complexe. Elle est partagée en deux parties (Figure IX.36):

- Editeur de diagramme de sous-connecteur complexe
  - La palette de composants

    Elle est de type JTappedPane à trois panneaux (composant, connecteur simple, connecteur complexe).

    Chacun de ces boutons représente un élément graphique du schéma de connecteur complexe en cours de conception (composant, port, point de connexion, connecteur simple, connecteur complexe, bus, adaptateur).
  - La zone de dessin

    Utilisant la gestion glisser déplacer, on peut dessiner touts ces éléments sous les règles et les contraintes d'exploitation suivantes :
    - Un port est présenté avec un et un seul point de connexion.
    - Un adaptateur d'un connecteur simple est connecté à un point de connexion.
    - Un connecteur simple a au moins deux bus.
    - A ce niveau un connecteur complexe n'aura pas un schéma détaillé et s'utilise seulement pour la représentation d'un composant d'interconnexion.
    - Un composant utilisé est de type bien précis (exemple : conversion de données).
- Vue EtatSortie (Propriétés des éléments, Sortie Générale, Problèmes)

  Même description que la vue EtatSortie de la fenêtre principale.

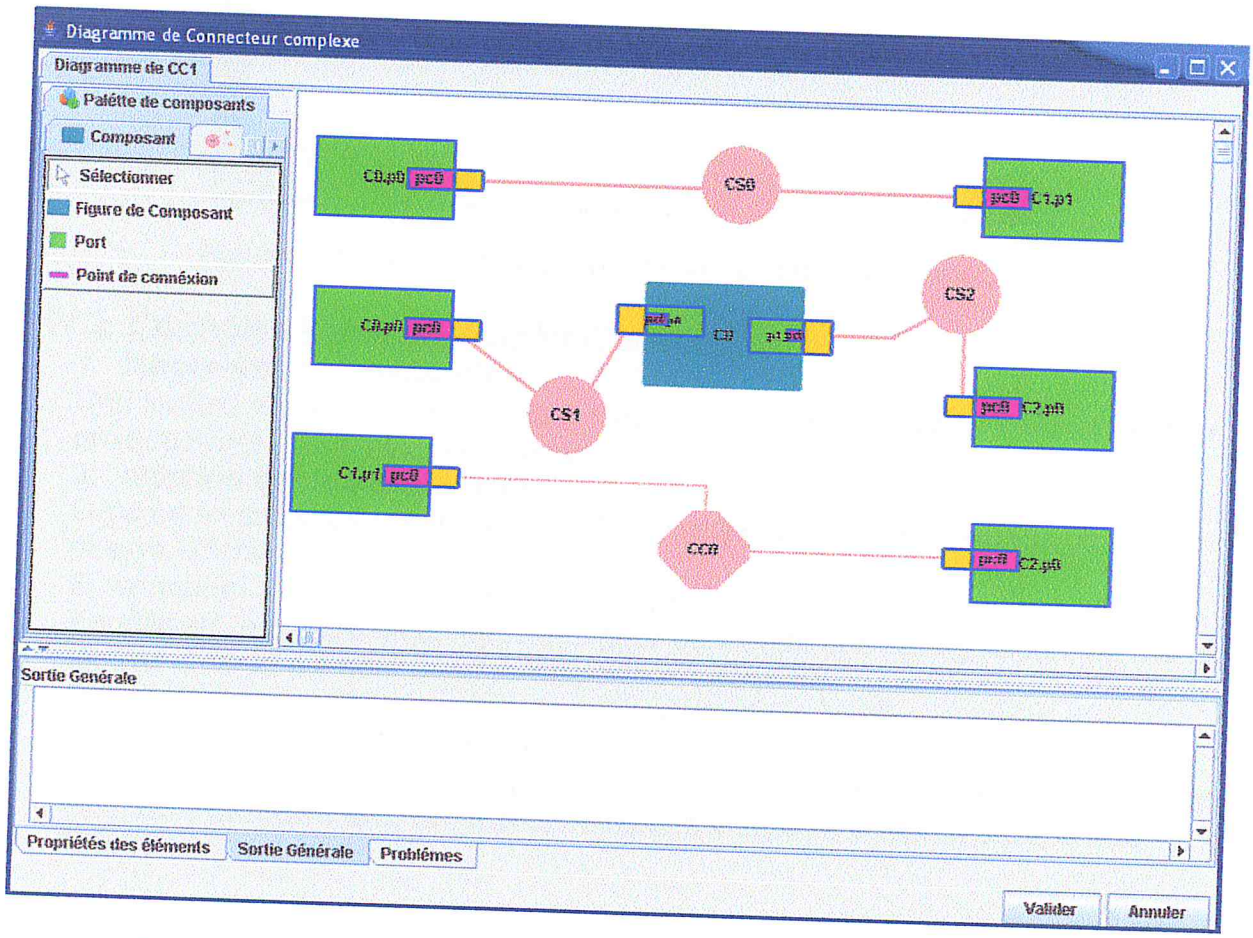


Figure IX. 36: Fenêtre de schéma détaillé d'un sous-connecteur complexe.

4. **validation, génération du code des connecteurs**
   - ***Validation***

     On clique sur le bouton « valider » pour lancer l'opération de validation. Des messages apparaître au niveau le panneau « sortie générale », soit d'erreurs, ou bien que la configuration est bien structurée.

   - ***Génération de code de chaque connecteur***

     Une fenêtre contient tout le code des connecteurs générés est affichée a partir un clic bouton « générer ».

5. ***utilisation de l'aide***

L'aide représente la source de compréhension des fonctionnalités de l'application. Temps en temps, l'utilisateur doit accéder à cette fenêtre pour mieux comprendre le mécanisme de travail avec l'éditeur.

6. ***l'enregistrement des projets***

Soit avec le même nom de création ou bien sous un autre nom. Notre projet dans cet projet1 reste avec le même nom.

7. ***fermeture de l'application***

La fermeture de l'application lance une fenêtre de sauvegarde de notre projet «projet1».

VI. CONCLUSION

Pour conclure, MCC-CASIC est une interface souple, flexible, capable d'offrir à ses utilisateurs l'ensemble des spécificités d'un modèle particulier.
Cela signifie également que MCC-CASIC n'est pas bridée par des contraintes imposées par un langage donné et n'est limitée que par l'imagination de ses utilisateurs.
MCC-CASIC est donc une interface ouverte dans le sens où elle ne se limite plus à un langage existant.
Elle est capable de s'adapter au mieux aux exigences d'un utilisateur et d'offrir ainsi la puissance pour cibler de manière efficace les spécificités nécessaires au développement d'une application donnée.

Bibliographie

[1] Java : les livres : Java 2 le Guide du Développeur, Total Java (Edition Eyrolles 2001) et Swing La Synthèse Développement des interfaces graphiques (Edition Dunod 2001).

[2] Sun Microsystems : entreprise américaine de Développement informatique (Software).
Site : http://java.sun.com/

IV.2. Fonctionnement avancé

Il se présente en deux opérations successives :

IV.2.1. L'opération de validation des connexions graphiques

Voir si les configurations respectent les règles et les contraintes de regroupement des éléments.

On clique alors sur les boutons associés existant dans le menu, la barre d'outils, et à l'intérieur des fenêtres de schéma détaillé de chaque connecteur complexe.

Des messages d'erreurs ou de validation apparaîtrent dans la zone « Sortie générale » de l'application, et l'affichage des problèmes dans la zone « problèmes ».

IV.2.2. L'opération de génération du code de connectivité (code de connecteur)

Un code de connecteur est généré, et leur affichage sera au niveau d'une fenêtre (figure IX.37)

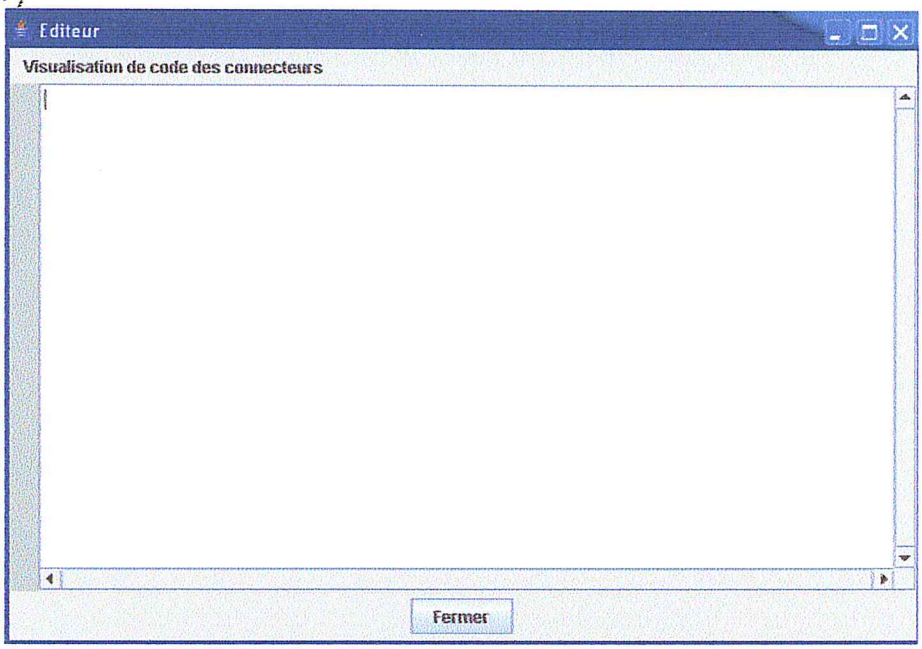


Figure IX.37 : Fenêtre de génération du code des connecteurs

IV.3. Exemple simple et complet d'utilisation

1. **lancement de l'application**

   Une fenêtre apparaît avec des vues vides, un nouveau projet ou l'ouverture d'un projet permet l'initialisation de ces vues.

2. **création de nouveau projet**

   On crée un projet avec le nom *Projet1*. Ce projet avoir comme extension « .mcc », et sera affiché dans le navigateur.

3. **travailler avec l'éditeur de configurations**

   En utilisant la palette de composants pour construire une configuration, de manière à ajouter chaque élément dans sa disposition adéquate.

   La vue Outline montrer un arbre des éléments de la configuration formée.

   Une multitude d'opérations d'édition disponibles pour travailler avec cet éditeur (supprimer, copier, coller, sélectionner tout) et autre (connecter, déconnecter…etc.)

   Une table associée affiche les propriétés des éléments apparaît dans la vue EtatSortie.

Conclusion et Perspectives

Dans un premier temps, nous avons abordé ce mémoire par une étude générale d'un Modèle en couches de connecteurs pour la Conception par Assemblage des Systèmes Informatiques Complexes.
Notre travail nous a permis de se familiariser avec ses concepts.
L'exploitation de ces concepts dans l'architecture logicielle permet d'aboutir à des améliorations remarquables à sa performance.

MCC-CASIC permet à l'architecte de créer facilement des configurations grâce à une interface graphique. L'outil permet à l'architecte de placer des composants et des connecteurs à partir de la palette.

Ces composants logiciels ont été ajoutés à une bibliothèque qui les regroupe.
Ce projet nous a permis, certes, d'approfondir nos connaissances de conception par assemblage des systèmes complexes.
Mais aussi et surtout de comprendre la notion de connectivité entre composants de ses systèmes.

Nous espérons avoir répondu aux besoins posés dans la problématique, tout en souhaitant que le présent travaille réalisés aide les futurs étudiants pour :

- Elaborer des configurations aux architectures logicielles qui représentent de manière claire les systèmes informatiques complexes.
- Donner au connecteur sa place réelle dans l'ensemble des concepts de l'architecture logicielle, qui est une entité de première classe vis-à-vis au concept composant quand on considère sa neutralité
- Exploiter au futur l'aspect transfère de contrôle.
- Exploiter les propriétés non fonctionnelles des connecteurs tels que la performance, plan de déploiement, la sécurité.

Enfin, le champ du travail qui reste à accomplir est considérable. Il ouvre la voie, à notre sens, vers plusieurs perspectives de recherche qui se situent sur deux plans : approfondissement de la recherche réalisée et élargissement du domaine de la recherche. En fait, les deux aspects : type de connecteur et catégories de service ne sont pas traités en leur totalité, mais on s'intéresse à en étudier et exploiter quelques cas de figure, car le domaine est très faste et nécessite des ateliers spécialisés avec un groupe de concepteurs et développeurs assez grand et un matériel puissant. De plus l'aspect transfert de contrôle sera exploité dans des futurs travaux.

Annexe

Eclipse

Eclipse est un projet open-source dont le but est de fournir un environnement de développement en JAVA (Figure). Ce projet a été initié par IBM qui a confié le développement à Object Technology International. Cette société s'est déjà illustrée avec IBM Visual Age.
En reprenant certains traits de ce dernier, Eclipse est devenu un environnement mature, un des leaders dans les ateliers de développement JAVA. Devant l'avancée de Borland sur le marché des IDE, IBM a décidé de réagir. Il s'illustre dans sa nouvelle orientation des logiciels open source. Néanmoins, il fournit aussi un logiciel payant Websphere Studio Application Developer. Il est basé sur le noyau d'Eclipse et les développements de ces deux plate-formes se font en parallèle, de manière centralisée pour assurer une cohérence. Le côté open source est un vecteur de diffusion pour IBM. La majorité des développeurs peuvent se former gratuitement aux produits d'IBM, en particulier dans les universités. Ceux-ci seront donc capables de maîtriser WSAD sans grande peine.
Eclipse est basé sur une architecture très originale qui offre un cadre très extensible. Le socle est composé du workspace et du workbench. Le workspace contient les données, c'est-à-dire les projets sur lequel le développeur travaille. Le workbench est l'interface graphique qui offre plusieurs perspectives. Il est possible de passer d'une perspective à un autre. Une perspective contient plusieurs éditeurs et vues. Il est facile d'en ajouter ou d'en retirer. Chacune des fonctionnalités est ajoutée avec des extensions. Par défaut, tous ceux qui sont nécessaires au développement JAVA sont présents.
Ainsi, Eclipse se démarque profondément des anciens environnements comme IBM Visual Age. C'était un outil de développement très fermé. Il était très compliqué de le faire interagir avec d'autres environnements. Eclipse change totalement d'orientation et mise sur un cadre extensible et le travail collaboratif. Par exemple, CVS offre une interface simple pour partager les ressources sur lesquels une personne travaille et permet de voir les différences entre les versions. La comparaison est basée sur une différence des fichiers source. Il n'est pas possible de faire la même chose pour des fichiers binaires.
Les extensions sont des composants que tout le monde peut développer. Actuellement, il en existe une multitude qui couvre la majorité des besoins pour les programmeurs: développements en J2EE, C++, UML. Ils sont gratuits ou commerciaux. Le noyau d'Eclipse gère l'ensemble des extensions, entre autres leur cycle de vie. Il est possible de charger un nombre important de extensions. L'environnement Eclipse, du fait de sa grande extensibilité, devient lourd. Le chargement d'un extension durant le runtime permet d'optimiser la gestion mémoire. Quand il n'est plus utile, il est déchargé de la mémoire. C'est ce que permet la dernière version d'Eclipse. Il existe aussi un ensemble de lignes de conduite pour la programmation de extensions. Il s'agit de les respecter pour assurer une meilleure cohabitation des extensions dans l'environnement.
Ces « règles » prônent la réutilisation (interfaces graphiques, code source) ou la facilité d'ajout pour les autres contributeurs. Une partie de mon travail pendant ce stage a été de contribuer à Eclipse, en développant mon travail sous la forme d'une extension. Cela représente une opportunité de diffuser le travail accompli actuellement sur la qualité des programmes.

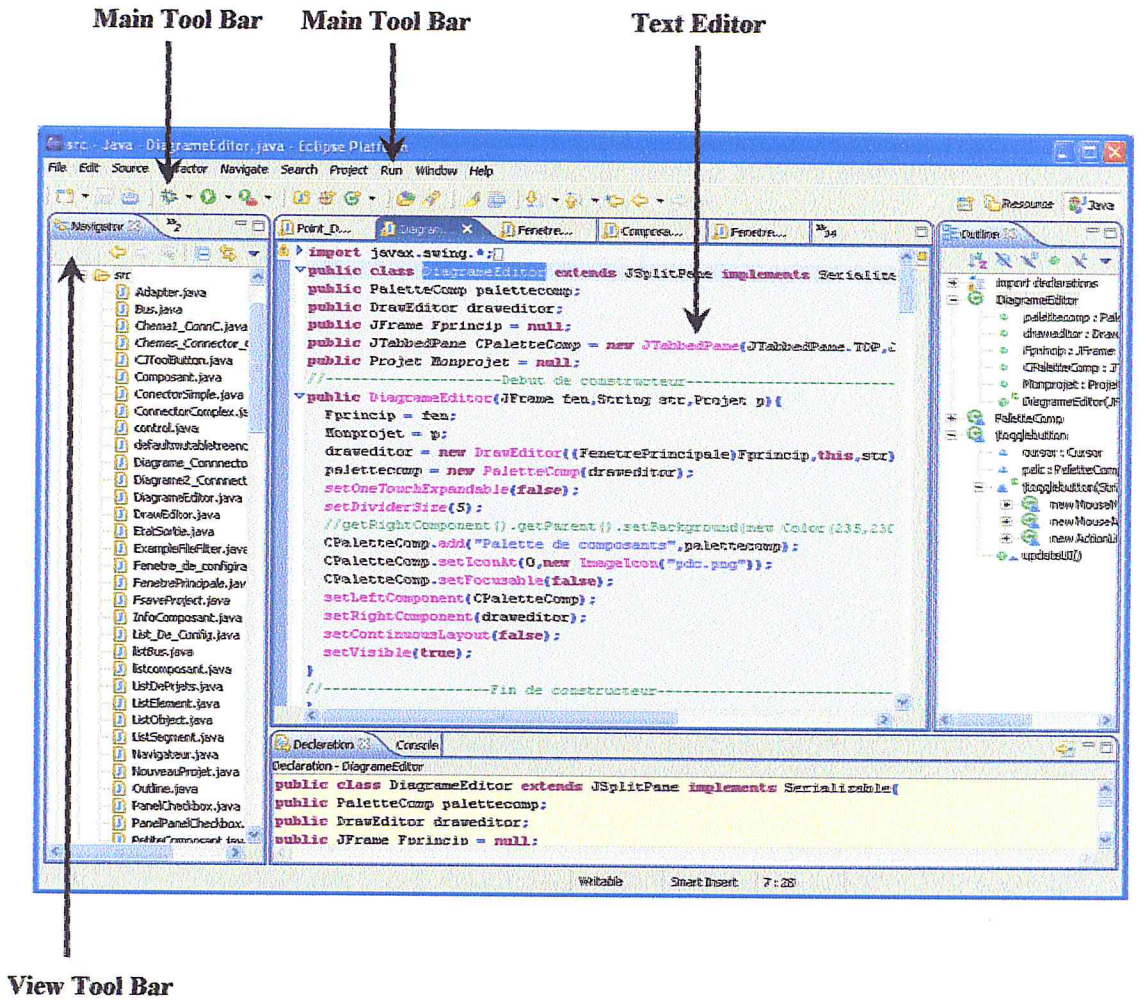


Figure : Interface graphique d'Eclipse