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 d'un environnement de développement intégré dédie à la spécification d'architecture logicielle selon l'approche a composants, basé sur l'ADL *ArchJava*

Présenté par : KHIDER Hadjer **Promoteur : BENNOUAR Djamel**

Organisme d'accueil : sujet interne.

Soutenue le: **, devant le jury composé de :**

- 2004/2005 -

Remerciements

Je tiens à remercier tous les gens qui m'ont aidé à réaliser ce travail de prés ou de loin.

Je remercie mon promoteur M. BOUNNOUAR Djamel pour son aide, sa patience et sa disponibilité.

Je remercie tout particulièrement mes chers parents pour leurs patiences, encouragements et leurs soutiens précieux tout au long ma carrière d'étudiante.

Merci a tous ceux qui de prés ou de loin m'ont aidé et soutenu pour que ce travail soit réalisé.

Dédicaces

Je dédie ce travail a :

Tout d'abord a mes chers parents, a mes chers frères Amine et Youcef, et a ma soeur Yasmina.

A ma bien aimée Nadjia et toute ma famille

A mes chers amis et collègues ainsi qu'à leurs familles

Hadjer

ملخص

عجلة التقدم تتسارع يوم بعد يوم، و الحاجة الى استعمال تقنيات متطورة يزداد يوم عن يوم، هده التقنيات تحتاج الىأنظمة برامجية معقدة وشديدة الدقة. تسيير تعقيدات الأنظمة البرامجية صار في الوقت الحالي تحديا يواجهه صناع الأنظمة البرامجية.عواقب القرارات المتخدة قد يكون له آثار وخيمة علىمشروع انجازالبرامج، خاصة في مراحل متقدمة من انجازه. اعادة انجازنظام برنامجي من أجل ادماج التقنيات الحديثة في البرمجة قد يصبح ضروريا و حتى اجباريا في بعض الأحيان.امكانية حدوث هدا الادماج و تكلفته يتعلقان بالطريقة المتبعة لتطبيقه و بالقرارات المتبعة في المراحل الاولىمن دورة حياته.

الحل الممكن هو اتباع سياسة هندسية في مراحل انجاز النظام البرامجي. الوصف الهندسي هو مرحلة هامة في دورة حياة انجاز البرامج و عامل اساسي في نجاحه.

هناك اتجاهان في وصف هندسة الأنظمة البرامجية، الاتجاه الاول يعتمد على ما يعرف بالأداة أو "Objet"، لكن هدا الاتجاه يتضمن نقائص وسلبيات كثيرة، الاتجاه الثاني يعتمد على المكون أو"Composant" ، عن طريق اللغات الخاصة بالوصف الهندسي للبرامج.

هدفنا هو انجاز محيط تطويرمخصص لوصف الهندسة البرامجية للأنظمة البرامجية، هدا المحيط المطور يتميز بسهولةالاستعمال ويمكن مستعمله من امكانية اعادة استعمال المكونات البرامجية من اجل ادماجها في برامج اخرى مما له من اهمية كبرى في الوقت الحالي. كما يحتوي هدا الاخير على أداة لتنفيذ البرامج من اجل التحقق من توافق الهندسة البرامجية مع التطبيق.

كلمات مفتاح : هندسة برامجية، مكوّن، لغات الوصف الهندسي، موصّل، وصف، 'Archjava'، نظام.

Résumé

L'évolution de monde se précipite jour après jour. Le recours aux technologies développés devient de plus en plus indispensable. Ces technologies vont faire appel à des systèmes logiciels de plus en plus complexes. La gestion de la complexité d'un logiciel est aujourd'hui un défi auquel les développeurs des systèmes logiciels sont confrontés. Les impactes des décisions prises sur un projet notamment a un stade assez avancé de sa mise en œuvre peut être catastrophique. La refonte d'un logiciel pour intégrer des nouvelles techniques d'implémentation peut devenir nécessaire et même obligatoire. La faisabilité et le coût de cette intégration vont dépendre du processus suivi pour sa mise en place et des décisions prise dans les premières étapes de cycle de vie. Une solution possible est de suivre une approche architecturale dans le processus de développement des gros systèmes logiciels.

La spécification architecturale est une étape importante dans le cycle de vie de développement d'un logiciel et un facteur critique dans sa réussite. Deux tendances se distinguent pour la spécification architecturale. La première est basée sur l'approche objet mais celle-ci comprend plusieurs inconvénients. La deuxième tendance est basée sur l'approche a composants. Dans l'approche a composants la spécification de l'architecture logicielle est réalisée par des ADL (Architecture Description Language) ou des langages de description d'architecture logicielle.

L'objectif de ce travail est de concevoir et de réaliser un environnement intégré de développement (IDE) pour l'architecture logicielle. Cet IDE est basé sur le langage de description d'architecture logicielle Archjava. Qui permet de spécifier une architecture d'un système logiciel, en terme de composants et de connecteurs. Il dispose de capacité permettant la réutilisation efficace de composants, connecteurs et d'architecture. Et offre une plate forme d'exécution pour vérifier l'uniformité de l'architecture avec l'implémentation.

Mots clés : Architecture logicielle, composant, ADL, connecteur, spécification, Archjava, système.

Abstract

The evolution of world precipitates day after day. The recourse to developed technologies becomes increasingly necessary.These technologies will require increasingly complex software systems.The management of the complexity of software is today a challenge, which the developers of the software systems are confronted.

The impacts of decisions taken on a project can be catastrophic. The redoing of software to integrate new techniques of implementation can become necessary and even obligatory. The feasibility and the cost of this integration will depend on the process followed for its installation and on the decisions taken in the first stages of cycle of life. A solution is possible is to follow an architectural approach in the process of development of the large software systems.

The architectural specification is an important step in the cycle of life of develops of software and a critical factor in its success. Two tendencies are distinguished for the architectural specification, first is based on the approach object but this one includes several disadvantages. The second tendency is based on the approach component. In the approach component the specification of software architecture has realized by *ADLs* or languages of description of software architecture.

The objective of this work is to design and produce an integrated environment of development (*IDE*) for software architecture; This *IDE* is based on the *ADL Archjava*. It allows specifying architecture of a software system in term of components and connectors.

This IDE provided services and tools that facilitate the architectural specification of a software system. It is characterized by its ease of use. Without forgetting that it makes it possible to specify architecture of a system in term of components and connectors. Preserve the concept of re-uses and offers an execution platform in order to check the uniformity of this architecture with the implementation.

Key words:Software architecture, component, *ADLs*, connector, specification, *Archjava*, system.

Table des matières

Table des figures

CHAPITRE I :

Introduction générale

Chaque jour, le monde évolue et son évolution implique l'utilisation des moyens de plus en plus puissants et par la suite fait appel a des systèmes logiciels de plus en plus complexes. La maîtrise de la complexité d'un logiciel est aujourd'hui le premier défi auquel font face les développeurs, une erreur peut être catastrophique sur un projet logiciel surtout à un stade assez avancé de sa mise en place et pourrait entraîner tout simplement l'arrêt du projet et causer ainsi des pertes énormes.

Un autre défi dans le développement du logiciel est provoqué par l'avancée technologique dans le domaine du logiciel lui-même, avec l'apparition notamment de nouvelles techniques d'implémentation. La refonte d'un logiciel pour l'intégration des nouvelles techniques pourrait devenir nécessaire. Sa faisabilité et son coût dépendront beaucoup de processus suivi pour sa mise en place et des décisions prises notamment dans les premières étapes de son cycle de vie.

Dans le domaine de l'ingénierie où la durée de développement est un facteur qui conditionne le coût économique d'un projet, les besoins de réutilisation des logiciels deviennent de plus en plus nécessaire. Pour les rendre réutilisables, il faut donc concevoir des composants logiciels, indépendamment du contexte où ils sont censés évoluer par la suite.

Face aux difficultés mentionnées précédemment, une solution est possible, c'est la définition d'une architecture logicielle de système. Cette architecture permettra ensuite d'étudier et d'estimer dans les premières étapes de cycle de vie de logiciel les impactes des décisions prises sur la forme finale du logiciel, la satisfaction des besoins aux quels doit répondre, ainsi la gestion des difficultés rencontrées par les concepteurs de gros logiciels et permette la réutilisation des logiciels. La maîtrise de la complexité des gros systèmes et la mise en œuvre des études prévisionnelles doit commencer par une modélisation centrée sur l'architecture logicielle.

Une architecture logicielle à base de composants décrit l'ensemble des composants qui la composent, donne la définition de leur assemblage et prend en compte les structures d'accueil nécessaires pour le déploiement et l'exploitation du système résultant. 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, La complexité des logiciels est prise en compte, les possibilités de réutilisation sont augmentées.

La spécification architecturale est une étape importante dans le développement d'un logiciel et un facteur critique dans sa réussite, C'est pour cette raison qu'elle doit être spécifiée de manière précise et commune. Deux tendances se distinguent pour la spécification architecturale. La première est basée sur l'approche objet mais celle-ci comprend plusieurs inconvénients. La deuxième tendance est basée sur l'approche a composants. Dans l'approche a composants la spécification de l'architecture logicielle est réalisée par des ADL (Architecture Description Language) ou des langages de description d'architecture logicielle.

La première approche comprend plusieurs inconvénients que nous résumons dans le fait que dans l'approche objet le seul mécanisme de connexion supporté est celui de l'envoi de messages entre les objets, une telle connexion est vue comme un niveau dans le contexte des ADL. Un autre inconvénient est que l'approche objet ne supporte pas les concepts de base de l'architecture logicielle tels que le concept de connecteur. On peut dire que l'approche objet n'offre pas une voie claire et cohérente pour la modélisation des concepts de base de l'architecture logicielle.

Les langages de description d'architecture (ou ADL pour *Architecture Description Language*) sont apparus pour permettent la définition d'un vocabulaire précis et commun pour les acteurs devant travailler autour de la spécification liée à l'architecture (Architectes, concepteurs, développeurs, intégrateurs et testeurs). Ils spécifient les composants de l'architecture de manière abstraite sans entrer dans des détails d'implantation, définissent de manière explicite les interactions entre les composants d'un système et fournissent un support de modélisation pour aider les concepteurs à structurer et composer les différents éléments de leurs architectures logicielles.

En fait, les ADLs sont un support pour la description de la structure de l'application. Ils offrent des facilités de réutilisation des composants et des moyens de description de la Composition par description des dépendances entre composants et des règles de communication à respecter.

Il est possible d'utiliser des outils formels ou des environnements dont l'objectif est de fournir un ensemble d'outils et de services pour faciliter le développement et la spécification d'architectures logicielle avec les différents langages de spécification d'architecture logicielle avec la génération des parties de code automatiquement.

Le travail présenté dans ce mémoire a pour objectif de concevoir et de réaliser "Un environnement de développement intégré (IDE)" dédié à la spécification d'architecture logicielle, avec le langage de description d'architecture logicielle l'ADL *ArchJava*[1].

Ce mémoire est organisé en quatre chapitres, dont le contenu sera détaillé ci-dessous :

Nous avons commencé par le chapitre I qui constitue une introduction générale, ensuite le Chapitre II où nous allons décrire les concepts généraux de l'architecture logicielle, et les différents langages de description d'architecture logicielles existants.

Quant aux chapitre III, IV,Vet VI, ils serons consacrés à la démarche de développement de l'outil de spécification, en commençant par une analyse des besoins dans le chapitre III, puis la conception dans le Chapitre VI, la réalisation ou implémentation dans la chapitre V, et on conclut cette démarche avec la partie teste et validation présenté dans la chapitre VI.

Dans le chapitre VII, nous allons conclure ce travail on rappelant l'intérêt de ce travail et son apport vis-à-vis le domaine d'architecture logicielle, ainsi nous proposons quelques perspectives au future a fin d'enrichir et améliorer ce travail.

[1] : Pour plus d'information sur l'ADL **Archjava** veuillez consulter l'annexe A

CHAPITRE II : Architecture logicielle (Concepts généraux)

II.1- Introduction :

Dans le chapitre d'introduction générale, nous avons parlé des nombreux problèmes auxquels font face les développeurs dans le domaine de développement du logiciel, tels que la maîtrise de la complexité des logiciels conçus, la refonte d'un logiciel pour intégrer des techniques nouvelles, sans oublier le facteur de temps qui est un facteur critique dans l'évaluation de coût économique d'un projet, ce qui rond le besoin de réutiliser les logiciels de plus en plus nécessaire. Pour palier a tous ces problèmes, nous sommes arrivés à une solution où l'architecture logicielle joue un rôle vital.

Dans ce présent chapitre, nous allons parler de notion d'architecture logicielle, sa définition, son intérêt, de ses éléments fondamentaux, et nous allons conclure ce chapitre par la spécification de l'architecture logicielle.

II.2- Notion d'architecture logicielle :

L'architecture logicielle est un domaine récent de génie logicielle qui a reçu une Attention particulière ces dix dernières années, les développeurs de logiciels en pris conscience que c'est un facteur critique dans la réussite de développement des projets logiciels car elle facilite la maintenance et l'évolution du logiciel et autorise le développement de système volumineux en termes de taille mais aussi de complexité, elle offre aussi une vue globale de l'application, et le rond ainsi beaucoup plus maîtrisable à son concepteur.

Le terme « **architecture logicielle** » a souvent été utilisé dans deux contextes :

- Dans le premier contexte, l'architecture logicielle à base de composants décrit l'ensemble des composants qui la composent, donne la définition de leur assemblage et prend en

compte les structures d'accueil nécessaires pour le déploiement et l'exploitation du système résultant.

- Dans le deuxième contexte, l'architecture est utilisée pour désigner une structure d'exécution 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.

Nous pouvons dire que la définition de l'architecture d'un système correspond à l'établissement du plan de construction du logiciel. 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. En effet la modification d'un plan est plus simple que la modification d'un système complet.

Pour mieux contrôler la complexité du logiciel, il est nécessaire d'avoir un niveau d'abstraction élevé et de disposer de modèle qui s'approche du modèle mental du développeur, et c'est à partir de ce modèle que l'architecture logicielle a pris son départ, Ce modèle est souvent représenté à un haut niveau d'abstraction par un ensemble de blocs interconnectés.

Dans un premier temps, 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 itérative et incrémental. Le processus de raffinement mène l'architecte à raisonner au sujet de son architecture dans plusieurs domaines de conception, tels que le domaine de l'organisation, la gestion du code source et le domaine de déploiement du logiciel.

L'objectif de l'architecture logicielle est de fournir les méthodes et outils permettant de prendre en charge cette démarche intuitive de construction de systèmes informatiques. Elle assure le suivi du processus itératif et incrémental de raffinement jusqu'à atteindre le domaine de l'implémentation, du déploiement et de la maintenance. Enfin, elle permettra d'assurer que l'architecture définie à un haut niveau d'abstraction correspond fidèlement à la réalisation..

L'architecture logicielle d'un système s'intéresse non seulement à la structure et au comportement, mais également à des contraintes et a des compromis d'utilisation, de fonctionnalité, des performances, d'adaptabilité, d'économie, de réutilisation et de technologie, sans oublier les questions d'esthétiques.

II.3-Définition de l'architecture logicielle :

La discipline de l'architecture logicielle tendant à devenir une discipline à part entière, il est devenu nécessaire de bien fixer son vocabulaire et de réduire les ambiguïtés de ses termes.

Si, à ce jour, le terme « **architecture logicielle** » ne possède pas une définition unanime de toute la communauté scientifique travaillant dans ce domaine, il n'en demeure pas que le champ d'activité de cette discipline commence à être bien cernée.

Medvidovic [1] définit l'architecture logicielle comme suit :

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.

Dans[2], l'auteur avance la définition suivante: L'architecture d'un programme ou d'un système informatique est la structure (ou les structures de ce système) qui est composée de composants logiciels, les propriétés externes visibles de ces composants et les relations entre eux.

Malgré ces diverses définitions, il en ressort les aspects communs suivants :

1) l'architecture logicielle se concentre sur la définition de la structure de gros systèmes suivant plusieurs vues (structurelle, comportementale, etc.),

2) Les éléments fondamentaux de définition d'architecture sont les composants et les interconnexions entre ces composants qui définissent une topologie appelée configuration,

3) Une architecture prend en considération et organise dès le début du cycle de vie du logiciel, les différents besoins et les objectifs à atteindre. Elle montre à travers les composantes utilisées et leurs topologies d'interconnexion comment ont pris en charge les besoins et les exigences du logiciel.

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 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.

II.4-Eléments fondamentaux de l'architecture logicielle :

Le *composant*, le *connecteur* et la *configuration* représentent les éléments fondamentaux de spécification de l'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 est aussi une notion 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 a titre d'exemple une architecture 2-tiers est un style d'architecture.

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, de telles propriétés peuvent être attachées globalement au système ou individuellement aux différents éléments architecturaux (composants, connecteurs, etc.)

II.4.1- Le concept de composant :

Un composant logiciel est une entité responsable de la réalisation d'une ou plusieurs fonctionnalité bien précise dans une architecture à un certain niveau d'abstraction. C'est une unité de calcul ou de stockage à laquelle est associée une unité d'implantation. Le composant est aussi un lieu de calcul et possède un état, Il inter-agit avec les autres composants pour réaliser un ou plusieurs objectifs d'une architecture.

Un composant peut être très simple, comme une fonction, un objet d'une classe C++ ou composé ou complexe; on parle alors dans ce dernier cas de composite 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.

On définit deux parties dans un composant, une première partie, dite extérieure, correspond à son interface, elle comprend la description des interfaces fournies et requises par le composant, elle définit les interactions du composant avec son environnement. La seconde partie correspond à son implantation et permet la description du fonctionnement interne du composant.

Les caractéristiques globales d'un composant [4] [6] [7] sont les suivantes :
- l'interface,
- le type,
- la sémantique,
- les contraintes,
- les propriétés non fonctionnelles.

- ***L'interface d'un composant :***

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 ou le code d'un message, l'interface c'est un moyen d'expression des liens du composant ainsi que ses contraintes avec l'extérieur.

Une interface est composée d'un certain nombre de points d'interaction appelés ports de communication (en Achjava et C2 [1]) ou players (en UNICON), 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.

- ***Le type d'un composant :***

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 un même architecture, soit dans des architectures différentes.

- ***La sémantique d'un composant :***

La sémantique permet de décrire le comportement du composant, le comportement peut être considéré sous deux formes [9] :

Un comportement dit statique dans lequel nous nous intéressons aux états discrets du composant à des instants bien précis, ce type de comportement n'exprime pas comment un tel état a été atteint. Le comportement dynamique quant à lui, explique comment un état est atteint à partir d'un autre.

La sémantique d'un composant s'apparente à la notion de type dans le modèle orienté objet.

- ***Les contraintes d'un composant :***

Les contraintes des composants spécifient les conditions et définissent les limites d'utilisation (d'exploitation) d'un composant et ses dépendances intra composants comme par exemple la spécification de la synchronisation entre composants d'une même application.

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.

- ***Les propriétés non fonctionnelles d'un composant :***

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 etc.) .
Cette séparation permet la simulation du comportement d'un composant à l'exécution dès la 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.

II.4.2- Le concept de connecteur :

Pendant longtemps les connecteurs ont été considérés comme implicite et leurs sémantiques étaient souvent incorporées dans les composants. Cette situation est devenue inadaptée avec l'apparition de nouvelles technologies permettant de construire des systèmes de plus en plus complexes, composés d'éléments totalement indépendants et provenant de différentes sources. L'incorporation de toute la mécanique d'interconnexion, au niveau du composant, va rendre ce dernier connectable à un nombre assez réduit de composants, et ne peut être mis en œuvre que dans des situations particulières et trop contraignantes.

Actuellement, il est largement accepté en architecture logicielle qu'un connecteur est un élément à part, indépendant de tout composant.

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, par exemple, un connecteur peut décrire des interactions simples de type appel de procédure ou accès à une variable partagée, mais aussi des interactions complexes telles que des protocoles d'accès à des bases de données avec gestion des transactions, la diffusion d'événements asynchrones ou encore l'échange de données sous forme de flux.

Un connecteur peut supporter des concepts très simples comme l'accès à une variable globale, l'appel de procédure, un fichier, une « pipe » ou des concepts complexes comme une requête SQL ou un protocole client-serveur tel que FTP ou HTTP. Un connecteur peut aussi correspondre à toute une infrastructure de communication tels que les RMI ou CORBA.

Un connecteur comprend également deux parties. La première correspond à la partie visible du connecteur, c'est-à-dire son interface, qui permet la description des rôles des participants à une interaction, la seconde partie correspond à la description de son implantation.

Le connecteur permet également de vérifier l'intégrité de la communication, c'est-à-dire de vérifier que les composants peuvent être connectés. Ainsi, il permet la réutilisation et l'adaptation des interfaces de composants déjà existants que l'on cherche à relier.

Le rôle primordial des connecteurs consiste à faire le nécessaire pour assurer une communication correcte entre les composants. La sémantique d'un connecteur est indépendante de la logique de l'application ce dernier aspect ouvre la voie vers les possibilités de changements statiques ou dynamiques des connecteurs sans aucun impact sur la fonctionnalité de l'architecture du logiciel.

Six caractéristiques importantes définies par Medvidovic et Taylor [4] [6] [7] sont à prendre en compte pour spécifier de manière exhaustive un connecteur. Ces caractéristiques sont les suivantes :
- l'interface,
- le type,
- la sémantique,
- les contraintes,
- la maintenance évolutive d'un connecteur,
- les propriétés non fonctionnelles.

- **L'interface d'un connecteur:**

L'*interface* d'un connecteur indique les services fournis et requis par les composants que le connecteur est capable de transporter, elle aussi définit les points d'interactions entre connecteurs et composants. Certains ADLs nomment ces points d'interactions comme étant des rôles. Un rôle (point d'interaction) définit quels sont les participants (« *players* » ou ports) pouvant participer dans une connexion.

Un connecteur transporte donc un concept émanant d'un port d'un composant vers un autre port de composant. Le port permettra d'indiquer au composant sur quel type de connecteur il peut être raccordé ou de manière plus simple, quel rôle peut-il jouer dans une connexion dont les concepts sont véhiculés par le connecteur. L'interface ne décrit pas des services fonctionnels comme ceux du composant mais s'attache à définir des mécanismes de connexion entre composants.

- **Le type de connecteur :**

Le type d'un connecteur correspond à sa définition 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. Par exemple, la spécification d'un connecteur de type RPC qui relie deux composants définit les règles du protocole RPC.

- **La sémantique :**

Le raccordement d'un composant à un connecteur nécessite la présence d'un port de raccordement compatible avec le connecteur. La nature de l'information véhiculée par un connecteur, le sens de la communication (unidirectionnelle, bidirectionnelle), le mode de transfert (alternat, simultanée) dépendra de sa nature et de la nature de la connexion.
Notons que la sémantique de l'interaction est un élément des composants en connexion et non pas des connecteurs. Ainsi un appel simple de procédure avec passage de paramètres par référence peut être considéré comme un connecteur bidirectionnel reliant deux procédures et transportant des valeurs d'un certain type de donnée, ce connecteur fonctionne à l'alternat.

- **Les contraintes :**

Les *contraintes* sur les connecteurs expriment les conditions et les limites d'exploitation. Ces contraintes peuvent être spécifiées à toutes les instances d'un type ou redéfinies et enrichies pour des instances bien précises.

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. Par exemple, le nombre maximum de composants interconnectés à travers le connecteur peut être fixé et correspond alors à une contrainte.

- **la maintenance évolutive d'un connecteur :**

Le changement des propriétés (interface, comportement) d'un connecteur doit pouvoir évoluer sans perturber son utilisation et son intégration dans les applications existantes. Il s'agit de maximiser la réutilisation par modification ou raffinement des connecteurs existants. Un ADL donnant la possibilité de définir un connecteur doit donc permettre de le faire évoluer de manière simple et indépendante en utilisant le sous typage ou des techniques de raffinement.

- **les propriétés non fonctionnelles :**

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. La spécification de ces propriétés est importante puisqu'elle permet de simuler le comportement à l'exécution, l'analyse, la définition de contraintes et la sélection des Connecteurs.

II.4.3- La configuration d'architecture:

La configuration d'architecture d'une application est la topologie indiquant la façon correcte avec laquelle chaque composant est exploité, les propriétés architecturales de connectivité c'est à dire comment les différents composants sont interconnectés, quels sont les connecteurs utilisés, comment ces connecteurs sont utilisés, ainsi que des propriétés de concurrence et de répartition dans certains ADLs.

Une configuration définit la structure et le comportement d'une application formée de composants et de connecteurs. Une composition de composants, appelée dans certains contextes composites est une configuration.

- La configuration structurelle de l'application correspond à un graphe connexe des composants et des connecteurs formant l'application, elle détermine les composants et les connecteurs appropriés à l'application et vérifie la correspondance entre les interfaces des composants et des connecteurs.

- La configuration comportementale, quant à elle, modélise le comportement en décrivant l'évolution des liens entre composants et connecteurs, ainsi que l'évolution des propriétés non fonctionnelles comme les propriétés spatio-temporelles ou la qualité de service, 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.

Une configuration doit être un support de communication entre plusieurs acteurs d'un projet ce qui exige quelle doit être claire, et facile à comprendre.

Plusieurs caractéristiques importantes définies par Medvidovic et Taylor [4] [6] [7] pour évaluer la configuration d'une architecture du logiciel, ces caractéristiques sont les suivantes :
- un formalisme commun,
- la composition,
- le raffinement et la traçabilité,
- l'hétérogénéité,
- l'évolution de la configuration,
- les contraintes,
- les propriétés non-fonctionnelles.

- ***Un formalisme commun :***

Une configuration doit permettre de fournir une syntaxe simple et une sémantique permettant de :

- Faciliter la communication entre les différents partenaires d'un projet (concepteurs, développeurs, testeurs, architectes),

- Rendre compréhensible la structure d'une application à partir de la configuration sans entrer dans le détail de chaque composant et de chaque connecteur,

- Spécifier la dynamique d'un système, c'est-à-dire l'évolution de celui-ci au cours de son exécution.

- ***La composition :***

La définition de la configuration d'une application doit permettre la modélisation et la représentation de la composition à différents niveaux de détail. La notion de configuration spécifie une application par composition hiérarchique. Ainsi un composant peut être composé de composants, chaque composant étant spécifié lui-même de la même manière, jusqu'au composant dit primitif, c'est-à-dire non décomposable.
L'intérêt de ce concept est qu'il permet la spécification de l'application par une approche descendante par raffinement, allons du niveau le plus général formé par les composants et les connecteurs principaux, définis eux-mêmes par des groupes de composants et de connecteurs, jusqu'aux détails de chaque composant et de chaque connecteur primitif.

- ***Le raffinement et la traçabilité:***

La configuration est également un moyen de permettre le raffinement de l'application d'un niveau abstrait de description général vers un niveau de description de plus en plus détaillé, et ceci, à chaque étape du processus de développement (conception, implantation, déploiement).

- ***L'hétérogénéité :***

La configuration d'une architecture doit permettre le développement de grands systèmes avec des éléments préexistants de caractéristiques différant indépendamment du langage de programmation, du système d'exploitation et du langage de modélisation.

- ***L'évolution de la configuration :***

La configuration doit être capable d'évoluer pour prendre des nouvelles fonctionnalités impliquant une modification ou une évolution de la structure de l'application. Elle doit permettre de faire évoluer l'architecture d'une application de manière incrémentale, c'est à dire par ajout ou retrait de composants et des connecteurs.

- ***Les contraintes***

Les contraintes liées à la configuration viennent en complément des contraintes définies pour chaque composant et chaque connecteur. Elles décrivent les dépendances entre les composants et les connecteurs et concernent des caractéristiques liées à l'assemblage de composants.
La spécification de ces contraintes permet de définir des contraintes dites globales, s'appliquant à tous les éléments de l'application.

- ***Les propriétés non fonctionnelles***

Certaines propriétés non fonctionnelles ne concernant ni les connecteurs et ni les composants doivent être exprimés au niveau de la configuration. Ces contraintes sont liées à l'environnement d'exécution.

II.5- La spécification de l'architecture logicielle :

II.5.1-Introduction :

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 deux approches [7][14] :

Une approche basée sur le modèle objet (UML) et une approche basée sur le modèle à composants (ADL).

Dans l'approche objet, la description architecturale est réalisée à l'aide du langage UML, qui est un langage très largement accepté, tant au niveau des laboratoires de recherche qu'au niveau industriel.

UML est un langage très riche, il représente aujourd'hui le langage de modélisation pour plusieurs méthodes de conceptions objet, il n'était pas destiné dans ces débuts pour la description d'architecture logicielle d'ailleurs, les concepts qu'en globe UML font montrer clairement le faible participation des chercheurs en architecture logicielle à la définition du langage UML.

L'adoption du langage UML par une grande communauté de chercheurs, praticiens et industriels n'allaient pas laisser les chercheurs en architecture logicielle sans réaction, un nombre important de travaux ont été réalisés dans le but de faire un UML un ADL, parmi ces travaux nous distinguons deux principales approches :

Dans la première, l'idée de base consiste à spécifier une architecture basée sur la modélisation des éléments de base de l'architecture logicielle (composant, connecteur, configuration, etc.) [14] [15][16].La deuxième approche se concentre sur la mise au point de méthodes pour l'architecture logicielle basée uniquement sur UML[17].

Pour la première approche, l'aspect le plus important se situe au niveau de la modélisation des connecteurs et des ports. Les connecteurs sont des éléments de modélisation indépendants de la fonctionnalité du système à modéliser et qui n'influent pas sur la fonctionnalité de l'architecture dans laquelle ils sont instanciés. Les connecteurs peuvent avoir un impact sur les propriétés non fonctionnelles tels que la performance et la sécurité. Le langage UML ne supporte pas le concept de connecteur dans le sens des ADL. Le seul mécanisme de connexion supporté de manière implicite est celui de l'envoi de messages entre les objets, une telle connexion est vue comme un niveau dans le contexte des ADL.

Les expériences précédentes, ainsi que d'autres, ont montré que de manière générale, la description d'une architecture en se basant sur la modélisation des éléments de base de l'architecture logicielle nécessite un travail important d'adaptation, qui en général n'est pas complet et comporte aussi des insuffisances.

Dans la deuxième approche, le travail consiste à définir une démarche pour l'architecture logicielle basée uniquement sur UML. Dans cette approche, de nouvelles vues, autre que celle d'UML sont définies [17]. Les vues UML sont intégrées pour former une vue globale et cohérente du système. A titre d'exemple, Hofmeister [17] a proposé une démarche qui consiste à voir le système selon quatre vues, chacune d'elles représentant un aspect bien précis d'une architecture.

Les quatre vues définies sont :

1) ***la vue conceptuelle*** dans laquelle la structure du système est présentée sous forme de composants et de connecteurs, appelées configuration avec les éléments du domaine,

2) ***La vue des modules*** dans laquelle les composants et les connecteurs sont assignés à des modules ou des sous-systèmes,

3) ***La vue exécution*** qui définit la structure du système en fonctionnement,

4) ***La vue du code*** qui définit l'organisation des éléments d'implémentation (code source, code objet, librairie, etc.).

Malgré que la deuxième approche permette de construire des vues cohérentes, elle n'apporte pas de solutions à la modélisation des éléments architecturaux de base. De plus, la mise au point d'une vue unique, même englobant les autres vues, sort du contexte standard et risque de ne pas être supporté par les outils basés sur le standard UML.

Nous pouvons résumer les facteurs qui vont rendre délicate la spécification d'architecture logicielle avec UML dans :

- la prédisposition des architectes de systèmes à raisonner dans le paradigme composant /Port/Connecteur,
- malgré que le langage UML soit doté d'une puissance de modélisation, peut être plus grande que celle des ADL, le mécanisme d'abstraction qu'il fournit ne correspond pas efficacement au modèle mental à partir duquel le raisonnement d'un architecte débute et
- UML n'offre pas une voie claire et cohérente pour la modélisation des concepts de base de l'architecture logicielle.
- L'adaptation d'UML aux méthodes de conception de l'architecture logicielle semble possible mais au prix d'un travail non négligeable d'adaptations. Ces difficultés sont dues principalement au méta-modèle d'UML qui n'est pas adapté à supporter l'expression des concepts de l'architecture logicielle.

Par contre, les ADL ont été conçus dans l'esprit d'une approche bien précise pour atteindre un but bien précis. Décrire l'architecture à un haut niveau d'abstraction dans un nombre de vues bien définies, puis appliquer un processus de raffinement sur toutes les vues, pour atteindre un ou plusieurs objectifs bien déterminés.

Dans l'approche des modèles à composants, la description est réalisée par des langages de description d'architectures (ADL), les ADL, sont des outils de spécification et de représentation formelle de l'architecture logicielle, ils sont arrivés pour la description d'architectures logicielles. Leur évolution est toujours dirigée pour une prise en charge de plus en plus meilleure de la description de l'architecture logicielle.

Nous s'intéresserons dans ce document a l'approche basée sur le modèle a composants pour ces nombreux avantages cités au préalable ainsi pour d'autre raison qui se résume dans le fait que le système réalise est basé sur cette approche.

II.5.2 -Les langages de description d'architectures (ADL) :

L'évolution des langages de description est passé par deux importantes étapes selon les objectifs couverts par ces ADLs, à leur naissance, au début des années 80, l'objectif Principal couvert par ces ADL se basait essentiellement sur l'aspect implémentation des applications, il consistait à identifier les atomes d'exécution d'une application, pour en gérer sa distribution sur plates-formes afin d'augmenter sa flexibilité et sa maintenabilité.

Au début des années 90, les langages de description ont intégré une approche plus Conceptuelle et formelle de la description d'une architecture, surtout pour la définition des connexions de composants. Le but de cette évolution est d'aboutir désormais à la vérification des propriétés d'une application avant son implémentation.

Vu le développement rapide des recherches dans le domaine d'architecture logicielle cette discipline 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 spécification de l'architecture logicielle selon l'approche basée sur le modèle a composants, il y a les langages de description d'architecture ou ADL.

Les langages de description d'architecture, sont des outils de spécification et de représentation formelle de l'architecture logicielle [5][6][8] qui ont pour but de décrire formellement les architectures logicielles et matérielles d'un système informatique en permettant la définition d'un vocabulaire précis et commun pour les acteurs devant travailler autour la spécification liée à l'architecture logicielle (architectes, concepteurs, développeurs etc.), ils spécifient les composants de l'architecture de manière abstraite sans entrer dans des détails d'implantation, définissent de manière explicite les interactions entre composants d'un système et fournissent un support de modélisation pour aider les concepteurs à structurer et composer les différents éléments.

En fait, les ADLs sont un support pour la description de la structure de l'application. Ils offrent des facilités de réutilisation des composants et des moyens de description de la composition par description des dépendances entre composants et des règles de Communication à respecter.

Les ADL se distinguent principalement par le domaine d'application et le contexte dans lequel ils ont été définis. Ils peuvent être classés en deux grandes familles :

- La première correspond aux langages qui privilégient la description des éléments de l'architecture et leur assemblage structurel. Ces langages sont accompagnés d'outils de modélisation, d'analyseur syntaxique et de générateur de code.

- la seconde définit les langages qui se centrent sur la description de la configuration d'une architecture et sur la dynamique du système. Cette famille de langages regroupe des langages accompagnés d'outils de modélisation et de génération de code mais aussi d'une plate-forme d'exécution ou de simulation d'un système, voire de modification dynamique pendant l'exécution. La particularité de ces langages est de définir un élément d'une architecture (composant ou connecteur) comme une instance. Il devient alors facile et simple de spécifier l'évolution dynamique d'une application au cours de son exécution.

Un certain nombre de langages de description d'architecture logicielle ont été définit pour décrire, modéliser, vérifier et implémenter les architectures logicielles, on cite par exemple *Darwin*, *Unicon* (Universal Connector Support), *Rapide*, *WRIGHT*[18] etc.

Beaucoup de ces langages supportent l'analyse et le raisonnement sophistiqué, par exemple l'ADL Wright permet à des architectes de spécifier des protocoles de communication temporels et de contrôle de propriété, quant à *Rapide* il supporte les événements basés sur la spécification comportementale et la simulation des architectures réactives.

Les ADL existant sont faiblement couplés aux langages d'implémentation, posent des problèmes d'analyse, implémentation, compréhension et évolution des systèmes logiciels. Certain ADL connecte des composants qui sont implémentés dans des langages différents, seulement ces langages ne garantissent pas que le code implémenté obéit aux contraintes architecturales, et n'imposent pas l'intégrité de communications.

Cependant l'ADL Archjava unifie l'architecture logicielle avec l'implémentation, s'assure que l'implémentation se conforme aux contraintes architecturales en obéissant principalement a une propriété d'uniformité appelée l'intégrité de communication, Archjava est une extension a java qui unifie la structure architecturale et l'implémentation, assure la tracabilité entre l'architecture et le code et supporte le co-évolution de l'architecture et l'implémentation, il garantit l'intégrité de communication entre l'architecture et son implémentation.

A la fin, nous pouvons dire que Archjava permet à des programmeurs d'exprimer facilement la structure architecturale et unifie cette structure a l'implémentation a chaque étape du cycle de vie de logiciel, ainsi aide les concepteurs à favoriser la conception efficace d'architecture, l'implémentation, la compréhension et l'évolution de programme.

Démarche de développement de l'outil de spécification

Pour créer des bons logiciels capables de satisfaire des besoins bien précis et pour le développement rapide et efficace des applications de façon cohérente et prévisible, il est nécessaire de suivre un processus de développement sain qui permet de décrire les étapes par la qu'elles le développement des applications doit passer, les ressources qu'il doit consommer et les objectifs aux quels doivent répondre les systèmes conçus.

La modélisation est le pilier de toute activité qui conduit à la conception des logiciels de qualité, elle consiste à créer une représentation simplifiée d'un problème, pour cette raison nous allons modéliser la démarche de développent de notre système avec UML.

UML est un formalisme et non une méthode [11] c'est pour ça il faut choisir une démarche de développement et la démarche suivie tout au long le développement de ce système comprend les étapes suivantes :

- Analyse ou Expression des besoins.
- Conception.
- Réalisation ou implémentation.
- Test.

CHAPITRE III : Expression des besoins

III.1-Introduction :

Toute modélisation informatique doit partir d'une expression des besoins claire et précise qui servira de référence tout au long de la réalisation du projet, et dans le cadre de ce projet l'environnement développé doit répondre aux exigences suivantes :

- Fournir un ensemble d'outils et de services pour faciliter la spécification d'architecture logicielle avec l'ADL ***Archjava***.
- Permettre la spécification d'architecture logicielle sous forme de composants interconnectés.
- La simplicité d'utilisation: permet une prise en main rapide de l'outil et de différents services offerts par celui-ci pour la spécification des architectures logicielles.
- Organisation des composants (primitifs ou complexe) en bibliothèque pour permettre la réutilisation de ces derniers pour spécifier d'autres architectures logicielles.
- Simplicité de modification d'architecture spécifiée au niveau graphique en permettant l'ajout/suppression des composants, des connecteurs, des ports… etc.
- Génération du code ArchJava spécifique a chaque élément de l'architecture spécifiée graphiquement, chaque fois que l'utilisateur créer un composant, ajoute un port a un composant, interconnecte deux ou plusieurs composants entre eux etc.
- La mise à jour instantanée du code ArchJava suite a toute opération graphique : Effacement de composant, effacement de connexion, ajout de connexion ou composant etc..
- Validation de l'architecture logicielle spécifiée en utilisant l'outil spécifique a la validation.

III.2-L'architecture générale de système :

Les composants essentiels de notre système sont :

- Le premier composant est **l'éditeur graphique**.
- Le deuxième est le **générateur de code.**
- Le troisième composant outils de **validation de l'architecture spécifiée.**

III.2.1- Présentation générale de l'éditeur graphique :

L'éditeur graphique est un outil pour la modélisation graphique de l'architecture logicielle à spécifier avec l'ADL ArchJava.
Cet outil offre au concepteur de logicielle (l'utilisateur de système en général) la possibilité de dessiner (modéliser) chacun des éléments de son architecture de manière graphique
En lui offrant les outils adéquats qui se sont :

- Une palette de dessine qui contient tous les boutons pour dessiner des composants, des ports, des connecteurs ...etc.

- Un menu qui guidera le concepteur durant le processus de spécification d'architecture ,ce menu va aider l'utilisateur a créer des composants, importer des composants déjà faites, interconnecter ces composantes selon les règles d'interconnexion des composantes sous **ArchJava**.

- Une Zone de dessin pour que l'utilisateur place les éléments de son architecture (les composants, les ports, connecteurs...etc.) dans cette zone de dessin, et réaliser une interconnexion entres ces composants).

Si les composants sont totalement définis, il faudrait que la connexion soit entre des ports connectable, si non l'outil n'acceptait pas de réaliser la connexion, tous en placent les composant et les connecteurs sur la zone de dessin, l'outil de spécification devra générer du code **ArchJava** adéquat.

Si les composants sont à créer, il faudrait les réaliser selon les règles de **ArchJava**.
Apres la représentation graphique de l'architecture à concevoir, il faudrait que l'outil soit capable de générer un code **ArchJava** qui puisse être exécutable.

Les objectifs de l'éditeur graphique sont:

- La modélisation (représentation) graphique de l'ensemble de l'architecture c à dire donner une vue générale de l'architecture qui aide le concepteur (l'architecte logiciel) à bien analyser son architecture.

- Cette interface graphique devra à la fois guider l'utilisateur (concepteur) pour la production d'architecture décrite en ArchJava et éviter un nombre important d'erreurs, l'édition des fichier '**.ArchJ**', et le contraindre à respecter les normes d'interconnexion des composants selon **ArchJava.**

• Un autre objectif qui est la création d'une interface graphique qui puisse guider l'utilisateur dans la création des fichiers **ArchJava** en lui proposant, à chaque étape de l'écriture ,les différentes actions possibles (par exemple ajout, suppression ou modification des composants).

• L'éditeur devait aussi offrir toutes les facilités d'utilisation habituelles que l'on retrouve dans la plupart des applications graphiques, par exemple copier, coller, suppression, ..etc.

III.2.2- Présentation générale de générateur de code :

Le générateur de code c'est un outil qui permet de générer du code ArchJava spécifique a chaque élément de l'architecture spécifiée, qui puisse être exécutable par le compilateur ArchJava, ce code est généré chaque fois que l'utilisateur créer un composant, ajoute un port a un composant, interconnecte deux ou plusieurs composants entre eux, c à dire lorsqu'en crée un composant en utilisant les outils offerts par le system graphique (palette de dessin etc.) le générateur de code génère le code source adéquat a chaque représentation graphique de manière systématique et synchrone avec l'ajout d'une nouvelle entité graphique tout en respectant la syntaxe de langage ArchJava.

Le code généré ne présente qu'un code partiel correspondant a la description graphique Effectuée au préalable et c'est à l'utilisateur de compléter le code restant (les données d'entrées ...etc.).

Le générateur de code a pour objectifs :

- Aider les concepteurs a générer de code exécutable son a être obliger d'éditer tous le code mais seulement une partie de code (c'est a l'utilisateur de remplire les détailles qui restent).
- Offrir une information textuelle contenant le code ArchJava partiel ou complet correspondant à la description graphique de l'architecture déjà conçue sur l'éditeur graphique.
- Offrir une grande facilité aux utilisateurs en générant lui-même la grande partie du code ArchJava.
- Validation de l'architecture en compilant et exécutant le code correspondant généré pour vérifie qu le code généré se conforme à l'architecture conçue dans un sens technique strict.

III.2.3- Outil de validation de l'architecture logicielle spécifiée :

Cet outil permis de valider l'architecture logicielle spécifiée, cette validation consiste à :

- La compilation de code **ArchJava** spécifique a l'architecture conçue au préalable et la récupération des erreurs de compilation sur le terminal de sortie de résultats de l'outil de spécification pour que l'utilisateur puisse corriger ses erreurs.

- Lancement de l'exécution de code **ArchJava** généré.

III.3- Modélisation des besoins :

Les besoins peuvent être exprimer sous forme de diagrammes des cas d'utilisation (use cases), qui vont servir à saisir le comportement attendu d'un system en cours de développement, cette modélisation comprend deux étapes :

- La détermination des acteurs de système
- La détermination des principaux cas d'utilisation (uses case) .

III.3.1-Les acteurs :

L'acteur est, par définition le rôle joué par une personne qui interagit avec le système et dans le cas de ce projet les acteurs qui vont interagir avec le système pouvons être :

- Des architectes logiciels.

- Des développeurs des systèmes informatiques.

- Des étudiants qui s'intéressent au domaine de génie logiciel et ont pour besoin de spécifier des architectures logicielles avec l'ADL **ArchJava**.
- Des enseignants, des concepteurs…etc.

- Ou toute personne s'intéresse au domaine d'architecture logicielle et a pour besoin de spécifier des architectures logicielles avec l'ADL **ArchJava**.

III.3.2-Les principaux cas d'utilisations :

Les cas d'utilisations (en anglais *use cases*) permettent de représenter le fonctionnement du système vis-à-vis de l'utilisateur.

Les principaux cas d'utilisations auxquels doit répondre mon system sont :

1) La création d'une architecture logicielle graphiquement.

2) Génération de code **ArchJava** spécifique a cette architecture.

3) Validation de l'architecture spécifiée.

Ces cas d'utilisation sont modélisés par le diagramme des cas d'utilisation comme montre la figure 3.1 .

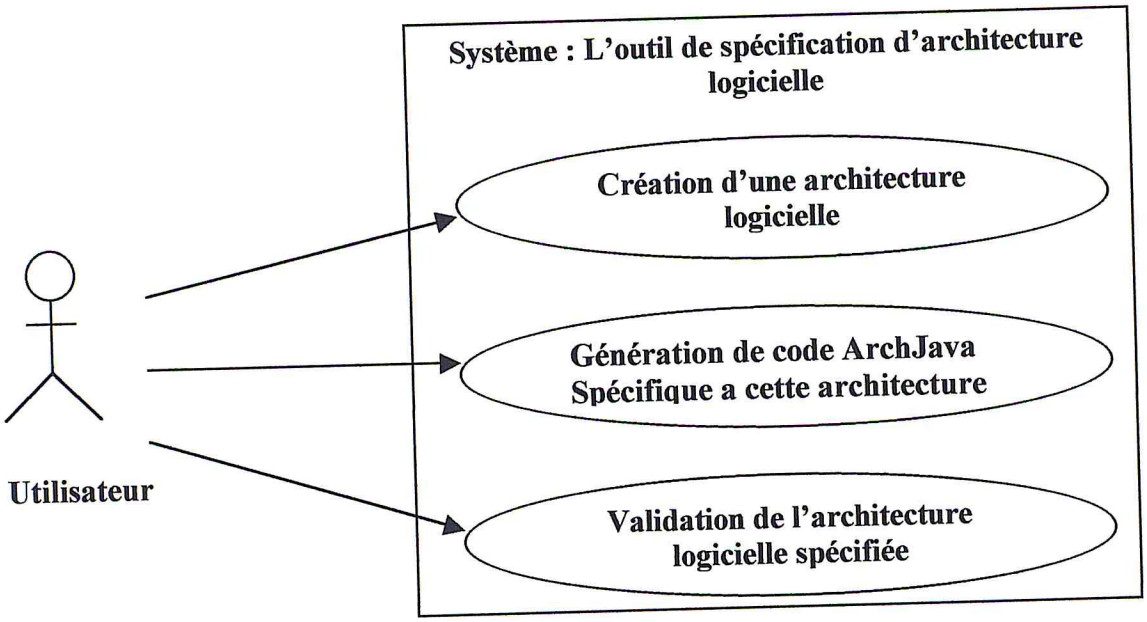

Figure 3.1:Diagramme des cas d'utilisation pour les cas d'utilisation principaux du système

Chaque cas de ces cas d'utilisation principaux sera ensuit raffiné, et représenté avec des diagrammes des sous cas d'utilisation, pour une grande clarification en décomposant chaque cas d'utilisation en sous cas d'utilisation afin de mieux comprendre le diagramme principal.

Les diagrammes des cas d'utilisation représentent les fonctions principales du système de point de vue des utilisateurs (voir Annexe B), mais ils ne modélisent pas ces fonctions de manière plus détaillée, c'est pour cette raison nous documentons ces diagrammes avec des diagrammes de séquence pour modéliser de manière temporelle les différentes interactions entre les utilisateurs du système et le système lui-même.

1-Cas d'utilisation « création d'une architecture logicielle graphiquement » :

Ce cas d'utilisation est un ensemble de sous cas d'utilisation comme montre la figure 3.2, ces sous cas d'utilisation serons détaillés cas par cas et documentés ensuit par des diagrammes de séquence.

La création d'une architecture comporte plusieurs opérations, comme créer des nouveaux composants logicielles, importer des composants déjà faits, de la bibliothèque des composants primitifs ou complexes, ensuit ajouter des ports de communication a ces composants pour qu'ils puissent communiquer entre eux, réaliser des interconnexions entre ces composants et a la fin, vérifier la justesse de ces interconnexions.

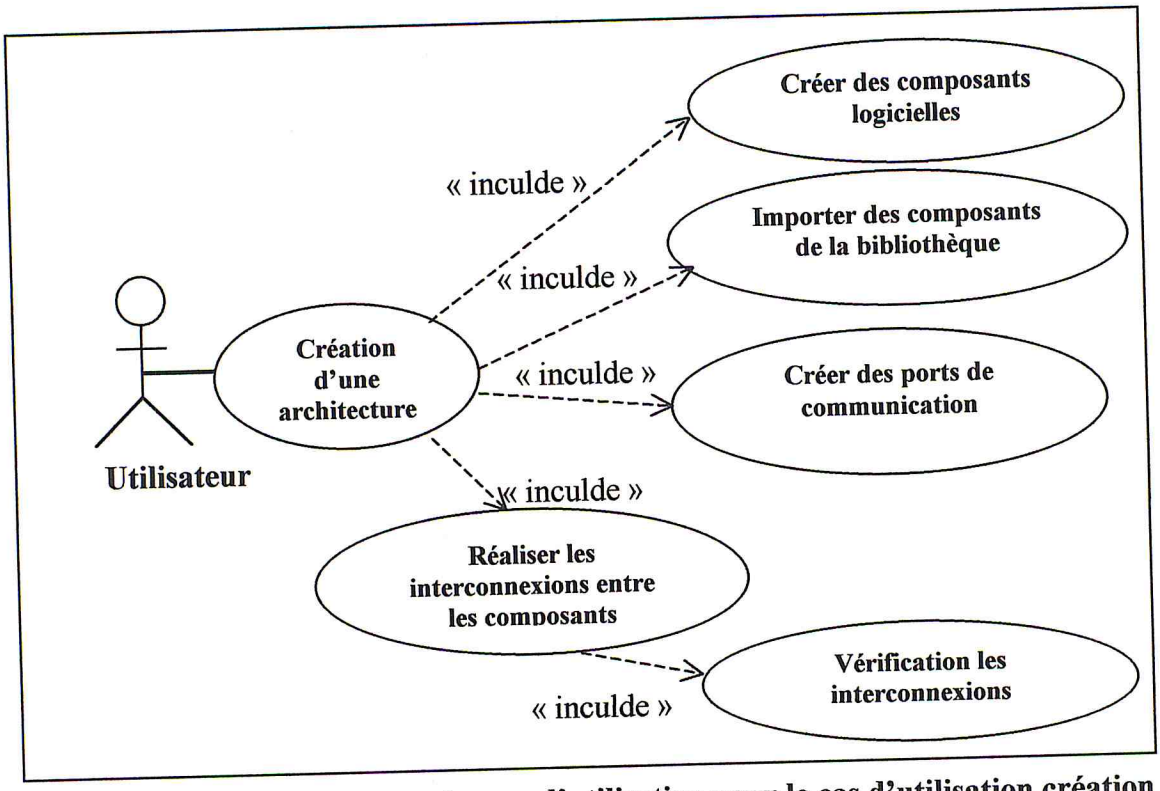

Figure 3.2:Diagramme des cas d'utilisation pour le cas d'utilisation création d'une architecture

1.1-Cas d'utilisation « Créer des composants logicielles» :

1) Scénario:

1- L'utilisateur clique sur la palette de dessin.
2- Choisit le bouton correspondant au dessin de composant.
3- Déplace la souris sur la zone dédiée au dessin.
4- Relâche la souris sur l'endroit ou il veut placer ce composant.
5- Le système dessine le composant.

2) Diagramme de séquence pour le cas d'utilisation « Créer des composants logicielles» :

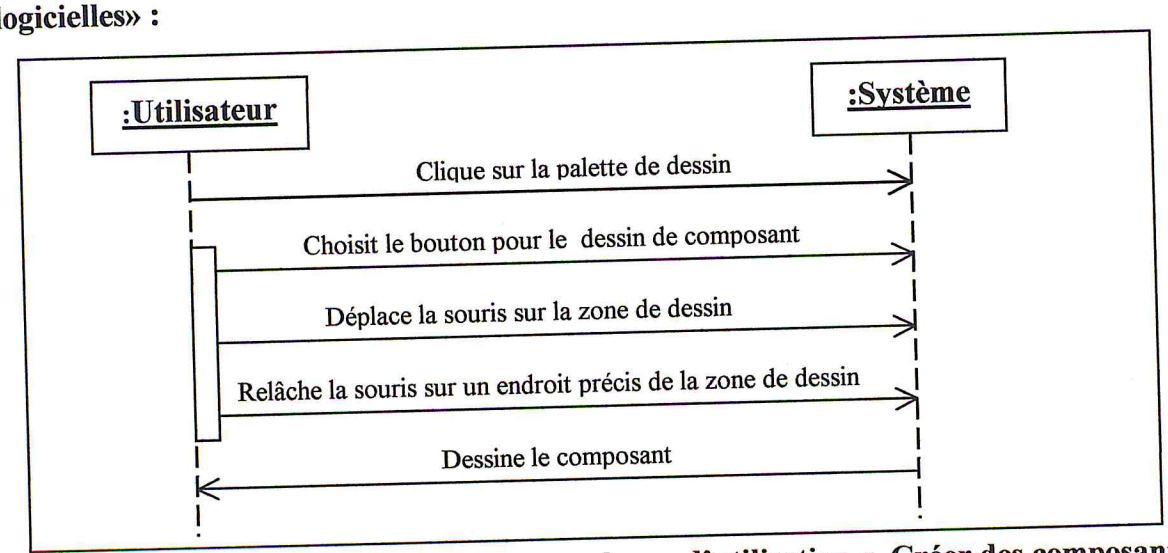

Figure 3.3: Diagramme de séquence pour le cas d'utilisation « Créer des composants logicielles»

1.2-Cas d'utilisation « Importer des composants de la bibliothèque» :

1) Scénario:

1- L'utilisateur demande d'importer un composant.
2- Choisit le composant a importé.
3- Le système importe le composant.

2) Diagramme de séquence pour le cas d'utilisation « Importer des composants de la bibliothèque» :

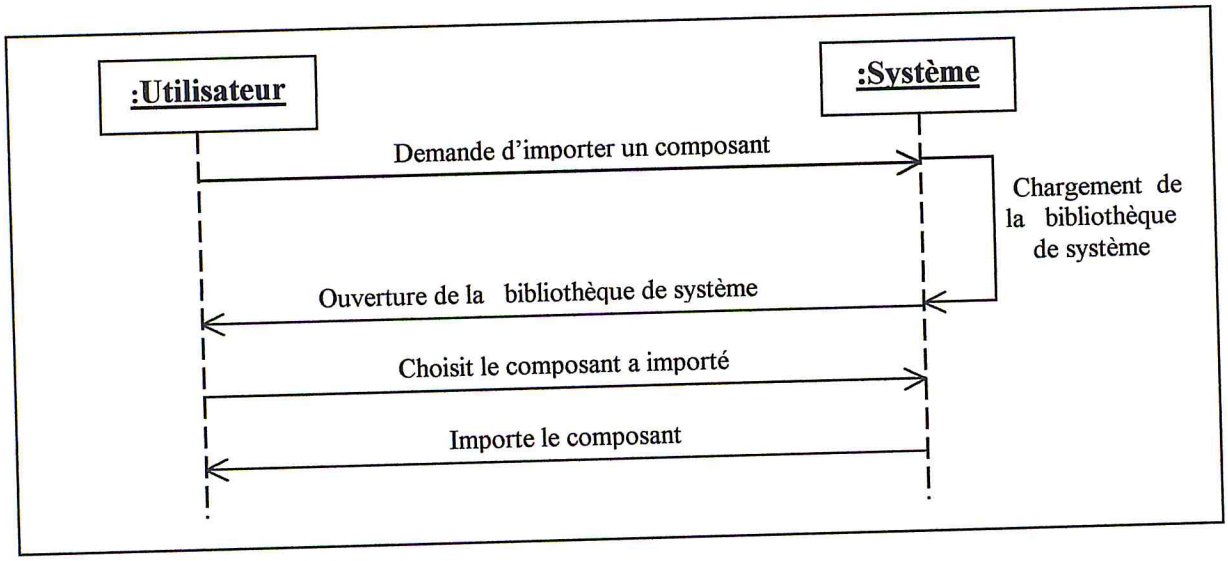

Figure 3.4: Diagramme de séquence pour le cas d'utilisation « Importer des composants de la bibliothèque»

1.3-Cas d'utilisation « Créer des ports de communication» :

1) Scénario:

1- L'utilisateur choisit le type de port a crée (IN ou OUT) .
2- Définit les propriétés de ce port.
3- L'utilisateur choisit le composant a qui doit ajouter le port.
4- Le système ajoute le port a ce composant

2) Diagramme de séquence pour le cas d'utilisation « Créer des ports de communication » :

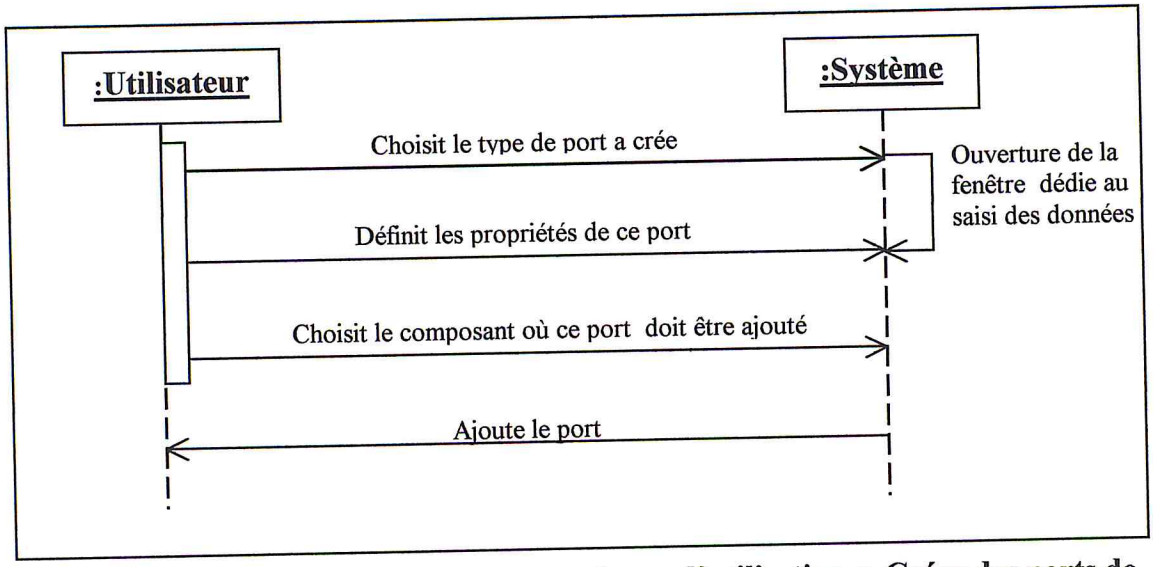

Figure 3.5: Diagramme de séquence pour le cas d'utilisation « Créer des ports de communication »

1.4-Cas d'utilisation « Réaliser les interconnexions entre les composants » :

1) Scénario:

1- L'utilisateur choisit le type de connexion (statique ou dynamique)
2- Le système dessine un connecteur.
3- L'utilisateur déplace les rôles de connecteur sur les ports à connecter.
4- Le système réalise l'interconnexion entre les ports des composants
5- Le système vérifie la justesse de l'interconnexion établit.
6- Si fausse, le système annule cette connexion.

2) **Diagramme de séquence pour le cas d'utilisation « Réaliser les interconnexions entre les composants » :**

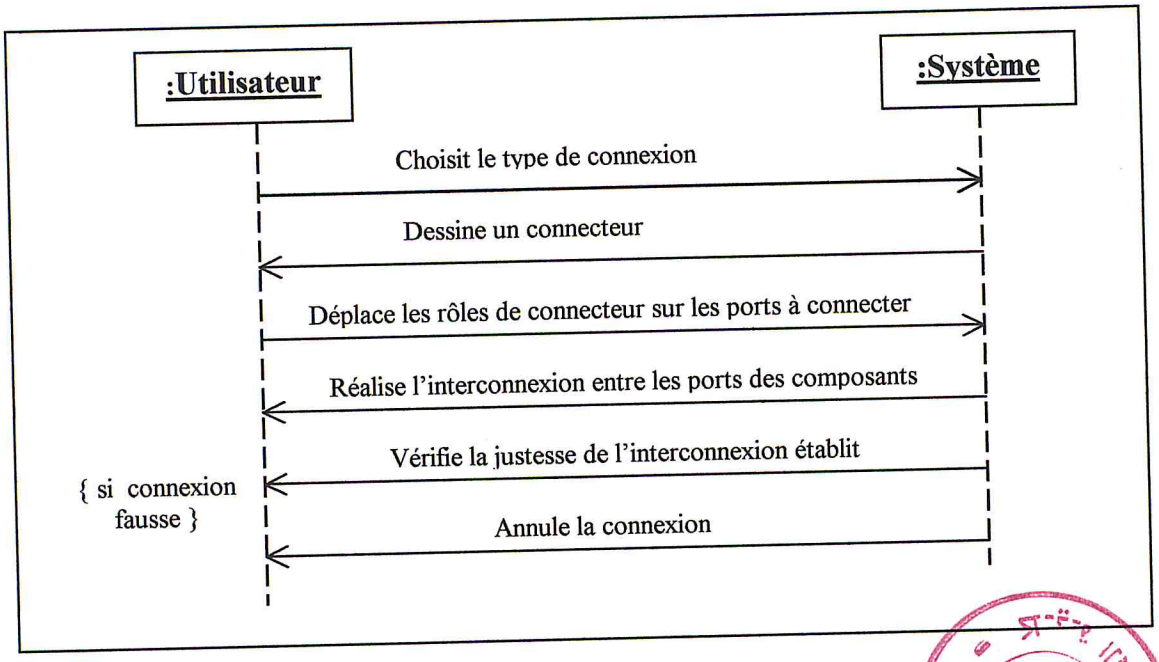

Figure 3.6: Diagramme de séquence pour le cas d'utilisation « Réaliser les interconnexions entre les composants »

2-Cas d'utilisation « génération du code ArchJava » :

La génération du code ArchJava correspondant à l'architecture crée graphiquement Au préalable par l'utilisateur se fait de manière systématique, et elle comprend deux Opérations essentielles, la génération de code correspondant à chaque composant logicielle, Et la génération du code ArchJava correspondant à chaque port de communication de cette architecture.

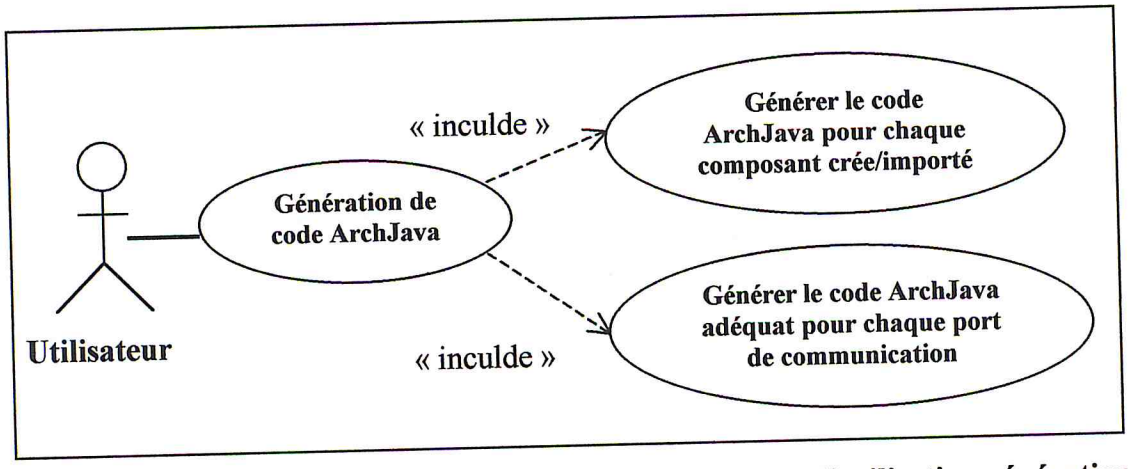

Figure 3.7:Diagramme des cas d'utilisation pour le cas d'utilisation génération de code ArchJava

2.1-Cas d'utilisation « Générer le code ArchJava pour chaque composant crée/importé » :

1) Scénario:

1- L'utilisateur dessine un composant sur la zone de dessin.
2- Le système génère le code ArchJava spécifique a ce composant.

2) Diagramme de séquence pour le cas d'utilisation « Générer le code ArchJava pour chaque composant crée/importé » :

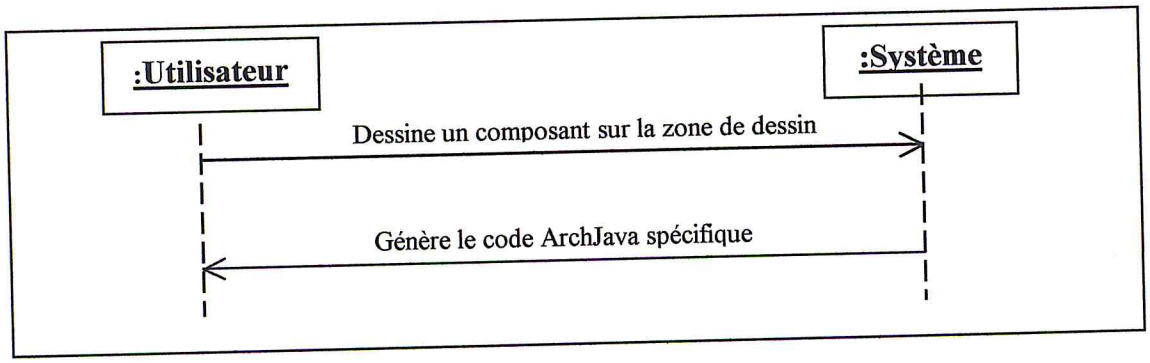

Figure 3.8: Diagramme de séquence pour le cas d'utilisation « Générer le code ArchJava pour chaque composant crée/importé »

2.2-Cas d'utilisation «Générer le code ArchJava adéquat pour chaque port de communication » :

1) Scénario:

1- L'utilisateur ajoute un port de communication a un composant.

2- Le système génère le code ArchJava spécifique a ce port

3- Le système fait la mise a jour de code spécifique a ce composant.

3) Diagramme de séquence pour le cas d'utilisation « Générer le code ArchJava adéquat pour chaque port de communication » :

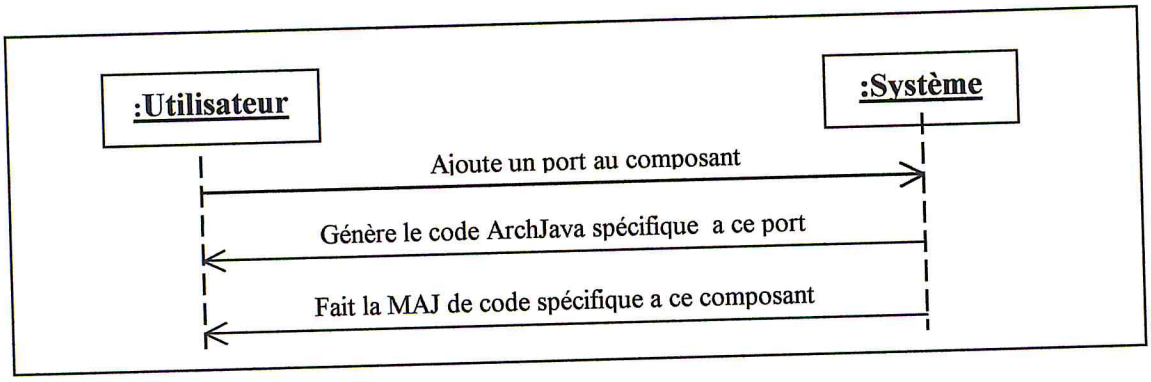

Figure 3.9: Diagramme de séquence pour le cas d'utilisation « Générer le code ArchJava pour chaque composant crée/importé »

3-Cas d'utilisation « validation d'architecture » :

L'étape de validation d'architecture est une étape essentielle dans la spécification d'architecture elle permet de vérifier que le code généré qui spécifie l'architecture se conforme aux contraintes architecturales, cette etape comprend deux étapes, la premiere est la compilation de code ArchJava généré et la deuxiéme est l'execution de ce code.

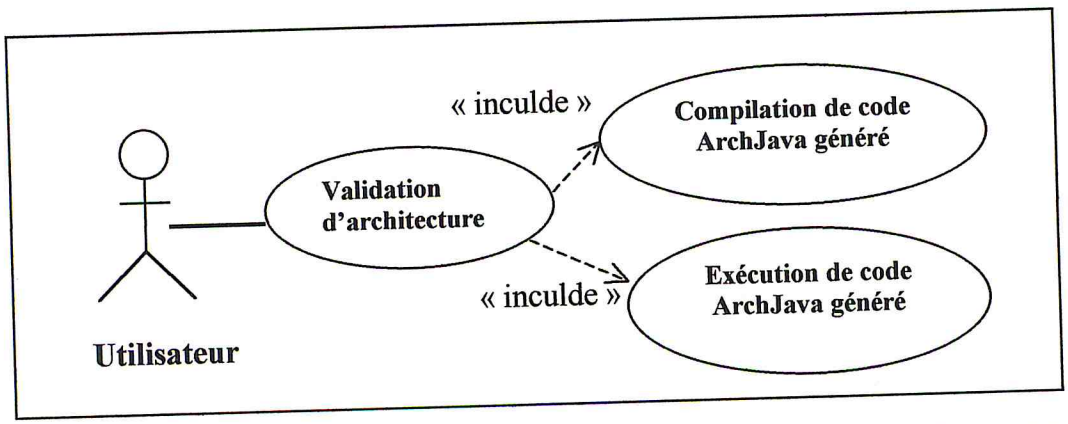

Figure 3.10:Diagramme des cas d'utilisation pour le cas d'utilisation validation d'architecture

3.1-Cas d'utilisation «Compilation de code ArchJava généré » :

1) Scénario:

1- L'utilisateur demande de compiler le code ArchJava spécifique a l'architecture générée (le fichier Main.archj) .
2- Le système lance la compilation.
3- Le système récupère les résultats de compilation sur le terminal d'affichage de système.

2) Diagramme de séquence pour le cas d'utilisation « Compilation de code ArchJava généré» :

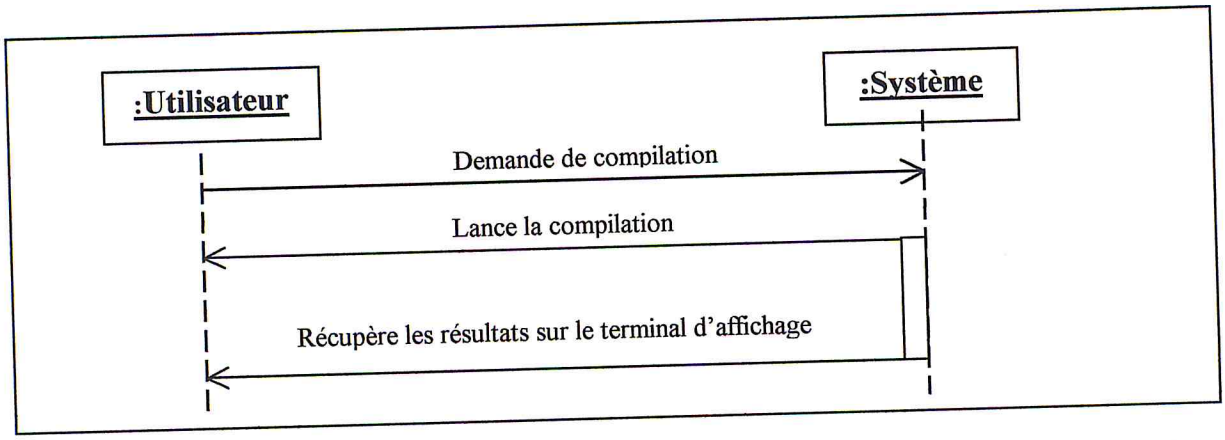

Figure 3.11: Diagramme de séquence pour le cas d'utilisation « Compilation de code ArchJava généré»

3.2-Cas d'utilisation «Exécution de code ArchJava généré » :

1) Scénario:

1- L'utilisateur demande d'exécuter le code ArchJava spécifique a l'architecture générée (le fichier Main.archj) .
2- Le système lance l'exécution.
3- Le système affiche les résultats sur le terminal d'affichage de système.

2) Diagramme de séquence pour le cas d'utilisation « Exécution de code ArchJava généré» :

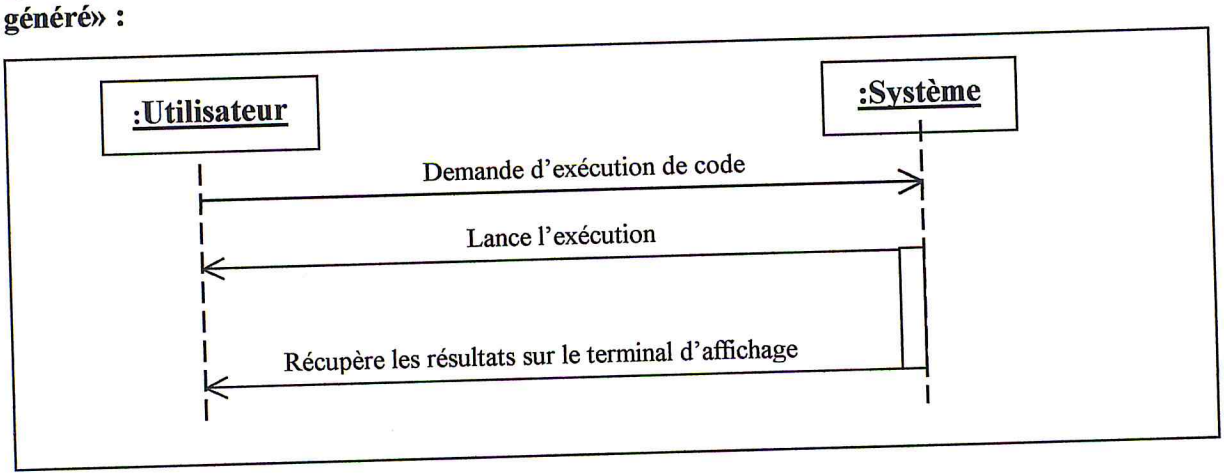

Figure 3.12: Diagramme de séquence pour le cas d'utilisation « Exécution de code ArchJava généré»

CHAPITRE IV : La Conception

IV.1-Introduction :

La conception a pour but de définir de façon très précise les fonctions et l'architecture du logiciel, a partir des besoins exprimés et des contraintes générales définies en phase d'expression des besoins [10].

Elle débute par la détermination de l'architecture globale du système développé, c'est à dire l'élaboration de ces structures statiques et dynamiques.

Cette partie sera organisée comme suit :

Nous commençons d'abord par la description de l'architecture globale de l'environnement développé, puis détailler cette architecture en terme de modules, ensuit pour chaque module nous spécifions ces structures statiques en se basant sur des diagrammes de classe, spécifier ces structures dynamiques en se basant sur des diagrammes de séquences et des diagrammes d'activités.

IV.2-L'architecture générale de système :

L'architecture logicielle d'un système est, l'ensemble des décisions fondamentales d'organisation de ce système, de choix des éléments structurels qui composent ce système, Et de comportement de ces éléments structurels, et dans le cadre de notre projet l'architecture de ce système peut être décrite comme suit (voir figure 4.1 si dessous):

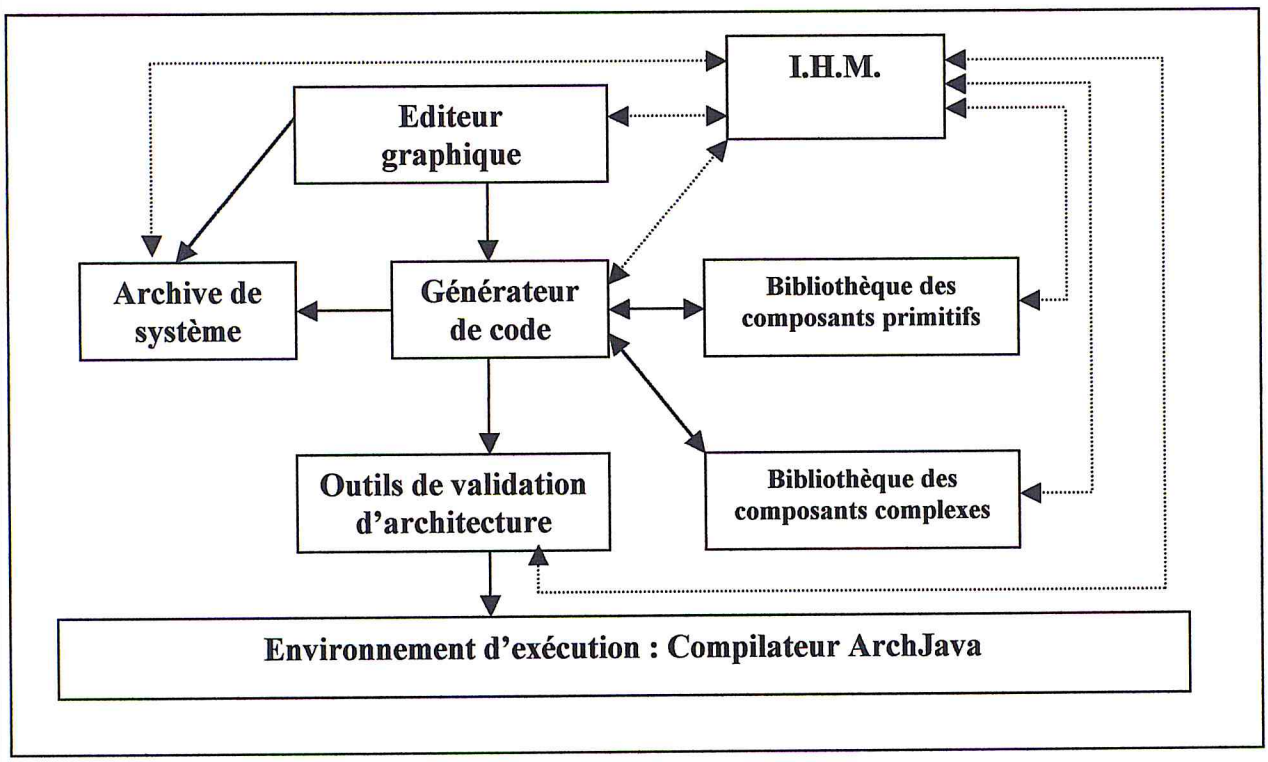

Figure 4.1: L'architecture globale de système

1) Module1 : L'éditeur Graphique.
2) Module2 : Le générateur de code Archjava.
3) Module3 : pour la validation d'architecture logicielle.
4) L'IHM : l'interface utilisateur.
5) Bibliothèque des composants primitifs.
6) Bibliothèque des composants complexes ou composites.
7) Archive de système.

IV.2.1-Module1 : L'éditeur graphique :

L'éditeur graphique est le composant primordial, essentiel dans l'architecture de ce système, car il constitue le cœur de toute opération de spécification d'architecture logicielle, il offre à l'utilisateur de système la possibilité de spécifier chacun des éléments de son architecture de manière graphique en lui offrant les outils suivants :

- Une Palette de dessin qui contient un ensemble de boutons pour dessiner des composants logiciels, des ports, des connecteurs.

- Une zone de dessin pour que l'utilisateur place les éléments de son architecture (Composants, ports, connecteurs... etc. dans cette zone de dessin et réaliser des interconnexions entre ces éléments.

Ce module offre plusieurs fonctionnalités qui facilitent la tâche a un utilisateur de système, parmi ces fonctionnalités nous citons :

- La possibilité de déplacer les composants crées tout au long la zone de dessin.

- La possibilité de supprimer des composants et des ports de la zone de dessin.

- La possibilité d'importer des composants de la bibliothèque des composants primitifs ou complexes dans le but de les réutiliser pour spécifier d'autre architecture logicielle.

- La possibilité de réaliser des interconnexions entre les composants logicielles graphiquement et vérifier la justesse des ces interconnexions avant de générer le code **Archjava** correspondant et suppression de ces interconnexions en cas d'erreurs de connexions (par exemple relier deux ***ports-in*** entre eux) et la correction automatique des connexions fausses en les supprimant de la zone de dessin.

IV.2.1.1-Objectifs de ce module :

Parmi les objectifs de l'éditeur graphique nous pouvons citer :

- La modélisation graphique de l'ensemble de l'architecture logicielle, ce qui donne une vue générale de l'architecture a fin d'aider le concepteur (l'architecte logiciel) a mieux analysé son architecture.
- Offrire aux utilisateurs de système les différents outils et services pour spécifier une architecture logicielle avec L'ADL **Archjava.**
- Permettre d'éviter un nombre important d'erreurs de génération de code **Archjava**.

IV.2.1.2- Modélisation de module éditeur graphique:

IV.2.1.2.1-Le comportement statique :

Pour visionner les parties statiques de ce module nous avons utilisé l'un des diagrammes structurels d'UML qui servent à visualiser, spécifier et documenter ces aspects statiques qui est le diagramme de classe.

Les diagrammes de classes représentent un ensemble de classes, d'interfaces ainsi que leur relation [11], ce sont des diagrammes que l'on utilise pour illustrer la vue conception statique d'un système.

N.B : Je modélise les modules par des paquetages.

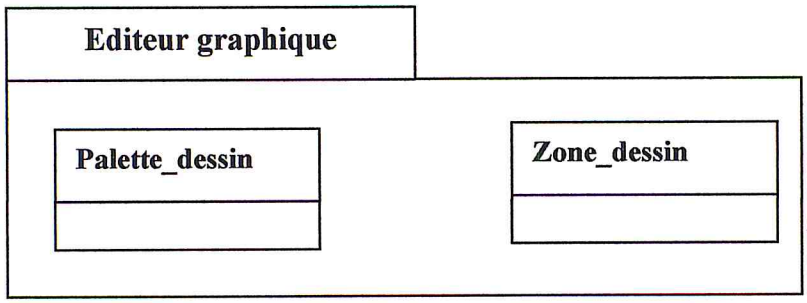

Figure 4.2: Les éléments de module éditeur graphique

IV.2.1.2.2-Le comportement dynamique :

Pour modéliser les aspects dynamiques de ce module nous avons utilisé un des diagrammes comportementaux d'UML qui est un diagramme de séquence.

Le diagramme de séquence est un diagramme d'interaction qui met en évidence l'ordre chronologique des messages [11], il représente un ensemble d'objet ainsi que les messages envoyés et reçus par ces objets, c'est un diagramme que l'on utilise pour illustrer la vue conception dynamique d'un système.

Les principales opérations réalisées par ce module sont :

- **Création des composants logiciels :**

La création des composants graphiquement se fait de la manière suivante :

1- L'utilisateur clique sur la palette de dessin.
2- Choisit le bouton correspondant au dessin de composant.
3- Déplace la souris sur la zone de dessin.
4- Relâche la souris sur l'endroit ou il veut placer ce composant.
5- Le système dessine le composant.

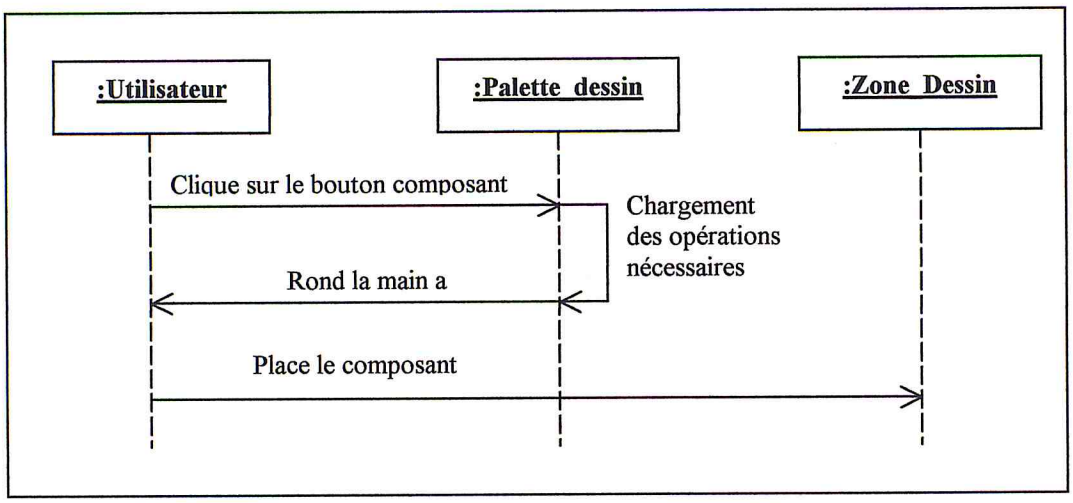

Figure 4.3: Diagramme de séquence pour l'opération « création d'un composant »

- **Ajout des ports aux composants :**

L'ajout des ports au composants est une opération très importante, car elle permet d'ajouter des interfaces de communication aux composants crées pour les permettre de communiquer avec d'autres composants logicielle, elle se fait de la manière suivante :

1- L'utilisateur choisit le type de port a crée (IN ou OUT) .
2- Définit les propriétés de ce port, c'est à dire toutes les informations sur la méthode requis ou fournie par ce port.
3- L'utilisateur choisit le composant a qui doit ajouter le port.
4- Le système ajoute le port a ce composant

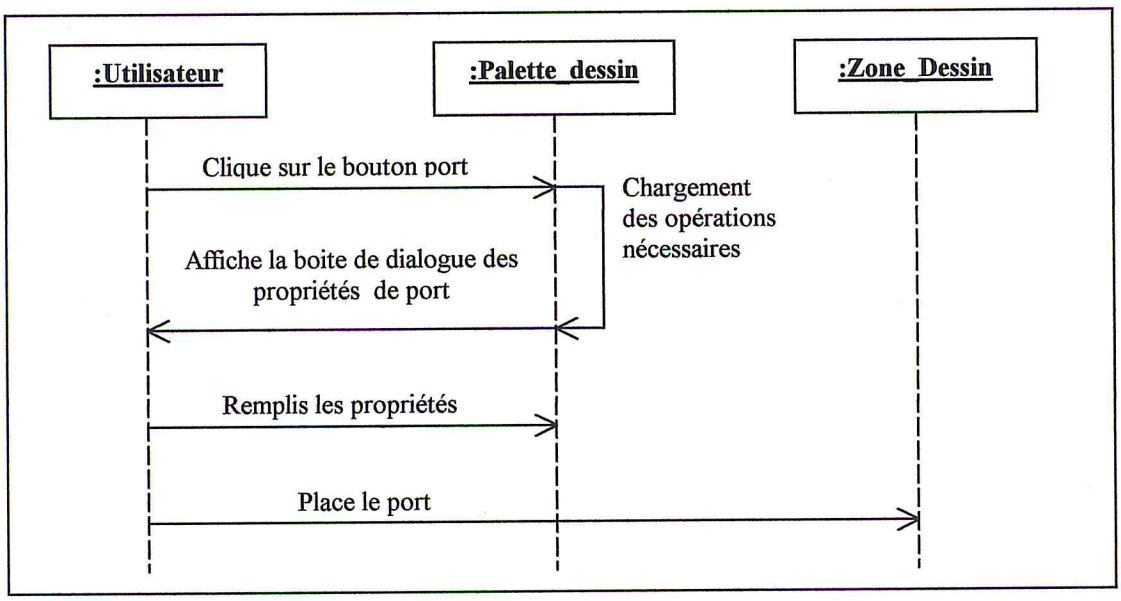

Figure 4.4: Diagramme de séquence pour l'opération « ajout d'un port »

- **Interconnexion des composants :**

L'interconnexion des composants permet de relié deux ou plusieurs composants via leurs ports de communications pour faire véhiculer des informations d'un certain type de donnée, elle est réaliser de la manière suivante :

1- L'utilisateur choisit le type de connexion (statique ou dynamique)
2- Le système dessine un connecteur.
3- L'utilisateur déplace les rôles de connecteur sur les ports à connecter.
4- Le système réalise l'interconnexion entre les ports des composants.
5- Le système vérifie la justesse de l'interconnexion établit.
6- Si fausse, le système annule cette connexion.

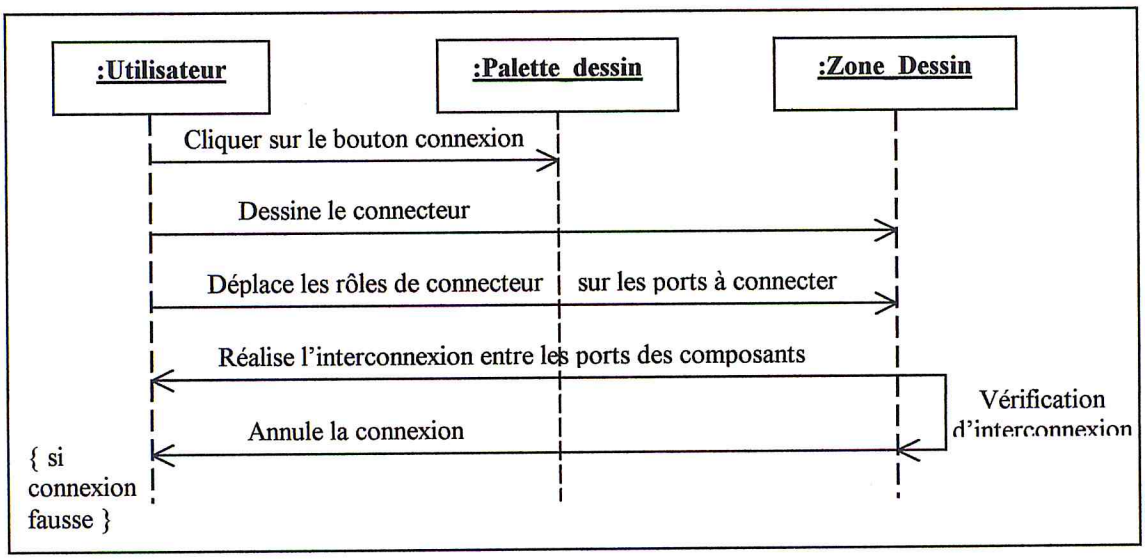

Figure 4.5: Diagramme de séquence pour l'opération « ajout d'un port »

- **Suppression des composants/ports :**

La suppression est un des nombreuses fonctionnalités offertes par l'éditeur graphique, elle s'effectue de la manière suivante :

1- L'utilisateur select le composant/port à supprimer sur la zone de dessin.
2- Clique sur le bouton suppression.
3- Le système supprime l'élément sélectionné de la zone de dessin.

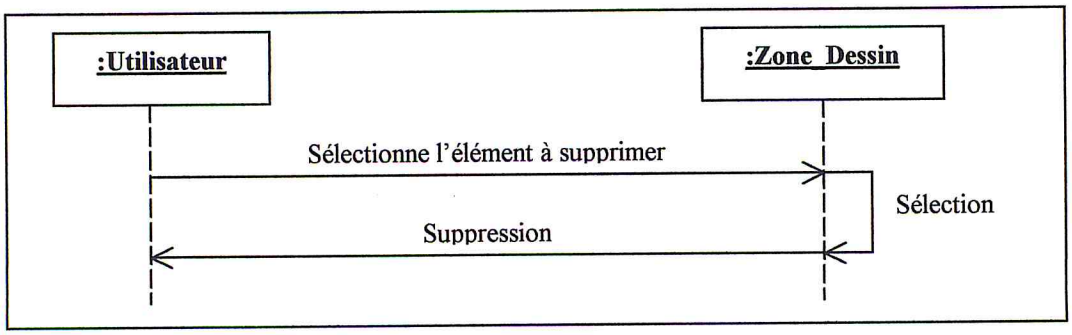

Figure 4.6: Diagramme de séquence pour l'opération « suppression des composants/ports »

IV.2.2- Module2 : Le générateur de code :

Le générateur de code est le deuxième élément essentiel dans l'architecture de ce système, car il constitue l'unité responsable de la génération de code **Archjava** qui doit à la fois correspondre à la description graphique de l'architecture logicielle déjà crée sur la zone de dessin de l'éditeur graphique, et générer de code correct qui soit conforme aux contraintes de description d'architecture logiciel avec l'ADL **Archjava.**

Ce module génère le code **Archjava** de manière systématique c'est à dire, chaque fois que l'utilisateur ajoute un composant, un port ou établit une connexion ça implique la génération automatique de code correspondant qui est un code partiel et c'est à l'utilisateur de compléter les détails qui restent avec des instructions de langage Java.

Ce module offre plusieurs fonctionnalités qui facilitent la tâche a un utilisateur de système, parmi ces fonctionnalités nous citons :

- Permettre de générer de code **Archjava** spécifique a chaque élément de l'architecture logicielle (composant, port, connecteur... etc.) qui va être exécuter ensuite par le compilateur **Archjava** de manière systématique et synchrone avec l'ajout/suppression d'une entité graphique, en respectant à chaque fois la syntaxe de l'ADL **Archjava.**

IV.2.2.1-Objectifs de ce module :

Parmi les objectifs de générateur de code nous pouvons citer:

- Aider les architectes logiciels à générer de code exécutable sans éditer eux même tout le code mais seulement une partie de code **Archjava**.

- Offrir une information textuelle contenant le code ArchJava partiel ou complet correspondant à la description graphique de l'architecture déjà conçue sur l'éditeur graphique.
- Offrir une grande facilité aux utilisateurs en générant lui-même la grande partie du code ArchJava.

IV.2.2.2- Modélisation de module éditeur graphique:

IV.2.2.2.1-Le comportement statique :

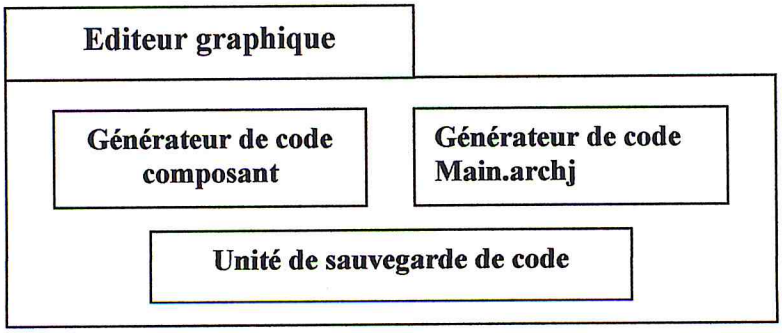

Figure 4.7: Les éléments de module générateur de code

IV.2.2.2.2-Le comportement dynamique :

Les principales opérations réalisées par ce module sont :

- **Génération de code pour chaque composant :**

La génération de code pour chaque composant est une opération qui s'effectue automatiquement après l'ajout d'un nouveau composant a la zone de dessin.

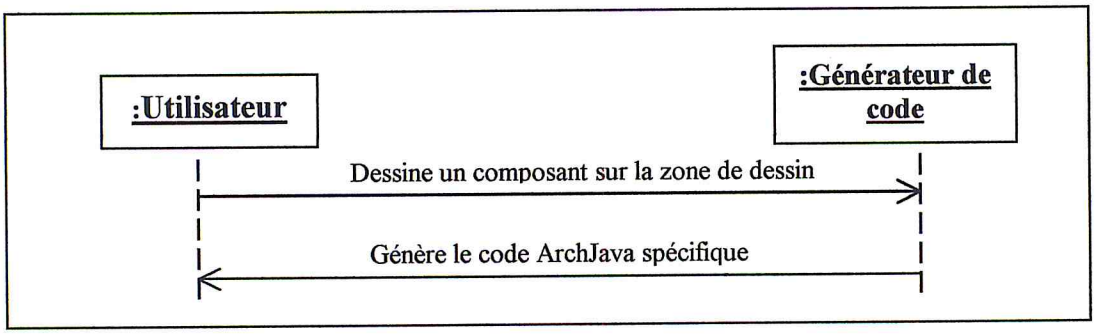

Figure 4.8: Diagramme de séquence pour l'opération « Générer de code composant »

- **génération de code pour chaque port de communication :**

Lorsque l'utilisateur ajoute un port de communication a un composant graphiquement, le générateur de code génère le code correspondant à sa description en langage Archjava, en faisant la mise a jour de code de composant où le port a étais ajouté.

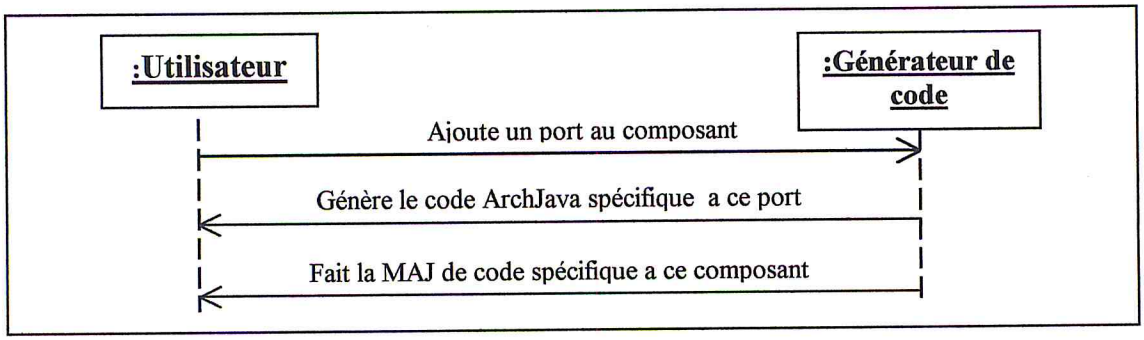

Figure 4.9: Diagramme de séquence pour l'opération « Générer le code pour chaque port de communication »

- **Suppression des composants/ports :**

Lorsque l'utilisateur supprime un composant/port de la zone de dessin, le générateur de code fait la mise a jour de code Archjava généré en supprimant la partie de code correspondant spécification de ce composant/port supprimé.

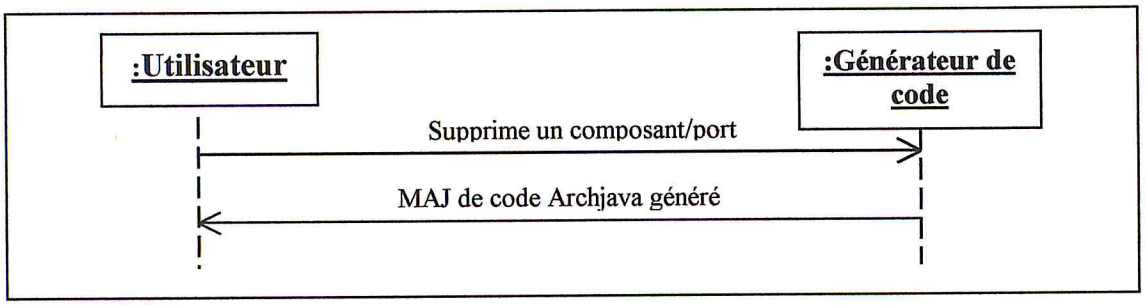

Figure 4.10: Diagramme de séquence pour l'opération Suppression des composants/ports »

- **Interconnexion des composants :**

Lorsque l'utilisateur fait une interconnexion entre deux composants sur la zone de dessin, le générateur de code génère le code Archjava pour l'interconnexion établie, et fait ensuite la mise a jour de code généré.

L'interconnexion c'est la liaison établie en reliant le port du composant au rôle du connecteur.

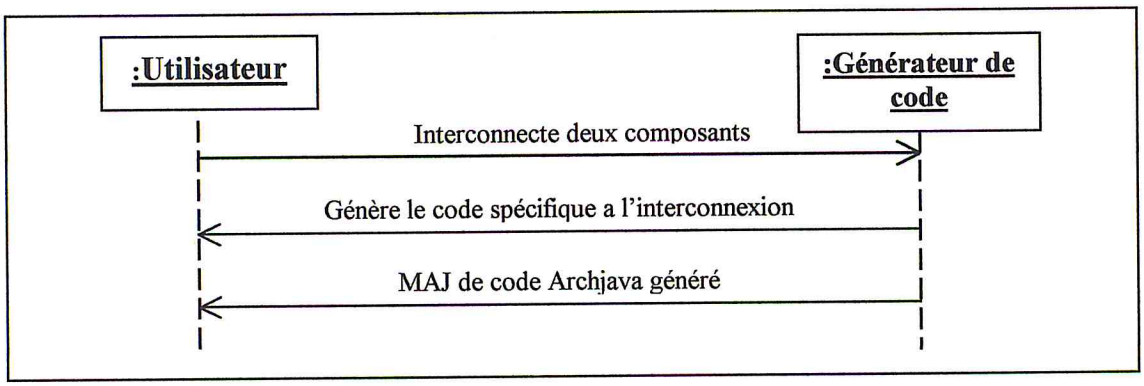

Figure 4.11: Diagramme de séquence pour l'opération Interconnexion des composants

1) Type de documents générés par ce générateur de code :

Ce générateur de code génère deux types de documents avec l'extension '**.archj**':

- **Main.archj :** Est en effet, le composant englobant de l'ensemble de l'architecture logicielle spécifié en langage Archjava, car pour compiler le code de l'architecture il faut un composant qui va l'englober qui est le composant *Main.archj*.
  Main.archj est le composant principal, celui qui représente l'entièreté de l'application.

- **ComposantX.archj :** Le générateur de code génère pour chaque composant crée le code Archjava spécifique a ce composant et le stocke dans le fichier *ComposantX.archj* tel que *ComposantX* est le nom de ce composant.

Exemple d'application:
Nous voulons réaliser l'architecture si dessous :

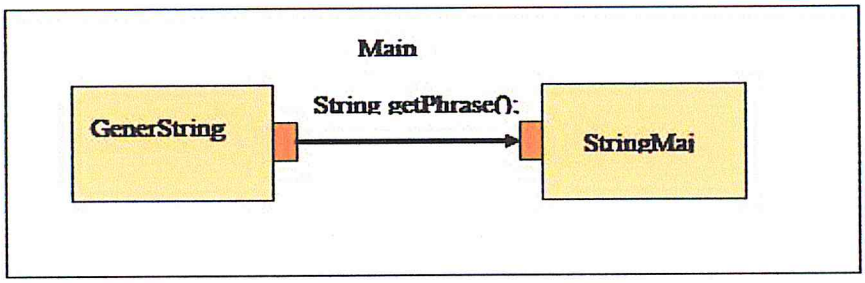

Figure 4.12: Exemple D'architecture

Cette architecture a deux composants :

- Le composant **GenerString** : Qui va générer une chaîne de caractères.
- Le composant **StringMaj** : Qui va transformer une chaîne de caractères en majuscule.

```
// Le fichier : GenerString.archj

public component class GenerString{
        public String phrase="hello world";
        public port out{
                provides String getPhrase(){return phrase;}
        }
}
```

```
//Le fichier : StringMaj.archj :

public component class StringMaj{
        public port in{
                requires String getPhrase();
        }
        public void run(){
                String s=in.getPhrase();
                s=s.toUpperCase();
                System.out.println("Maj: "+s);
        }
}
```

- Après la compilation de ces deux composants nous obtiendrons l'architecture suivante :

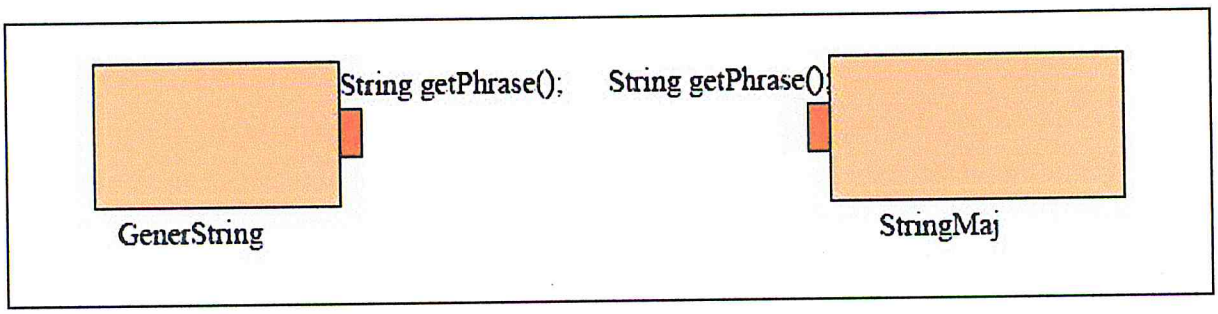

Figure 4.13: Les deux composants de l'architecture

- Pour interconnecter ces deux composants nous avons besoin d'un autre composant qui va les englober et les interconnecter. Nous allons appeler ce composant *Main.archj* qui est le composant principal, celui qui représente l'entièreté de l'application.

```
// Le fichier : Main.archj

public component class Main{
        private final GenerString gs=new GenerString();
        private final StringMaj smaj=new StringMaj();
        connect gs.out, smaj.in;
        public void run(){ smaj.run();
        }
        public static void main(String[] arg){
                new Main().run();
        }
}
```

2) Stockage/Sauvegarde des documents :

L'opération de sauvegarde est une opération très importante dans le cas de notre environnement de développement, elle permet de stocker les différents codes générés par le générateur de code dans des fichier avec l'extension **'.archj'** dans les répertoires de système, qui sont des répertoires dédier aux sauvegardes des codes composants générés, cette opération s'effectue comme suit :

1- L'utilisateur demande de sauvegarder un code composant ou le composant *Main.*
2- Le système ouvre la boite de dialogue de sauvegarde.
3- L'utilisateur saisi le nom et l'endroit de sauvegarde.
4- Le système sauvegarde le document.

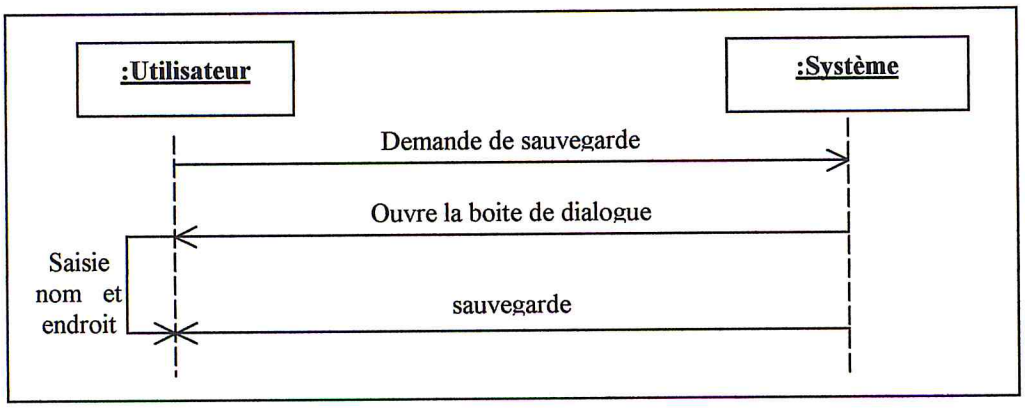

Figure 4.14: Diagramme de séquence pour l'opération de sauvegarde

IV.2.3 -Module3 : outils de validation d'architecture logicielle:

Ce module offre un ensemble d'outils qui permis de valider l'architecture logicielle spécifiée, cette validation consiste à :

- Compilation de code archjava spécifique a l'architecture conçue et la récupération des erreurs de compilation sur le terminal de sortie de résultats.
- Lancement de l'exécution de code archjava généré.

Parmi les principaux outils offerts par ce module on cite :

- **L'outil Compiler:** c'est l'outil responsable de compilation des différents codes générés par le module générateur de code.
- **L'outil Executer:** c'est l'outil responsable de l'exécution des codes **Archjava** générés par le module générateur de code.
- **Un console d'affichage:** C'est un terminal dédié a l'affichage des différents résultas de compilation ou d'exécution.

Ce module offre plusieurs fonctionnalités qui facilitent la tâche a un utilisateur de système, parmi ces fonctionnalités on cite :

- La possibilité de compiler le code de chaque composant crée tout au long l'opération de spécification de l'architecture logicielle et récupérer les résultas de compilation de ce dernier sur le terminal d'affichage.
- La possibilité de compiler le code de composant *Main.archj* qui est le composant qui englobe toute l'architecture logicielle spécifiée et récupérer les résultas de compilation de ce dernier sur le terminal d'affichage.
- La possibilité d'exécuter le code de composant *Main.archj* qui est le composant qui englobe toute l'architecture logicielle spécifiée et récupérer les résultas de l'exécution sur le terminal d'affichage.

IV.2.3.1-Objectifs de ce module :

L'objectif essentiel de l'ensemble d'outils offerts par ce module est de :

- Vérifier la conformité de code généré avec les contraintes de description d'architecture logicielle avec l'ADL **Archjava**, qui permet de décrire une architecture d'application de façon simple, et intégrée à java.
- Permet de valider le comportement de l'architecture décrite sous forme de composants en les simulant grâce à l'environnement d'exécution **Archjava** intégré.

IV.2.3.2- Modélisation de module éditeur graphique:

IV.2.3.2.1-Le comportement statique :

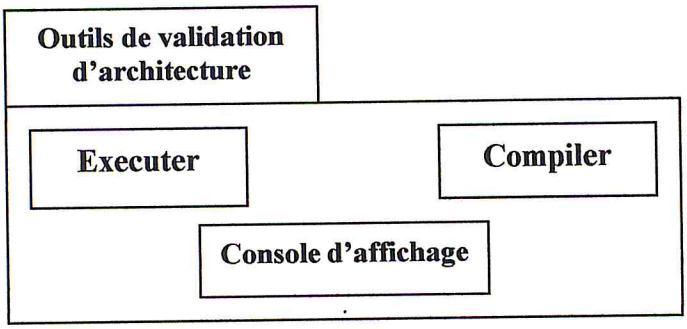

Figure 4.15: Les éléments de module générateur de code

N.B : Le composant **Main.archj** est le composant principal qui englobe toute l'architecture logicielle spécifiée, il contient des instances de chaque composant logiciel de cette architecture et les différentes interconnexions établies entres ces composants.

IV.2.3.2.2-Le comportement dynamique :

Les principales opérations réalisées par ce module sont :

- **Compilation de code Archjava pour chaque composant**
  1- L'utilisateur clique sur le nœud représentant le code de composant qui veut compiler.
  2- Le système lance la compilation.
  3- Le système récupère les résultats de compilation sur le terminal d'affichage de système.

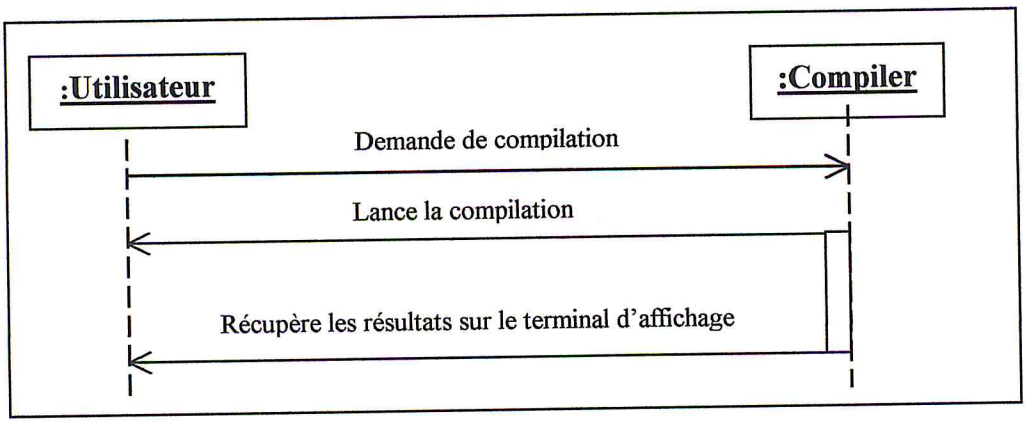

Figure 4.16: Diagramme de séquence pour l'opération « Compilation de code ArchJava généré»

- **Compilation de code Main.archj**

1- L'utilisateur demande de compiler le code ArchJava spécifique a l'architecture générée (le fichier Main.archj) .
2- Le système lance la compilation.
3- Le système récupère les résultats de compilation sur le terminal d'affichage de système.

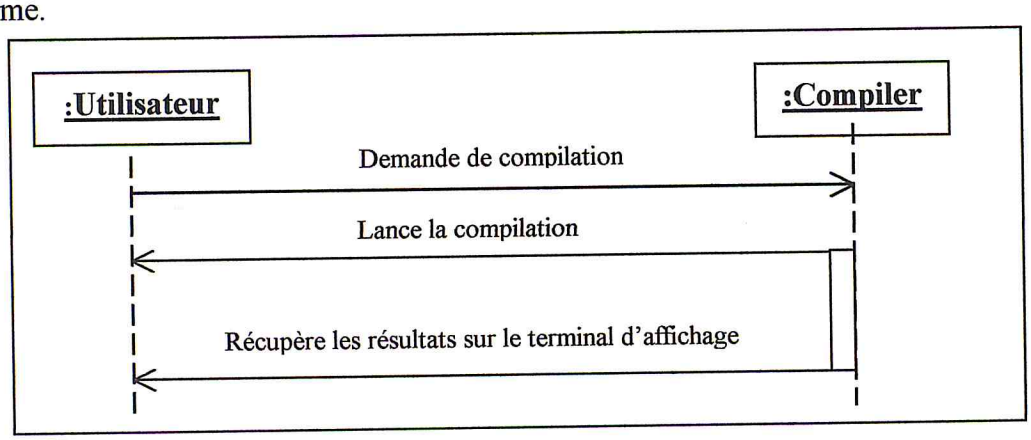

Figure 4.17: Diagramme de séquence pour l'opération « Compilation de code Main.archj »

- **Exécution de code Main.archj**

1- L'utilisateur demande d'exécuter le code ArchJava spécifique a l'architecture générée (le fichier Main.archj) .

2- Le système lance l'exécution.

3- Le système affiche les résultats sur le terminal d'affichage de système.

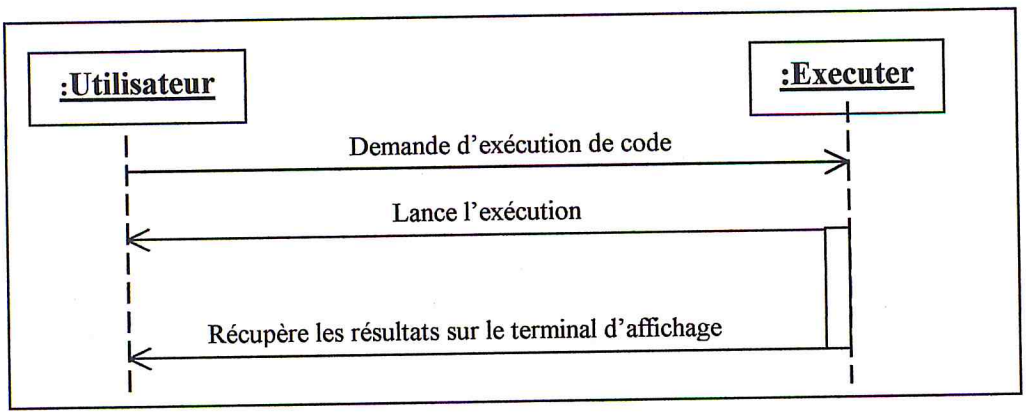

Figure 4.18: Diagramme de séquence pour l'opération « Exécution de code ArchJava généré»

IV.2.4- Bibliothèque des composants primitifs :

C'est une librairie où les composants créent serons stocker pour qu'un utilisateur de système puisse les réutiliser pour spécifier d'autres architectures logicielles en les important Tout simplement de cette bibliothèque.

Cette bibliothèque contient des composants primitifs c'est à dire des composants élémentaires non décomposables en d'autres composants.

IV.2.5- Bibliothèque des composants complexes :

C'est une librairie où les composites créent serons stocker pour qu'un utilisateur de système puisse les réutiliser pour spécifier d'autres architectures logicielles. Elle va contenir des composants complexes ou composite qui sont des composants composés d'un ou de plusieurs composants primitifs.

IV.2.6- Archive de système:

C'est un répertoire ou les code composants serons sauvegarder ainsi que d'autres informations concernant les composants.

IV.2.7- L'interface utilisateur ou l'IHM:

L'interface utilisateur est la partie visible a l'écran de cette application, en effet l'interface est le premier contact qu'a un utilisateur avec l'application, elle constitue la partie perceptible de logiciel et elle est composée de:

- **L'arbre dynamique :**

c'est une zone qui décrie l'arborescence des composants que l'utilisateur a crée durant le processus de spécification d'architecture logicielle, elle sert d'une part à donner un aperçu de tous les composants de l'architecture spécifiée et d'autre part chaque nœud de cette arborescence sert à une interface entre l'utilisateur et le système pour lui offrire des fonctionnalités telles que :

- L'ajout d'un composant a la bibliothèque.
- Sauvegarde de code correspondant a ce composant.
- La compilation de code **Archjava** décrivant ce composant

- **Architecture view :**

C'est une zone qui présente la liste exhaustive des instances des composants participants a la spécification de l'architecture logicielle, chaque nœud de cet arbre est une instance d'un composant qui fait partie de l'architecture spécifiée.

- **SmallWin :**

Est un ensemble de fenêtres qui contient des zones de texte éditable dédient à l'affichage de code source pour chaque composant logiciel crée et dont l'utilisateur a la possibilité d'ajouter un code supplémentaire (des instructions java) pour compléter ce code.

- **System Consol :**

Est une zone dédiée a l'affichage des résultas de compilation/exécution des codes générés.

- **Menu :**

C'est une barre de menus qui contient des raccourcis pour réaliser quelques opérations tel que suppression, zoom in, zoom out, sélection, enregistrer sous …etc.

- **Archj_Editor** :

C'est une zone de texte qui sert à visualiser le code **Archjava** correspondant à la description de l'architecture logicielle spécifié.

IV.3 - Diagramme de classe de toute l'architecture de l'environnement développé :

Pour illustrer la vue conception statique de toute l'architecture de l'environnement développé nous avons utilisé un diagramme de classes qui va donner une vue générale de la conception statique de notre système.

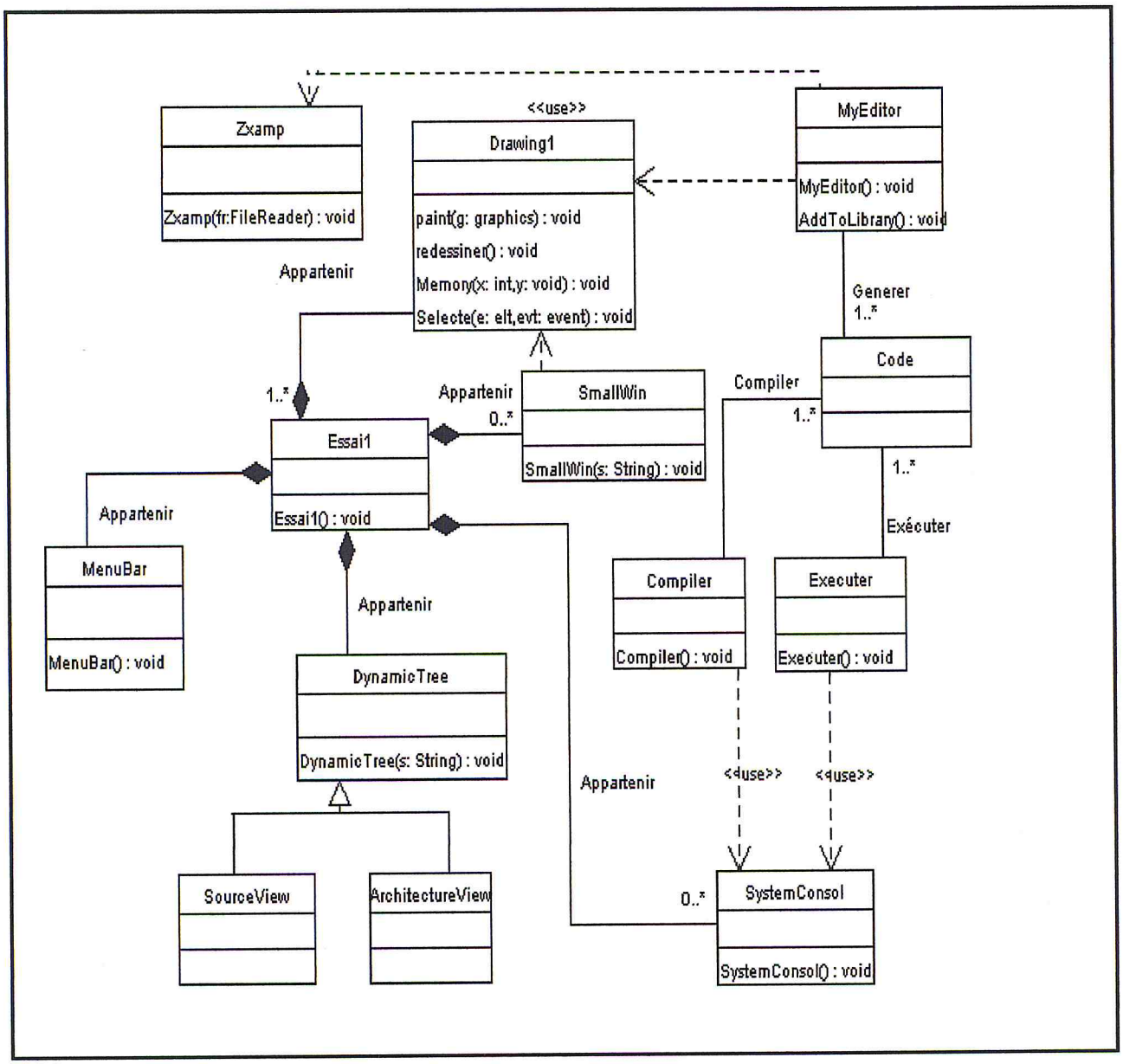

Figure 4.19: Diagramme de classe générale (vue générale sans attributs)

IV.4- Diagrammes d'activités générale :

Les diagrammes d'activité représentent le flot d'activité à l'intérieur d'un système, ils sont importants dans la modélisation des fonctions de système [11], c'est pour cette raison, nous avons vu qu'ils sont nécessaires pour modéliser les différents flots d'activités au sein de notre l'environnement développé. Caque activité est placée dans une travée (*swimlanes* en anglais)
Afin de diviser les activités en groupe sur le diagramme d'activité, chaque groupe représente le département responsable de ces activités.

Les principales activités que le système réalise sont :

- **Création d'un composant logiciel :**

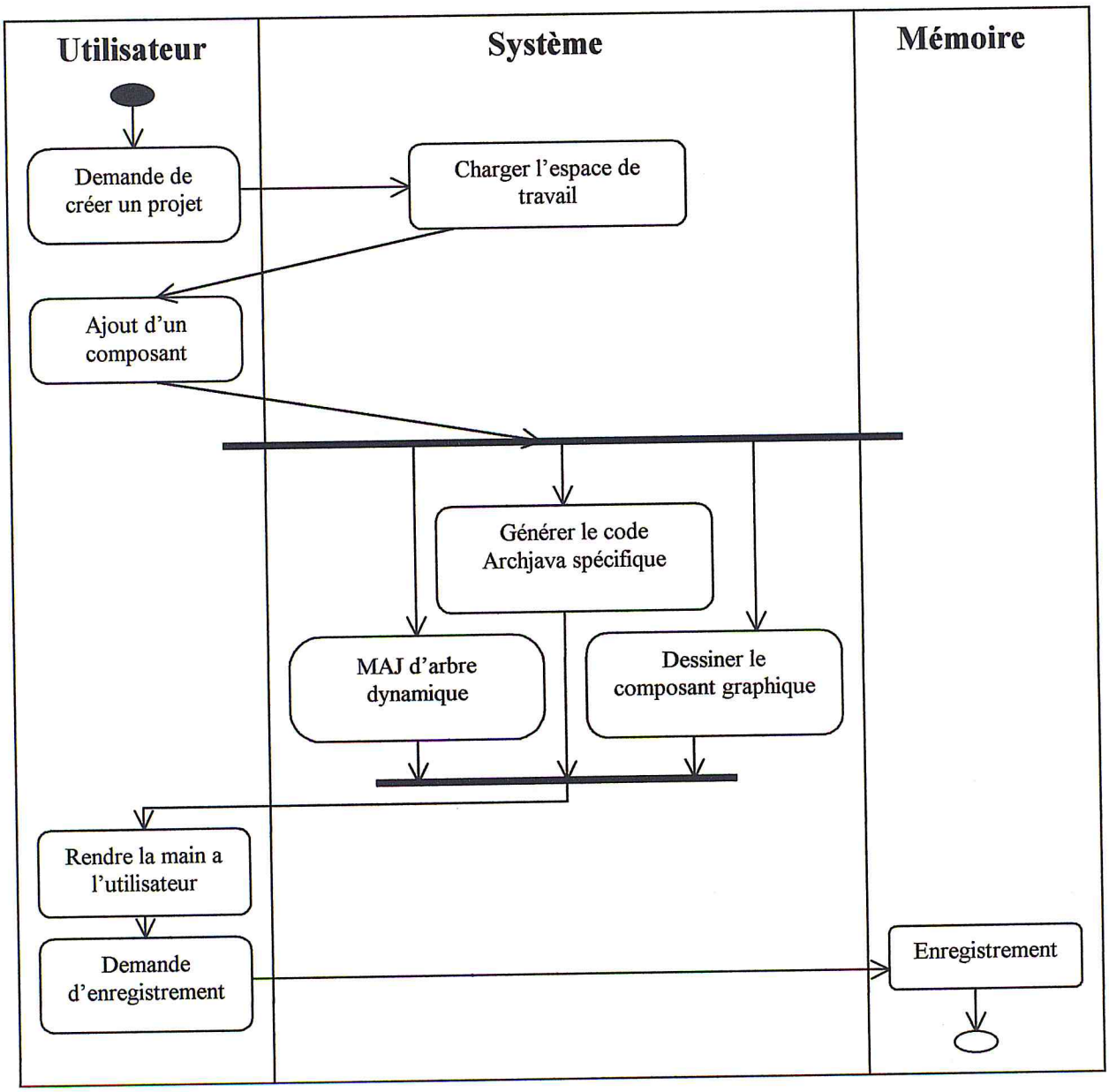

Figure 4.20: Diagramme d'activité pour l'opération « Création d'un composant logicielle »

- **Ajout d'un port de communication :**

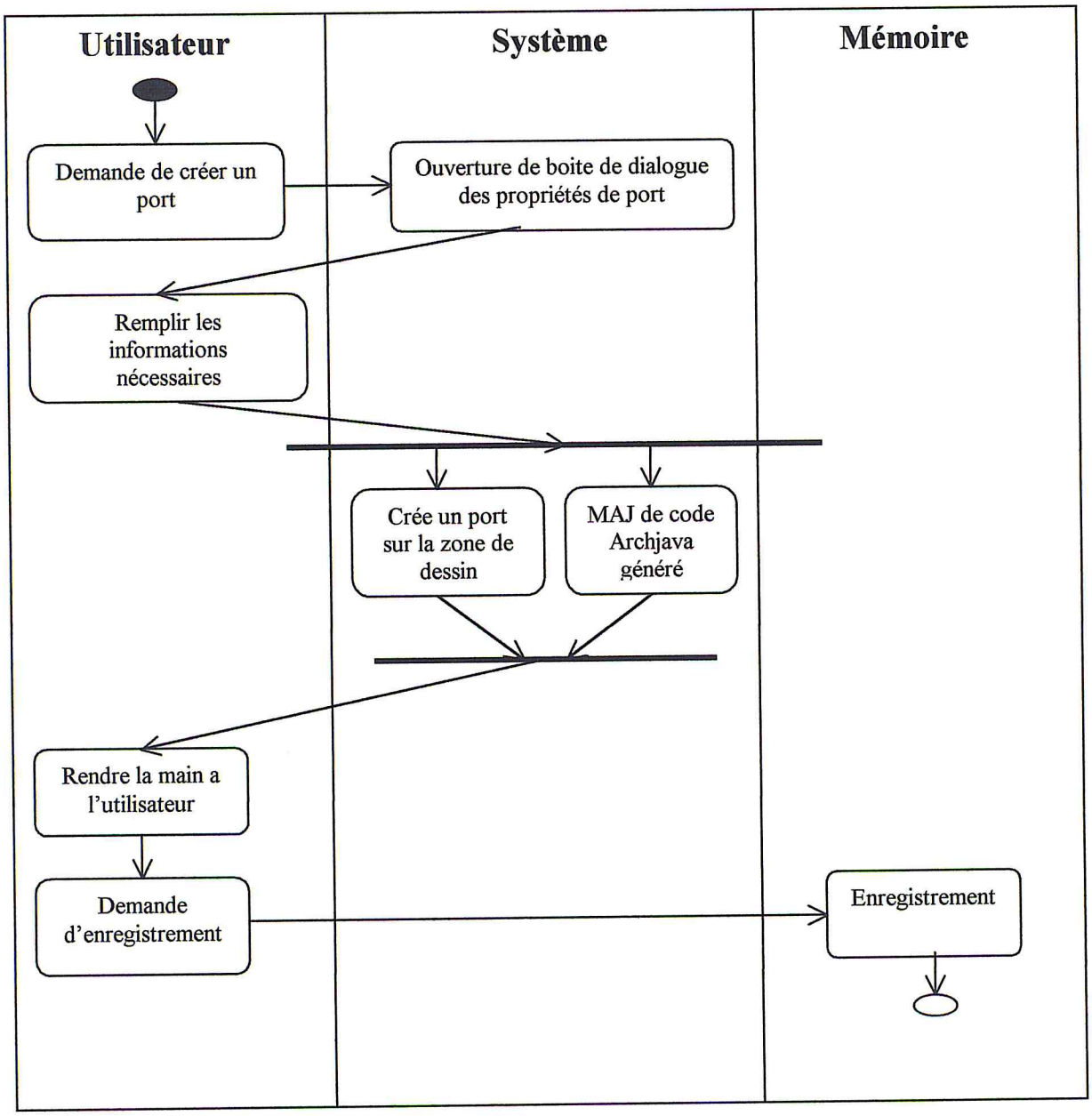

Figure 4.21: Diagramme d'activité pour l'opération « Ajout d'un port de communication»

- **Réalisation d'une Interconnexion entre composants :**

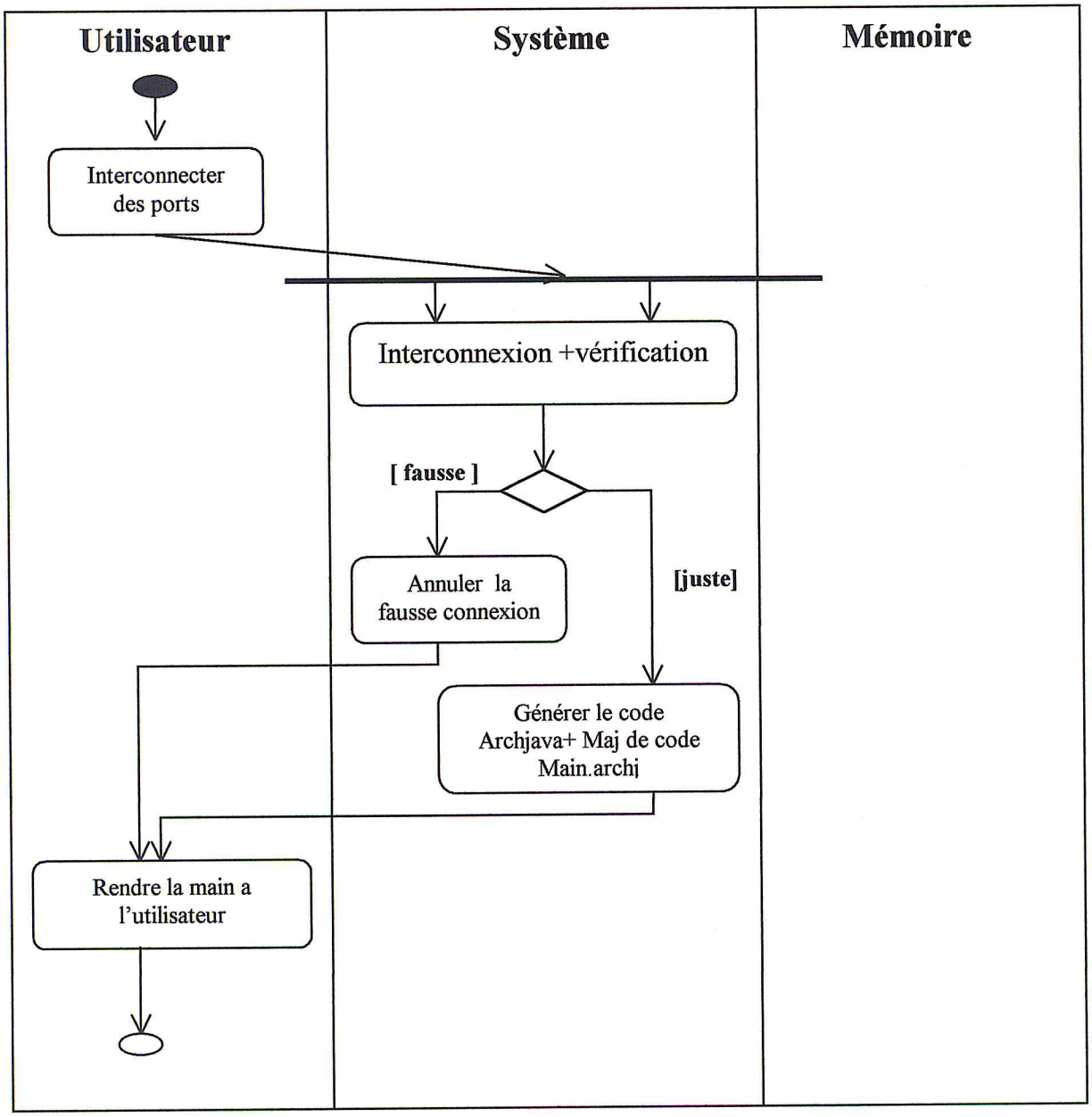

Figure 4.22: Diagramme d'activité pour l'opération « Réaliser une Interconnexion »

- **Importer des composants de la bibliothèque des composants primitifs :**

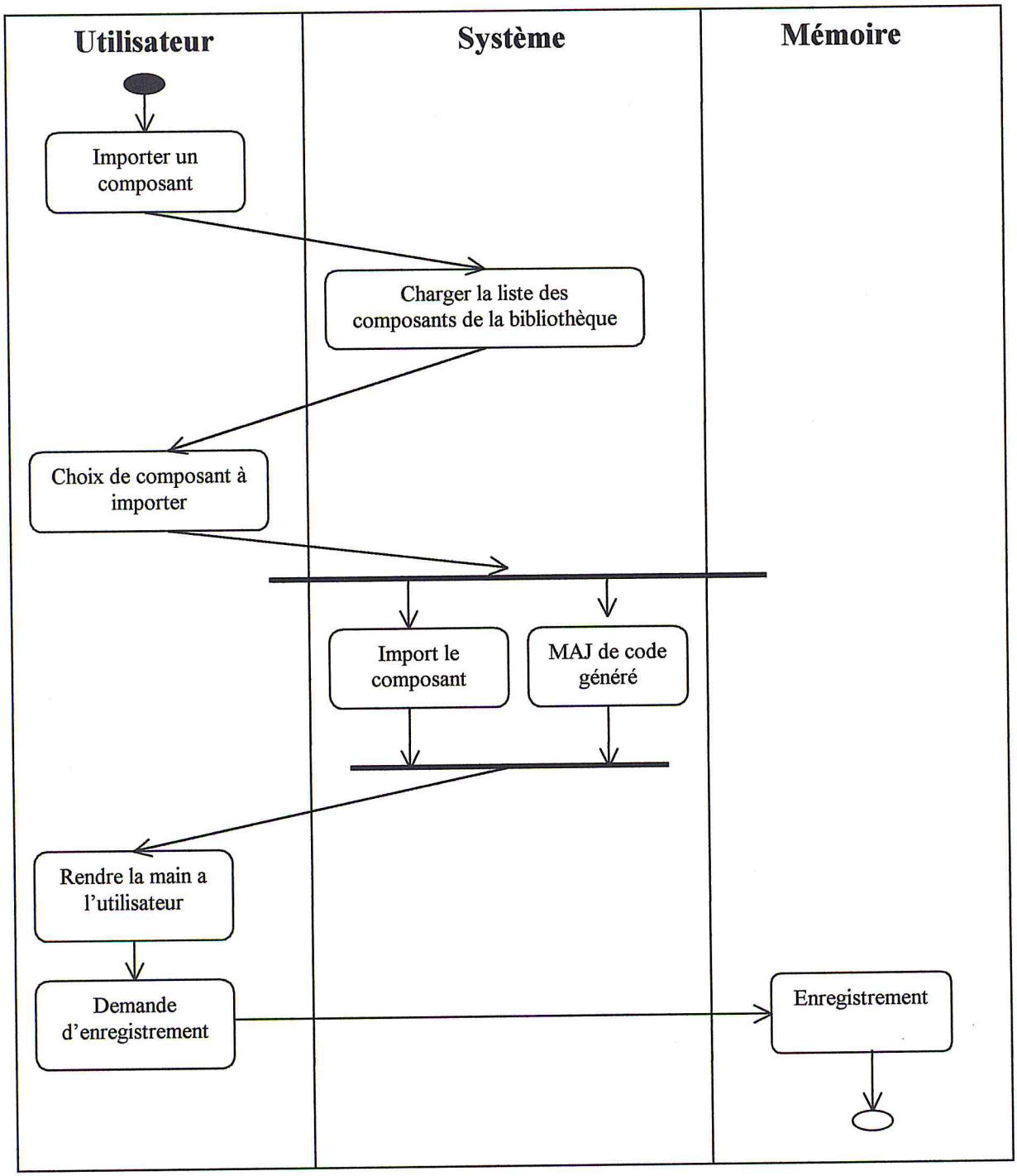

Figure 4.23: Diagramme d'activité pour l'opération « Importer des composants de la bibliothèque des composants primitifs»

- **Validation d'une architecture logicielle :**

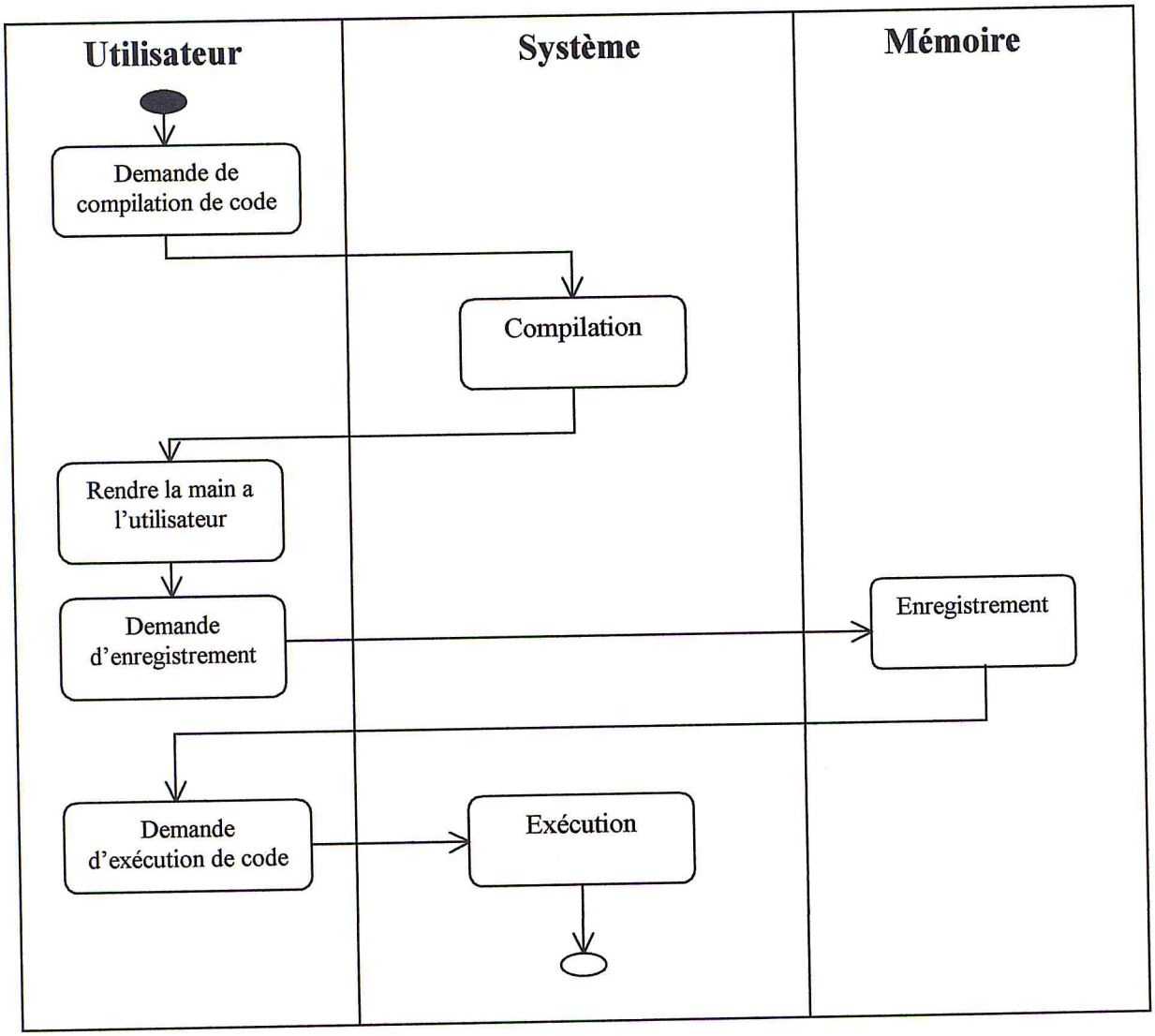

Figure 4.24: Diagramme d'activité pour l'opération « Validation d'une architecture logicielle »

CHAPITRE V : *Implémentation*

V.1-Introduction :

Cette phase correspond à la programmation proprement dite des fonctions sur la base des informations venant de la phase conception[10].

Cette partie sera organiser comme suis, nous commençons par décrire l'environnement De programmation ensuite, nous allons décrire comment notre architecture est implémentée, En se basant sur les diagrammes de composants.

Les diagrammes de composants sont des diagrammes que l'on trouve dans la modélisation des aspects physiques des systèmes, on les utilise pour modéliser la vue d'implémentation statique [11] d'un système.

V.2- Environnement de programmation :

L'IDE développé est implémenté sous le système d'exploitation Windows XP, sur un ordinateur personnel (PC) Intel Pentium 4, avec le langage de programmation *Java* version *j2sdk1.5.0,*

Java est un langage de programmation orienté objet, multi plates formes qui permet son concepteur d'écrire une fois pour toutes des applications capables de fonctionner dans tous les environnements, et pour implémenter ce système (IDE) on a fait appelle à de nombreux packages et classes qu'offre *Java*, parmi les on cite les principaux entre eux qui sont :

- **La bibliothèque SWING :** c'est une librairie fournie par java pour construire des interfaces graphiques, elle offre un ensemble de composants graphiques pour la construction des interfaces graphiques plus sophistiquées, parmi ces composants on a utilisé JFrame, JButton, JPanel … etc.

- **Java2D** : c'est une librairie qui offre un ensemble d'outils de dessin pour développer des graphiques élégants, professionnels, et de haute qualité, le point d'entrée a cette librairie est la classe *Graphics2D* qui permet au programmeur de programmer des dessins.

- **Java.io :** java comme tout langage de programmation fourni des API de manipulation des flux de données, quasiment touts les classes de cette API sont localisées dans le package java.io.* dont les classes principales qu'on a utilisées sont :

  - **Java.io.File :** c'est une classe qui permet de manipuler des fichiers.

  - **Les flux d'entrés/sorties java :** (stream en anglais) qui servent à effectuer des entrées *Inputstream* ou sorties *Outputstream* de données en Java.

V.3 -Implémentation des modules de notre architecture :

Nous avons vu précédemment dans la section III.3.1, l'architecture générale de notre environnement développé ainsi que les différents modules qui constituent cette architecture; Dans cette phase nous allons décrire comment chaque module est implémenter.

Alors que tous les diagrammes précédents représentaient une vue logique des différents modules de système, il s'agit cette fois-ci d'une vue des composants physiques des classes Qui forment chaque module, il est possible de modéliser cette vue physique par des diagrammes de composants.

Le diagramme de composants qui correspond à l'architecture de notre système est le suivant :

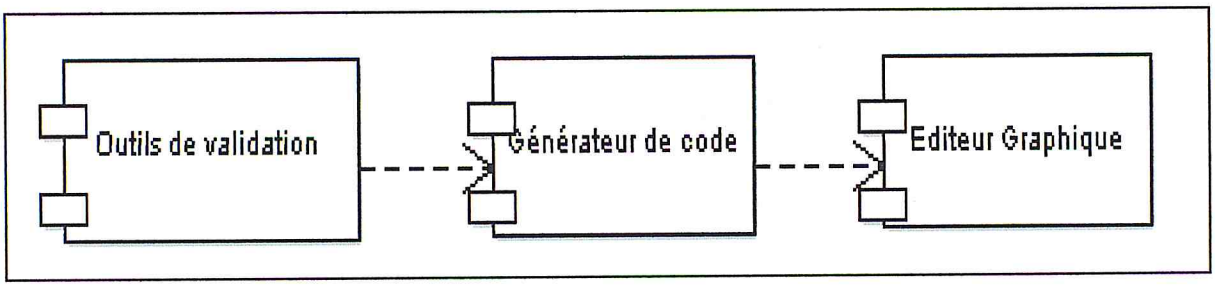

Figure 5.1: Diagramme de composants de l'architecture de système

V.3.1-Implémentation de module Editeur graphique:

Comme on l'a déjà cité, ce module offre à l'utilisateur de système la possibilité de spécifier chacun des éléments de son architecture de manière graphique en lui offrant un ensemble d'outils.

Pour implémenter ce module nous avons crée un ensemble de classes; Pour modéliser la vue physique de chaque classe, on a fait appelle aux diagrammes de composants.

- **Classe Drawing :**

C'est la principale classe de l'éditeur graphique, elle est responsable de la création de tous les éléments graphiques qu'offre notre système comme les composants graphiques, les ports, les connecteurs ...etc., Elle gère aussi toutes les opérations graphiques tels que le dessin d'un composant sur la zone de dessin, dessin d'un port, d'un connecteur etc., l'animation des éléments graphique de l'architecture (déplacements des composants et le calcul les nouvelles positions de ces derniers), et la suppression des éléments graphiques etc.

NB : les éléments graphiques sont les composants, les connecteurs, les ports de communication, les composites.. etc.

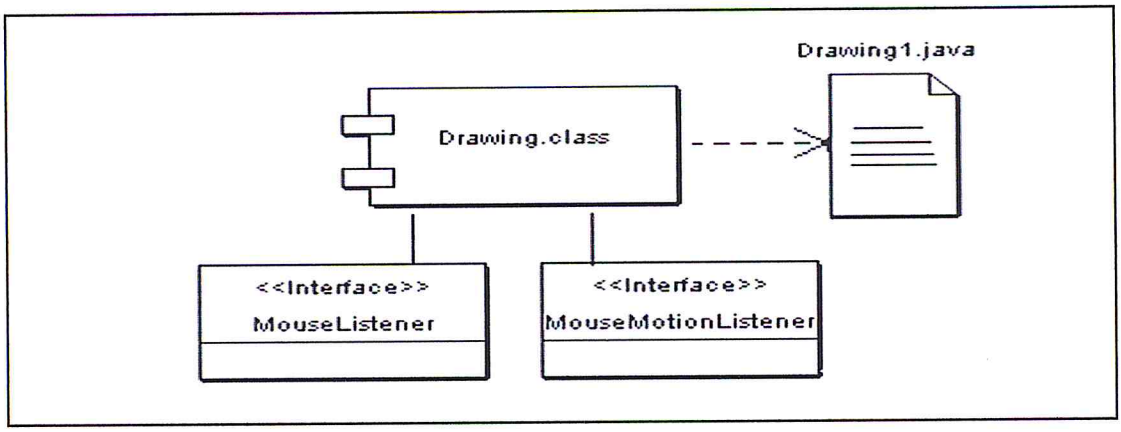

Figure 5.2: Diagramme de composants pour la classe Drawing.java

Les principales méthodes fournies par cette classe sont les suivantes :

1. ***Méthode paint(Graphics g) :*** gère le dessin sur la zone de dessin de système.
2. ***Méthode redessiner() :*** Rafraîchie la zone de dessin.
3. ***Méthode Connexion() :*** Responsable de l'interconnexion graphique des composants logiciels.
4. ***Méthode verifierConnexion() :*** Elle contrôle la justesse des interconnexions graphiques établies entres les composants, si par exemple l'utilisateur interconnecte deux composants via deux ports identiques in/in ou out/out elle signalera erreur et annulera en suite cette connexion.
5. ***Méthode Memory(int x,int y) :*** Mémorise les nouvelles positions de chaque composant déplacé.
6. ***Méthode Select(element e, event e) :*** elle permet la sélection des différents éléments architecturaux.
7. ***Méthode MoveMe(element e, event e, NewposX x, NewposY y) :*** Elle permet de déplacer les différents éléments architecturaux.

- **Classe Methode :** Cette classe définie un ensemble de méthodes que la classe *drawing* les appelle pour dessiner les différents éléments architecturaux parmi ces méthodes on cite :

1. **Méthode PaintComp(entier x, entier y, String nom, Graphics g) :** Dessine un composant.
2. **Méthode PaintConnector(entier x, entier y, entier x1, entier y1,entier x2, entier y2,Line2D l1,Line2D l2,Graphics g2) :** Dessine un connecteur.

3. **Méthode PaintPortIN(entier x, entier y, String nomportOut,Graphics g) :**Dessine un port In.
4. **Méthode PaintPortOut(entier x, entier y, String nomportOut, Graphics g) :** Dessine un port Out.

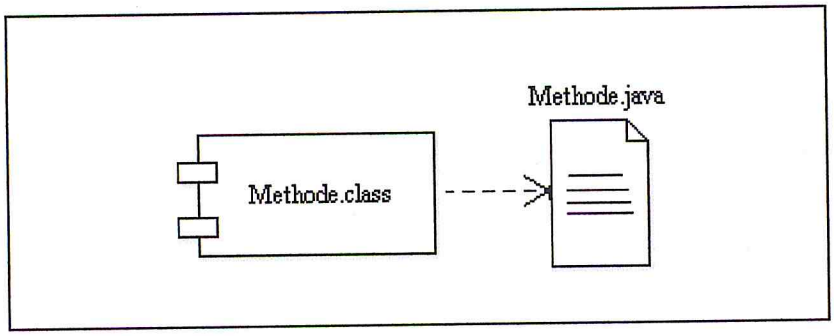

Figure 5.3: Diagramme de composants pour la classe Methode.java

V.3.2- Implémentation de module Générateur de code:

Comme nous avons déjà vu, ce module est responsable de la génération de code Archjava spécifique a la représentation graphique, parmi les principales classes de ce module nous citons:

- **Classe MyEditor:**

Cette classe est responsable de la gestion de génération de code, elle permet de générer un code cohérent qui correspond à la représentation graphique de l'architecture logicielle dessiné sur la zone de dessin de l'éditeur graphique.

- Elle permet de générer le code suite a chaque modification apportée sur la zone de dessin c'est à dire en cas d'ajout ou suppression d'un composant ça entraîne une modification de code généré, nous pouvons dire qu'elle fait le nécessaire pour que le code soit cohérent avec l'architecture spécifiée.

- En cas de connexion, d'importation d'un ou plusieurs composants de la bibliothèque des composants primitifs elle contrôle et gère la génération de code.

- Cette classe permet aussi de générer des fichiers avec l'extension **'.archj'** .

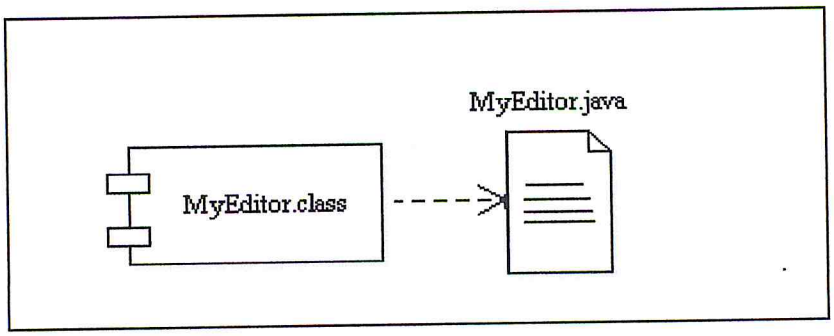

Figure 5.4: Diagramme de composants pour la classe MyEditor.java

V.3.3-Implémentation de module Outils de validation d'architecture:

Ces outils sont responsables de la validation de l'architecture logicielle c'est à dire vérifier si le code Archjava générer par le générateur de code se conforme à l'architecture conçue. Elle A composé principalement des classes suivantes :

- **Classe Compiler :**

Cette classe est responsable de compilation de code Archjava généré, que se soit le code des composants ou le code Main.archj qui correspond a la spécification de toutes l'architecture avec ADL Archjava.

Pour compiler ces codes cette classe fait appelle au compilateur *Archjava.exe* version 1.3.

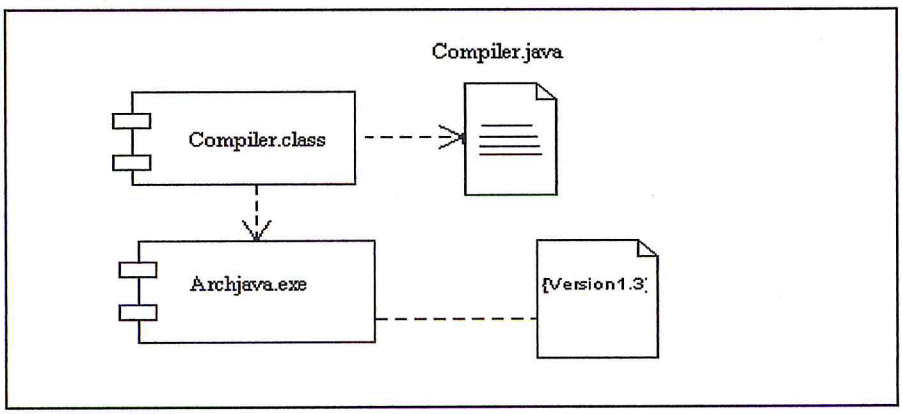

Figure 5.5: Diagramme de composants pour la classe Compiler.java

- **Classe Executer :**

Cette classe est responsable de l'exécution de code *Archjava* généré et qui correspond a la spécification d'architecture logicielle en ADL Archjava. Pour exécuter ces codes cette classe fait appelle au compilateur *Archjava.exe* version 1.3.

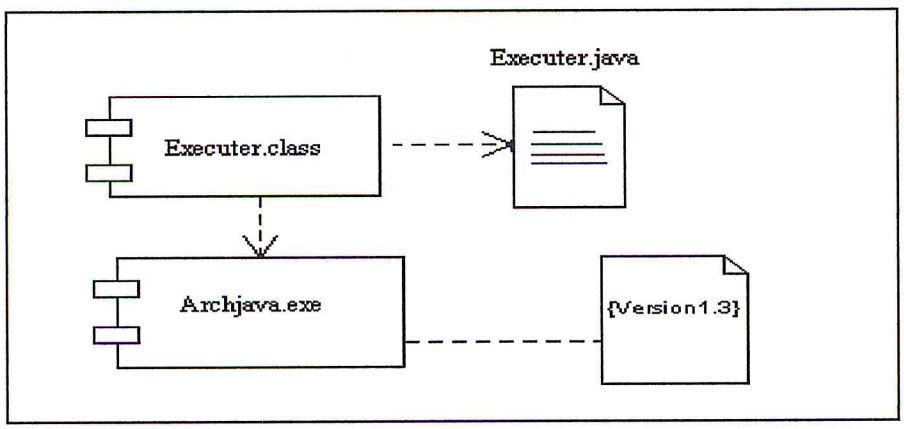

Figure 5.6: Diagramme de composants pour la classe Executer.java

- **Fichier batch :**

Pour nous faciliter le travail nous avons utilisé un ensemble de fichier batch pour lancer quelques commandes système.

V.3.4-Implémentation de l'interface utilisateur :

L'interface utilisateur constitue une partie très intéressante de l'application développée notamment avec l'avancé technologique au niveau ergonomique, elle est implémentée en se basant sur les classes suivantes :

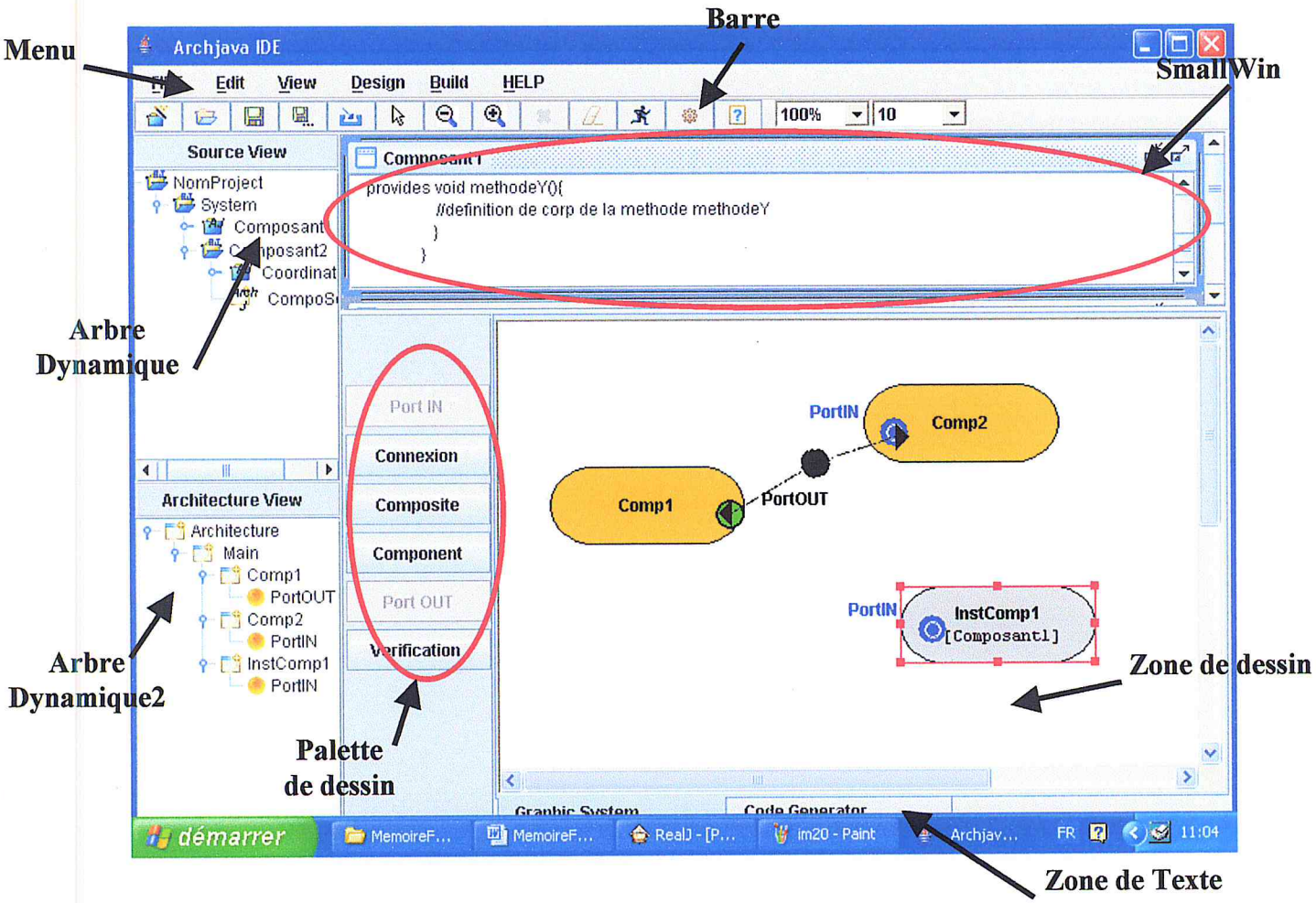

Figure 5.7: Vue générale de l'interface utilisateur de système

- **Classe DynamicTree :** qui est l'implémentation physique de l'arbre dynamique de l'application, cet arbre comme on a déjà mentionné sert à donner un aperçu de tous les composants logicielles créés, et offre à l'utilisateur plusieurs fonctionnalités qui lui facilitent la tâche tels que :

1. ***Add Component to library***: Ajout d'un composant (le nœud de l'arbre sélectionné par l'utilisateur) a la bibliothèque des composants primitifs.
2. ***Save Component*** : Sauvegarde le code Archjava de composant (le nœud de l'arbre sélectionné) dans l'archive de système avec l'extension '.archj'.

3. ***Compile Component code*** : Lance la compilation de code Archjava de composant (nœud de l'arbre sélectionné) .

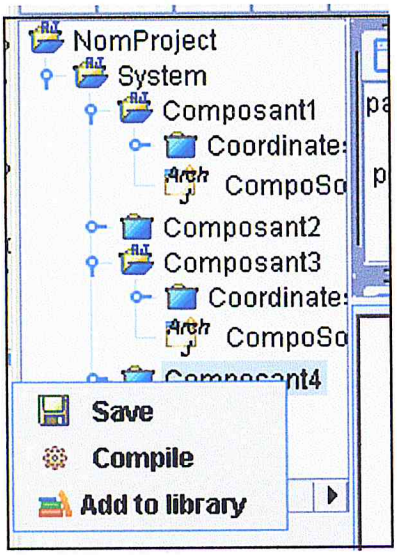

Figure 5.8: Vue générale de l'arbre Dynamique de système (Arbre Source view)

- **Classe SmallWin :** La classe SmallWin est l'implémentation physique de la fenêtre de code de composant, qu'a pour rôle d'afficher le code Archjava de chaque composant crée par l'utilisateur, elle contient une zone de texte éditable pour permettre à l'utilisateur d'ajouter un code Archjava supplémentaire.

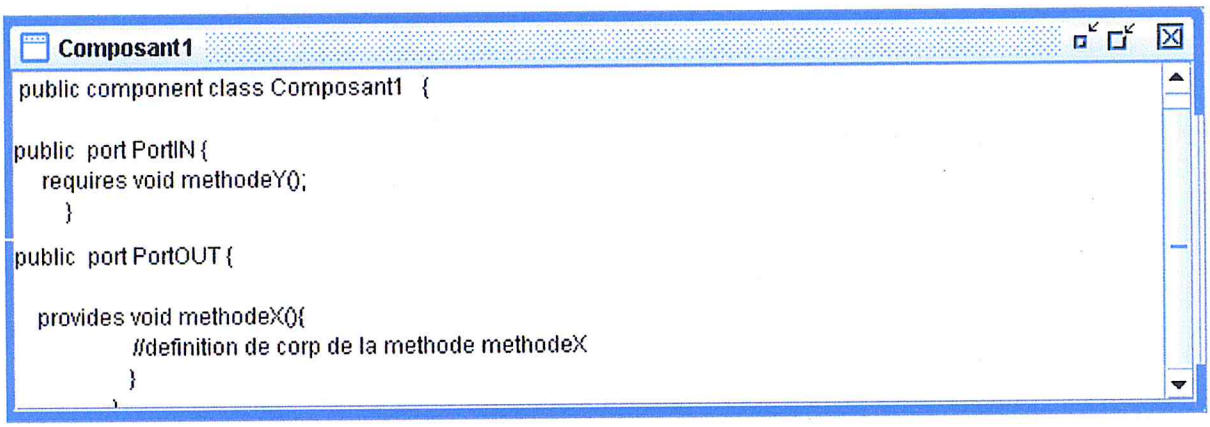

Figure 5.9: Vue générale de smallWin

- **Classe MyMenubar :** Cette classe est l'implémentation physique de menu (voir fig.5.7) qui est conçu pour offrire a l'utilisateur de système quelques fonctionnalités, tels que : fichier, édition, affichage, conception, … et l'aide.

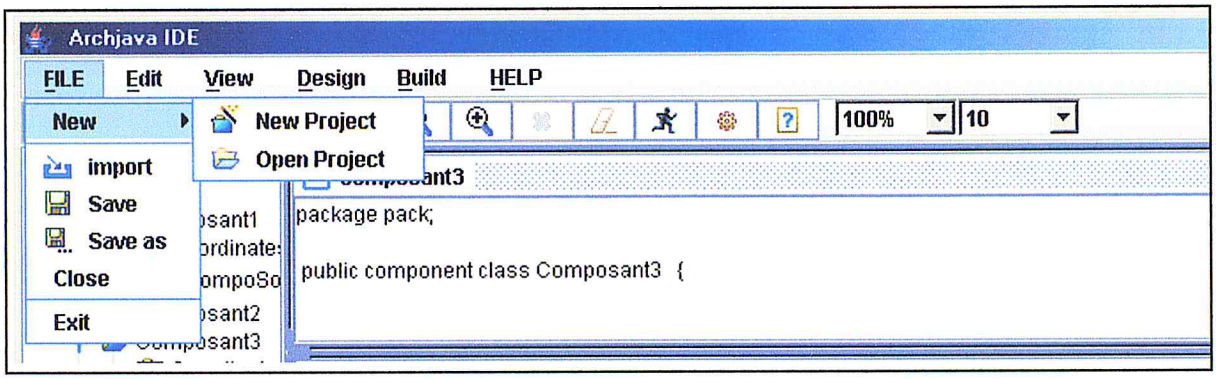

Figure 5.10: Vue générale de menu de système

- **Classe Architecture view :**

Cette classe est l'implémentation physique de l'arbre dynamique2 de l'application, cet arbre comme on a déjà mentionné sert à donner un aperçu de toutes les instances des composants participants a la spécification de l'architecture logicielle, chaque nœud de cet arbre est une instance d'un composant qui fait partie de l'architecture spécifiée. Elle permet de donner une vue générale de contenu de la zone de dessin, chaque nœud contient des sous nœuds qui représente les ports de nœud parent comme montre la figure fig.5.11 ci-après.

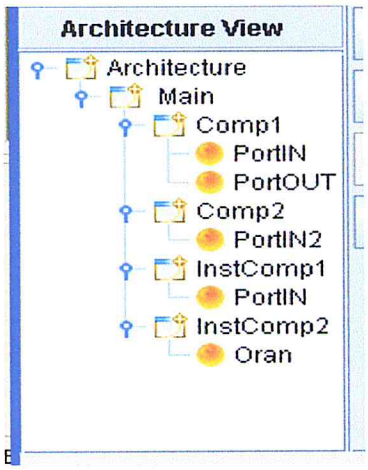

Figure 5.11: Vue générale de l'arbre dynamique2 de système (Arbre architecture view)

- **Classe ZMyBar :** Cette classe est l'implémentation physique de menu2 (voir fig.5.7) qui est une barre de menus qui contient des raccourcis pour réaliser quelques opérations tel que suppression, zoom in, zoom out, sélection, enregistrer sous, import …etc.

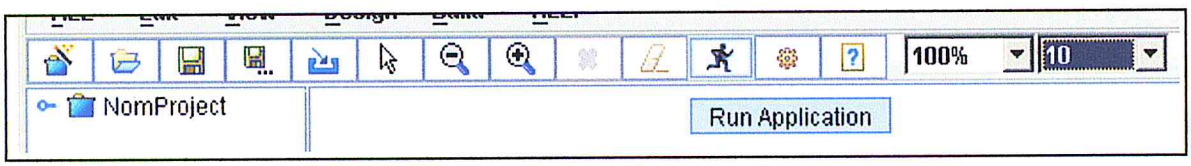

Figure 5.12: Vue générale de la barre de système

- **Classe System Consol :** Cette classe est l'implémentation physique de console d'affichage des résultats de système, elle reçoit les résultas de compilation/exécution des codes générés et les affiche pour visualiser à l'utilisateur les résultats pour qu'il puisse ensuite corriger ces erreurs s'ils y'on a.

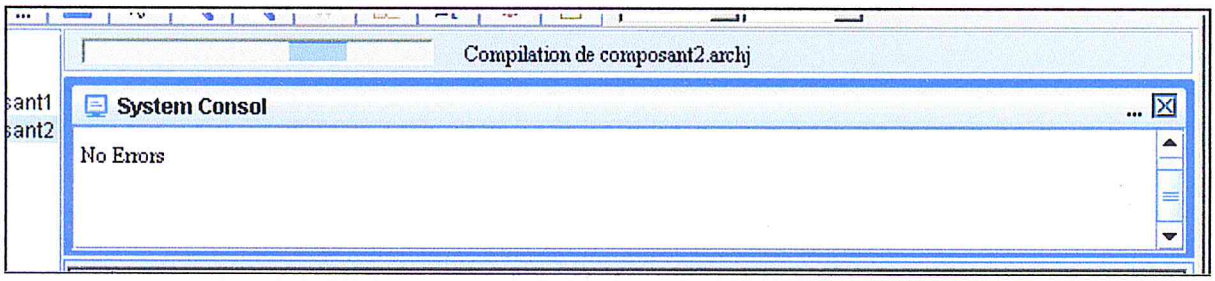

Figure 5.13: Vue générale de console d'affichage de système

Un ensemble de boite de dialogue qui servent comme une interface entre l'utilisateur et le système, parmi les on cite :

- **Classe ImportDialog :** Pour importer des composants de la bibliothèque des composants primitifs afin de les réutiliser pour spécifier des nouvelles architectures logicielles.

- **Classe FileBoite :** Pour sauvegarde les code des composants dans des fichier avec l'extension '.archj'.

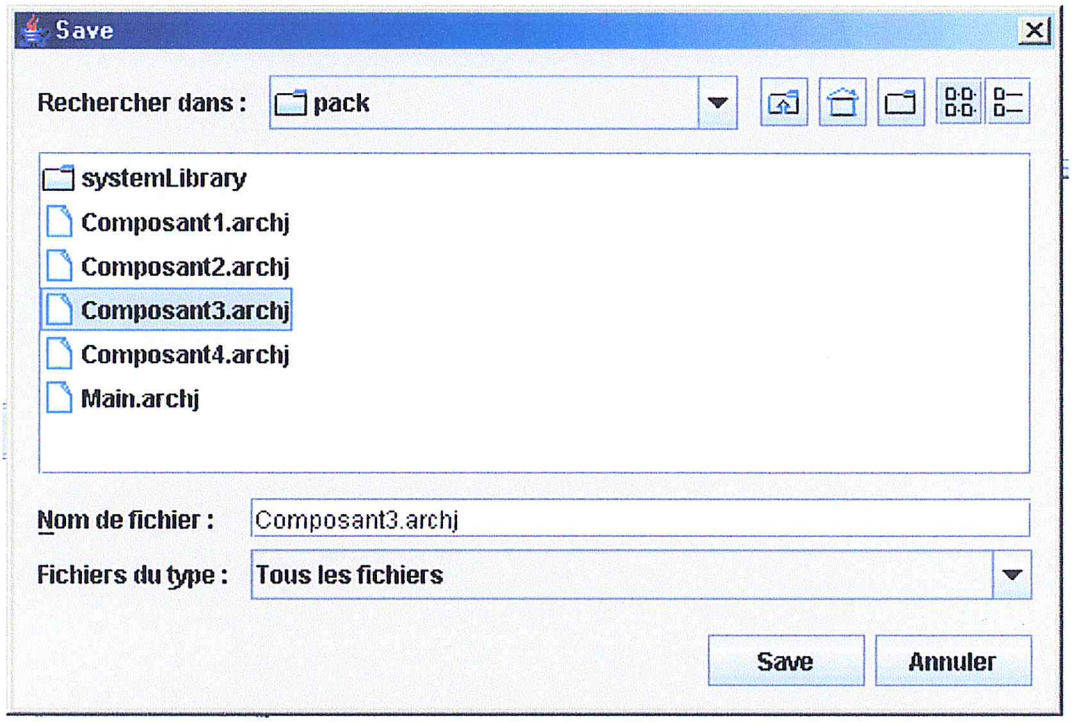

Figure 5.14: Vue générale de FileBoite

- **Classe PortInfoIn :** C'est une interface pour remplir des données concernant les ports in ajoutés tels que e nom de port, nom de la méthode requise par ce port, les type de retours de la méthode tels que void, String, et. etc.

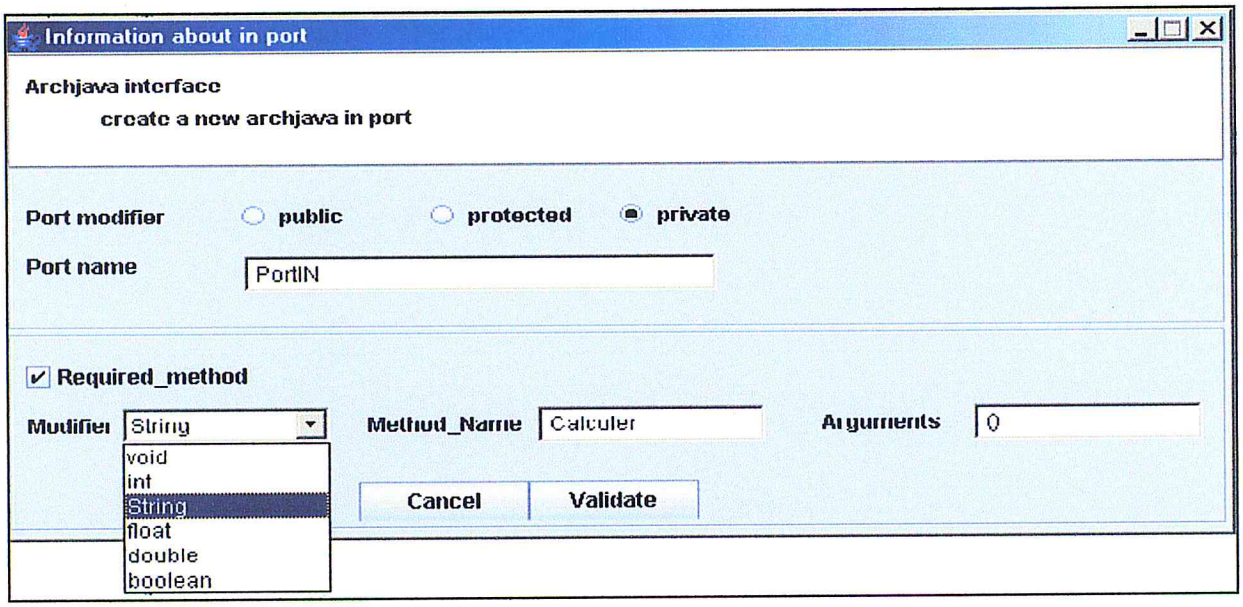

Figure 5.15: Vue générale de la boite de dialogue portinfoIn

- **Classe PortInfoOut :** C'est une interface pour remplir des données concernant les ports out ajoutés tels que e nom de port, nom de la méthode fournie par ce port, les types de retours de la méthode tels que void, String, et. etc.

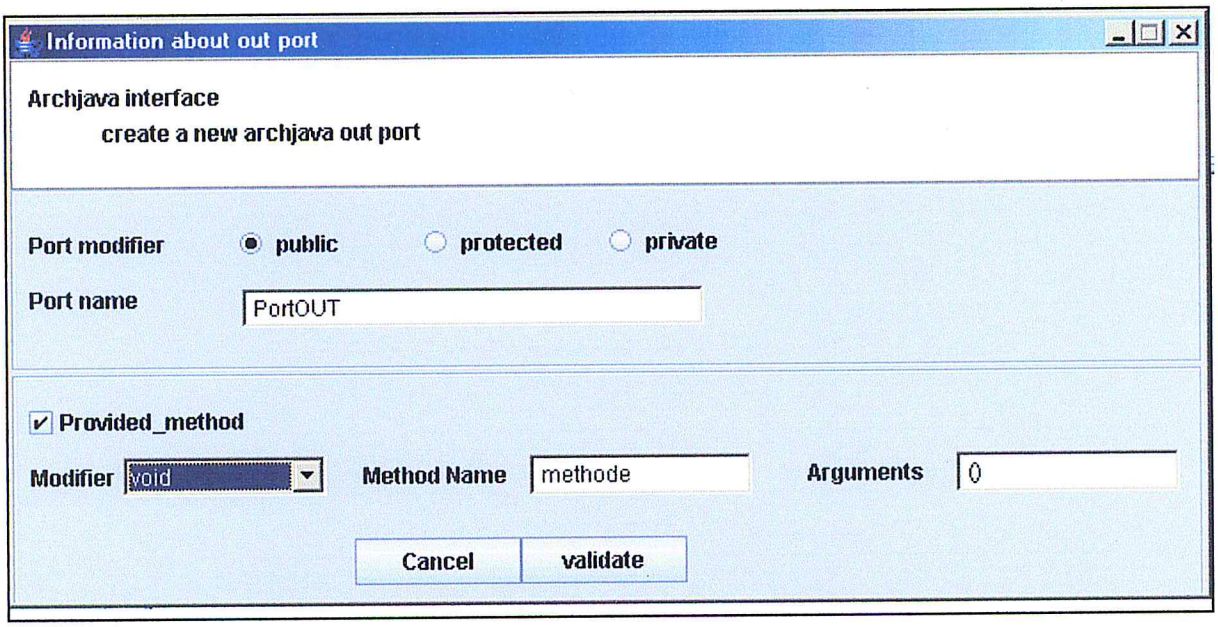

Figure 5.16: Vue générale de la boite de dialogue portinfoOut

- **Classe MyDialog :**C'est une fenêtre de dialogue qui sert à visualiser un message d'erreur a l'utilisateur en cas d'établissement d'une fausse connexion.

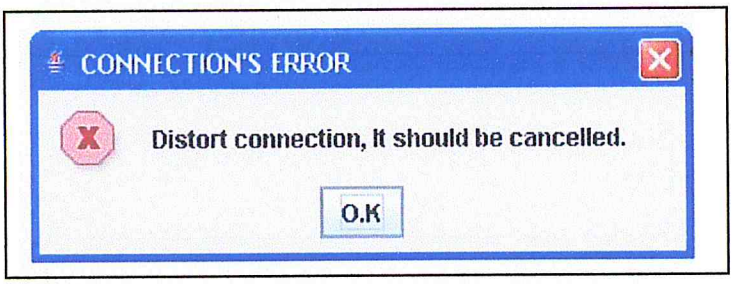

Figure 5.17: Vue générale de console de fenetre d'erreur de système

V.4- Problème rencontrer lors de réalisation :

V.4.1-probleme de rafraîchissement :

L'un des problèmes majeurs qu'on avait rencontrés durant le développement de système et le rafraîchissement du dessin après que une fenêtre de dialogue s'ouvre ou après que la fenêtre principale se fait masquer, redimensionner, icônifier ...etc, c'est à dire dans les cas cités précédemment, le dessin graphique disparaître ou s'efface simplement de la zone de dessin.

Chaque composant graphique possède une méthode qui définit comment il doit se dessiner
public void paint(Graphics g) pour les composants awt
public void paintComponent(Graphics g) pour les composants swings

Mais dès que l'application gère ses propres graphiques via un contexte graphique (objet Graphics) comme le cas de notre application elle devra se soucier de leur rafraîchissement

Solution adaptée :
Pour résoudre le problème de rafraîchissement du dessin on a procédé comme suit :

- Redessiner l'architecture disparus :

1- Pour ceci il faut stocker les informations pour afficher à nouveau tous les éléments de l'architecture déjà dessinée, c'est pour cette raison nous avons défini la méthode suivante

Méthode public void Memory (élément e, emplacement p) {}

- L'élément peut être un composant, un connecteur, un port etc.
- Emplacement est les coordonnées de l'élément en x et y dans le repère de dessin.

2- Redessiner à nouveau tous les éléments de l'architecture disparus à partir des donnés stockés par la méthode Memory(), c'est pour cette raison nous avons défini la méthode suivante:

Méthode public void redessiner() {}

V.4.2- problème de non-stabilité de dessin lors de déplacement d'un élément graphique :

Ce problème est apparu lorsqu'on a déplacé fréquemment les composants graphiques sur la zone de dessin ce qu'a impliqué le rafraîchissement fréquent de dessin en redessinant le composant déplacé dans les nouvelles positions ce qu'a causé des flashs ou une sorte d'instabilité de dessin.

Solution adaptée :

L'utilisation de ce qu'on appelle en java le double buffered en utilisant la classe `BufferedImage`

La classe `BufferedImage` peut être employée pour préparer les éléments graphiques a l'extérieur de l'écran sur un objet `image`, puis les recopient à l'écran, cette technique est particulièrement utile quand un graphique est complexe et animé à plusieurs reprises comme le cas de notre système.

- Dessiner dans un objet image (un `BufferedImage`) en appelons la méthode `BufferedImage.createGraphics`, qui renvoie un objet `de Graphics2D`. Avec cet objet, en peut appeler toutes les méthodes `de Graphics2D` pour dessiner les éléments de notre architecture.

Voilà un extrait de code :

```
// Récupérer le contexte graphique de système (l'écran)
Graphics2D g = (Graphics2D)getGraphics();

//Créer un BufferedImage
BufferedImage bi ;
bi=(BufferedImage) createImage(this.getWidth(),this.getHeight());

//Crée un graphics2D qui vas être utilisé pour dessiner dedans

Graphics2D g2=bi.createGraphics();
//dessiner les éléments graphiques
g2.draw(……);
```

- Dessiner le `BufferedImage à l'écran`

Voilà un extrait de code :

```
// Dessiner le BufferedImage à l'écran
   g.drawImage(bi,0,0,this);
```

V.5- Sauvegarde de l'architecture :

Pour sauvegarder notre architecture nous avons fait appelle aux différentes classes et méthodes suivantes :

- **Classe Zxamp** : C'est une classe qui fait extraire les données nécessaires au redessine des éléments de l'architecture sauvegardée à partir des fichiers de données.

- **Classe AddToLibrary :** Cette classe est invoquer quand l'utilisateur décide d'ajouter un composant a la bibliothèque des composants primitifs.

- **Methode SaveMainCode(String code, String NomFichier) :** Sauvegarde le code Main.archj qui est le code principal qui correspond à la spécification de toute l'architecture logicielle dans l'archive du système.

- **Methode void Sauver(String code, String NomFichier) :** Sauvegarde le code de chaque composants dans des fichier avec l'extension '.archj' dans l'archive du système.

- **Methode SaveCompoInfo(données d, emplacement c) :** Cette méthode est convoquée lorsque l'utilisateur décide de sauvegarder des composants pour les réutiliser plus tard, elle sert à sauvegarder des donnés concernant des composants logiciels crées comme les positions de ces derniers, les noms des composants, le nombre de ports qu'ils contiennent ainsi les méthodes fournies/requis par ces ports.

Tous les fichiers qui continents de code Archjava sont organisés en package ce qui facilite à l'utilisateur leur manipulation, importations et réutilisation.

CHAPITRE VI :
Test & Validation

VI.1-Introduction :

Dans ce chapitre, nous s'assurons par l'exécution du logiciel, que le système conçu satisfait à ces objectifs fonctionnels spécifier en phase d'expression des besoins en testant quelques cas d'utilisation déjà vue dans cette phase. Nous représentons ensuite la spécification d'une architecture logicielle avec l'ADL Archjava en utilisant notre système conçu.

VI.2- Test et validation des principaux cas d'utilisations:

Les cas d'utilisation que nous allons tester sont les suivant :

1- Création d'une architecture logicielle.
2- Génération de code Archjava spécifiant cette architecture.
3- Validation de l'architecture spécifiée.

VI.2.1- Création d'une architecture logicielle :

Comme nous avons vu précédemment, le cas d'utilisation "Création d'une architecture logicielle" comprend plusieurs opération ou étapes comme la création des composants logiciels, l'importation des composants logiciels déjà fait c'est à dire la réutilisation des composants qui se trouvent au niveau de bibliothèque des composants primitifs ou complexes, l'ajout des ports a ces composants, et en fin la réalisation des interconnexions entre les différents composants en reliant les ports de ces composants entre eux. Pour chaque étape nous allons valider le cas d'usage qui lui correspond.

- **Création d'un composant logiciel :**

Pour créer un composant logiciel l'utilisateur clique sur la palette de dessin, puis choisit le bouton correspondant au dessin de composant comme montre la figure ci-dessous et déplace en suite la souris sur la zone de dessin et relâche la souris sur l'endroit ou il veut placer le composant, et le système le dessine.

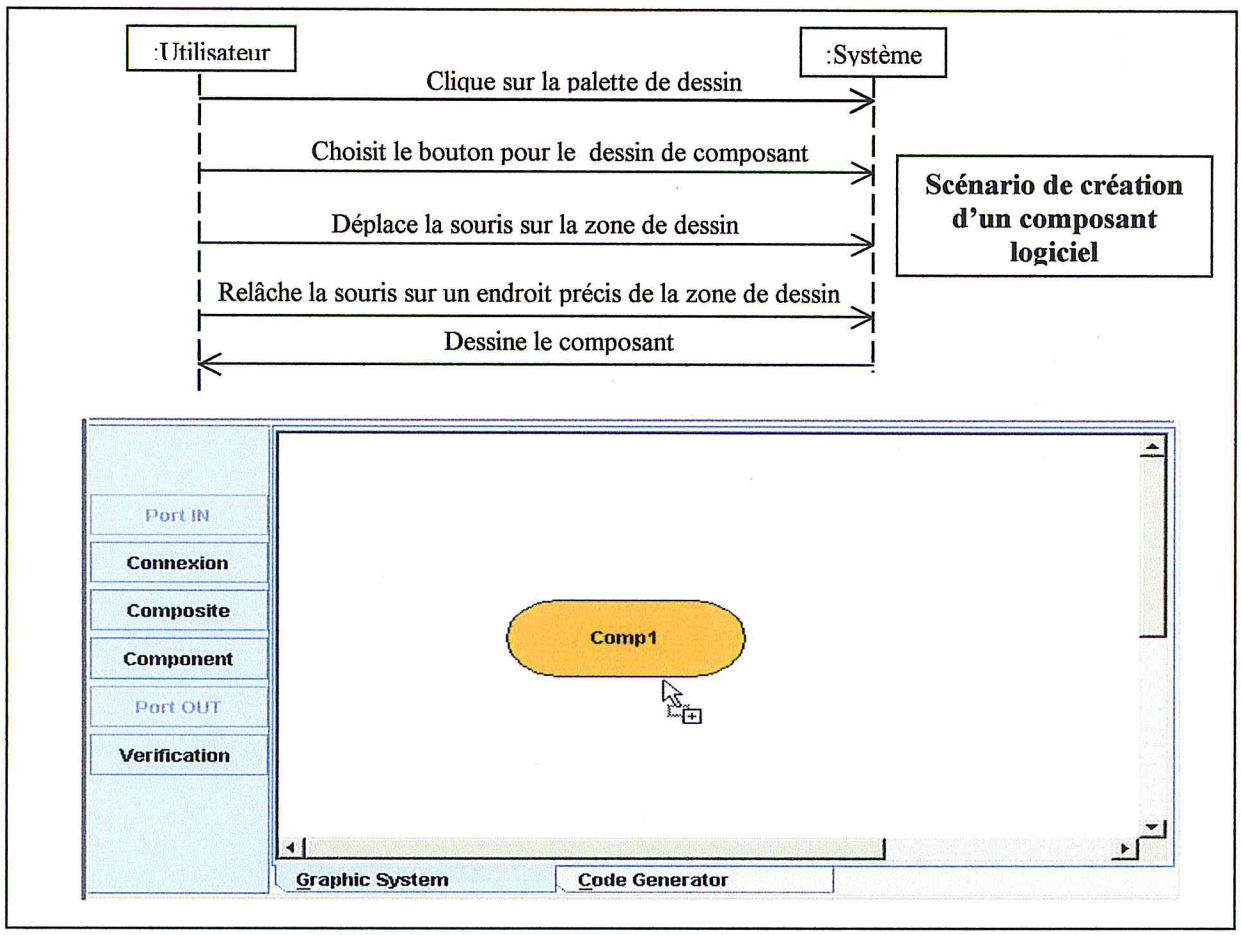

Figure 6.1: Validation de cas d'usage "Création d'un composant logicielle"

- **Importation d'un composant :**

Pour importer un composant logiciel de la bibliothèque des composants primitifs afin de le réutiliser, l'utilisateur demande d'importer un composant puis choisit le composant a importé et enfin le système importe le composant comme montre la figure ci-dessous.

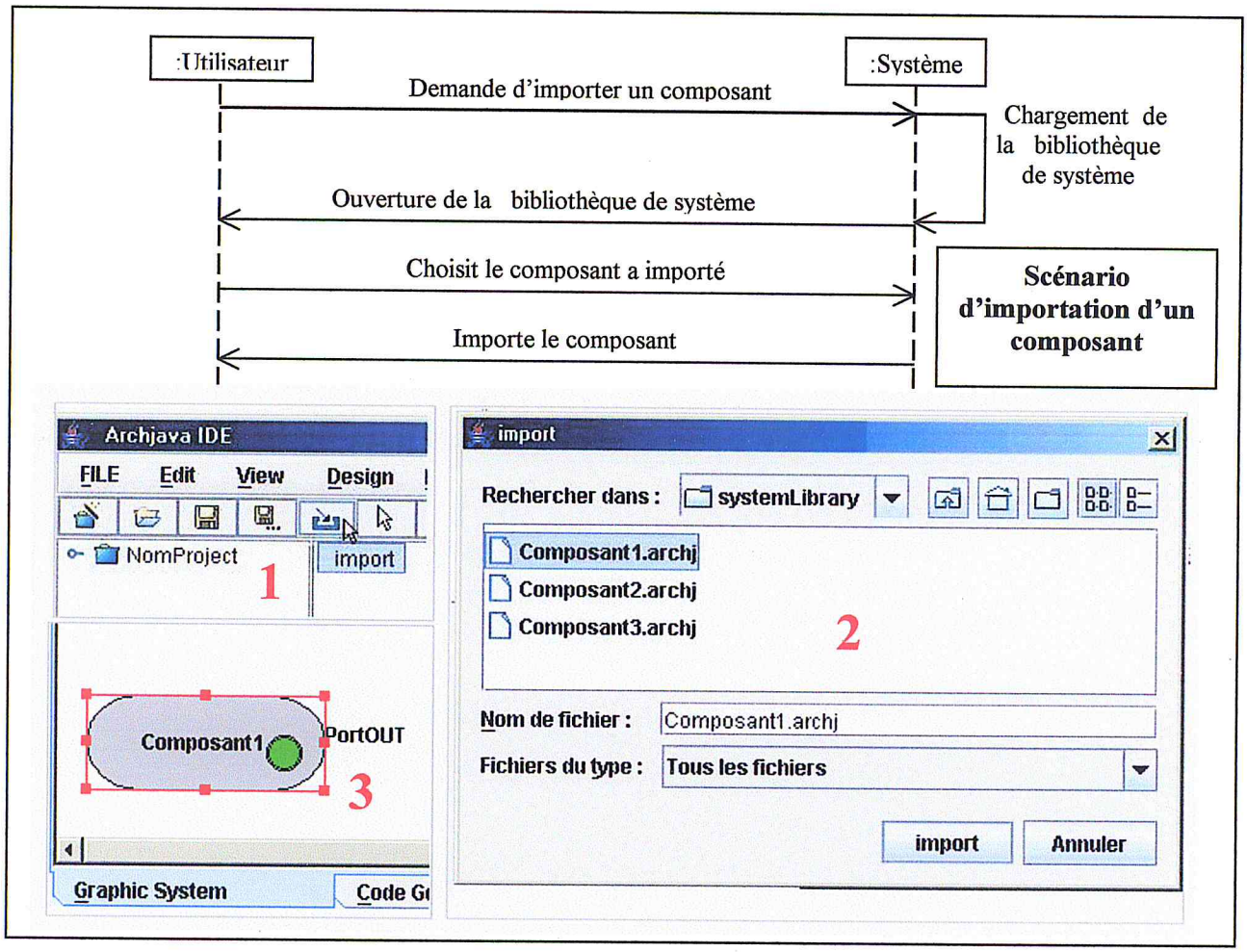

Figure 6.2: Validation de cas d'usage "Importation d'un composant logicielle"

- **Ajout des ports de communication :**

L'ajout d'un port au composant ce fait de la manière suivante: L'utilisateur choisit le type de port a crée (IN ou OUT), en suite il définit les propriétés de ce port. Choisit le composant a qui doit ajouter le port et le système l'ajoute ensuite le port au composant

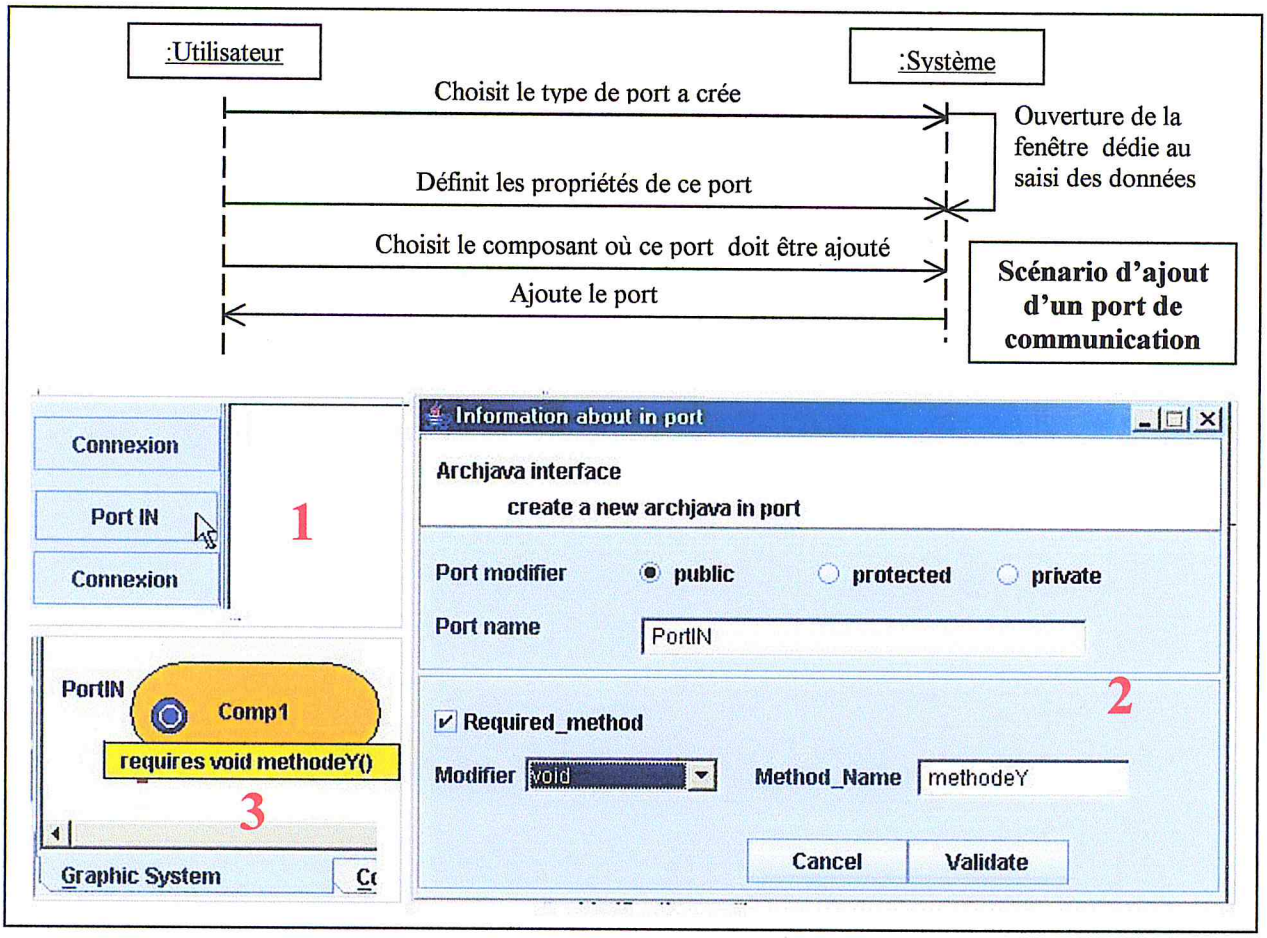

Figure 6.3: Validation de cas d'usage "Ajout d'un port de communication"

- **Réalisation des interconnexions :**

Pour que l'utilisateur réalise des interconnexions entre les composants il choisit le type de connexion c'est à dire statique ou dynamique (voir Annexe B) puis le système dessine un connecteur, l'utilisateur déplace ensuite les rôles de ce connecteur sur les ports à connecter puis le système réalise l'interconnexion entre les ports de ces composants, après il vérifie la justesse de l'interconnexion établit, si la connexion est fausse, le système l'annule si non il la garde.

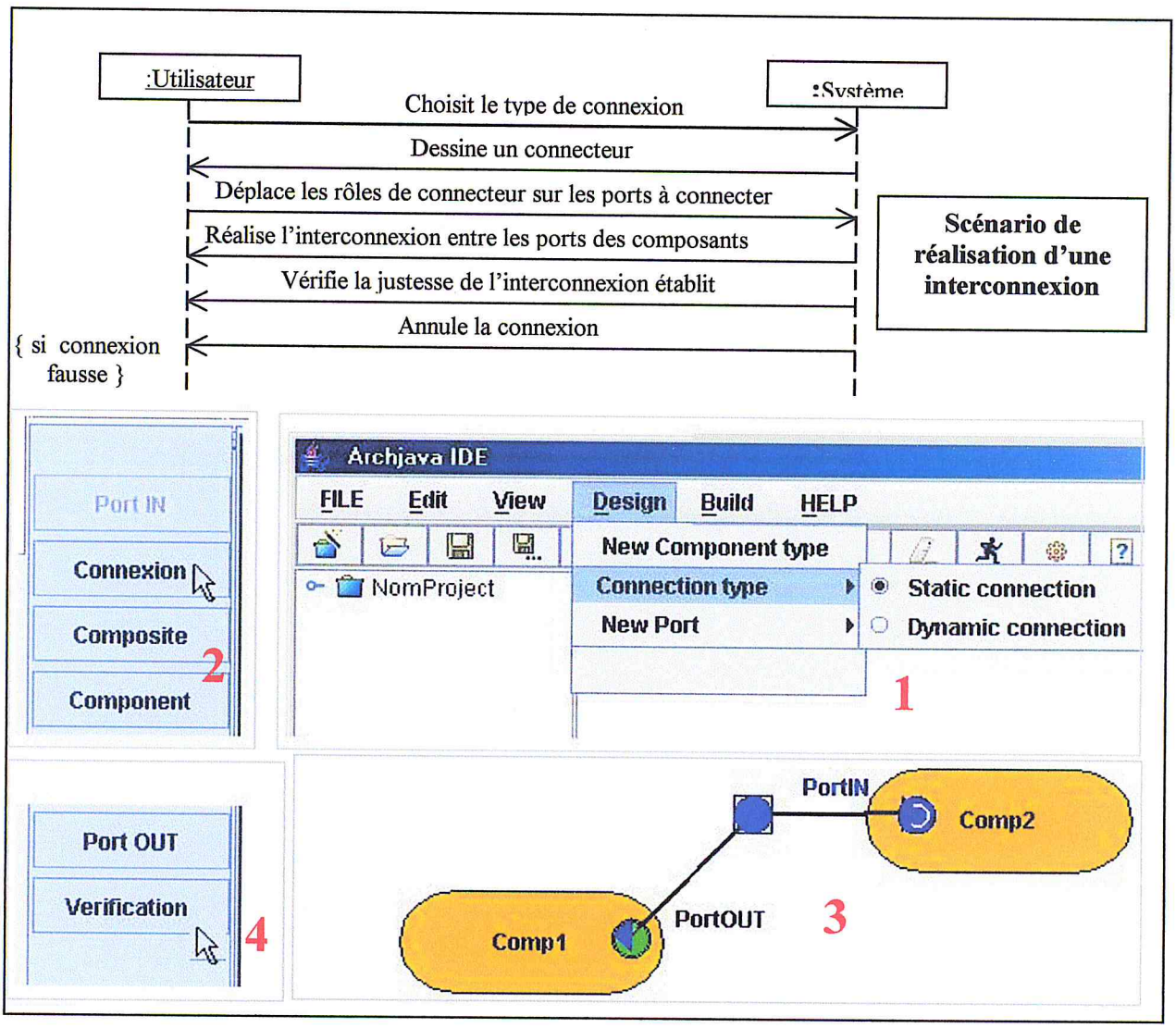

Figure 6.4: Validation de cas d'usage
"Réalisation d'une interconnexion dynamique juste"

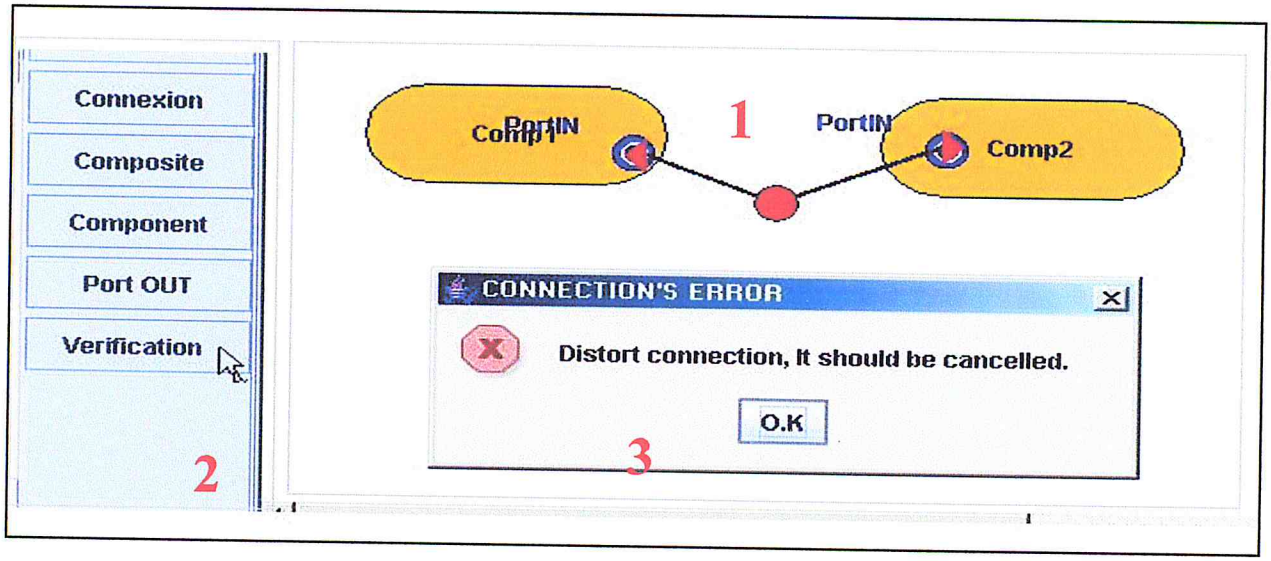

Figure 6.5: Validation de cas d'usage
"Réalisation d'une interconnexion statique fausse"

VI.2.2- Génération de code Archjava:

La génération du code ArchJava correspondant à l'architecture logicielle crée par l'utilisateur, se fait de manière systématique, c'est à dire quand l'utilisateur réalise une action sur la zone de dessin comme la création d'un composant par exemple, le générateur de code vas interpréter cette action en un code Archjava, c'est à dire, il génère automatiquement le code qui correspond à la spécification de composants créés en Archjava.

La génération du code ArchJava comprend les opérations suivantes: la génération de code correspondant à chaque composant logicielle crée/importé, la génération du code ArchJava correspondant à chaque port de communication, et la génération de code principale spécifiant l'ensemble de l'architecture. Pour chaque opération on va valider le cas d'usage qui lui correspond.

- **Générer le code Archjava pour les composants crées:**

Pour générer le code Archjava pour les composants crées l'utilisateur dessine un composant sur la zone de dessin ensuite le système génère le code ArchJava spécifique a ce composant comme montre la figure ci-dessous.

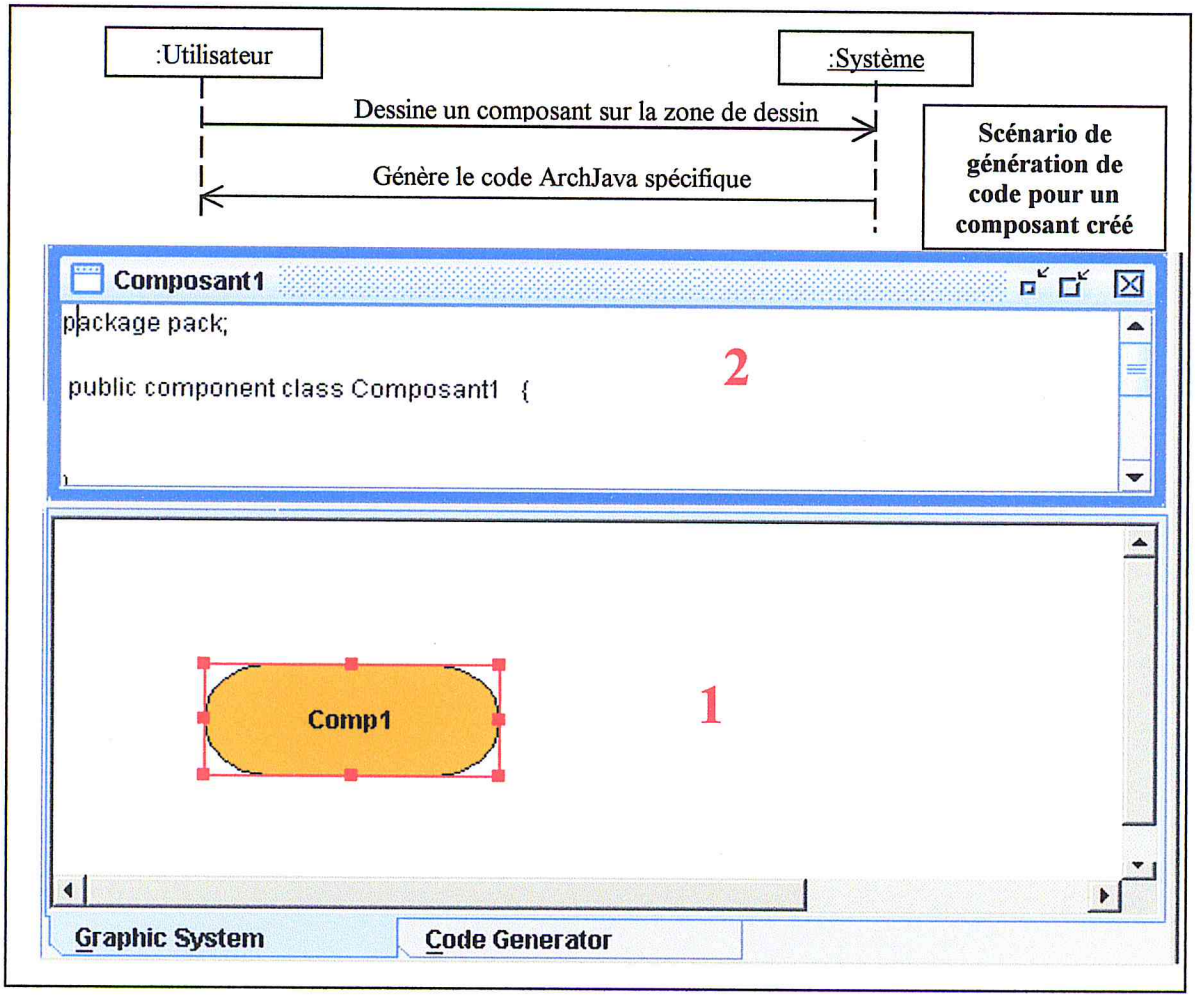

Figure 6.6: Validation de cas d'usage
"Génération de code ArchJava pour chaque composant crée "

- **Génération de code pour les ports des communications ajoutés aux composants:**

L'utilisateur ajoute un port de communication a un composant puis le système génère le code ArchJava spécifique a ce port après le système fait la mise a jour de code spécifique a ce composant.

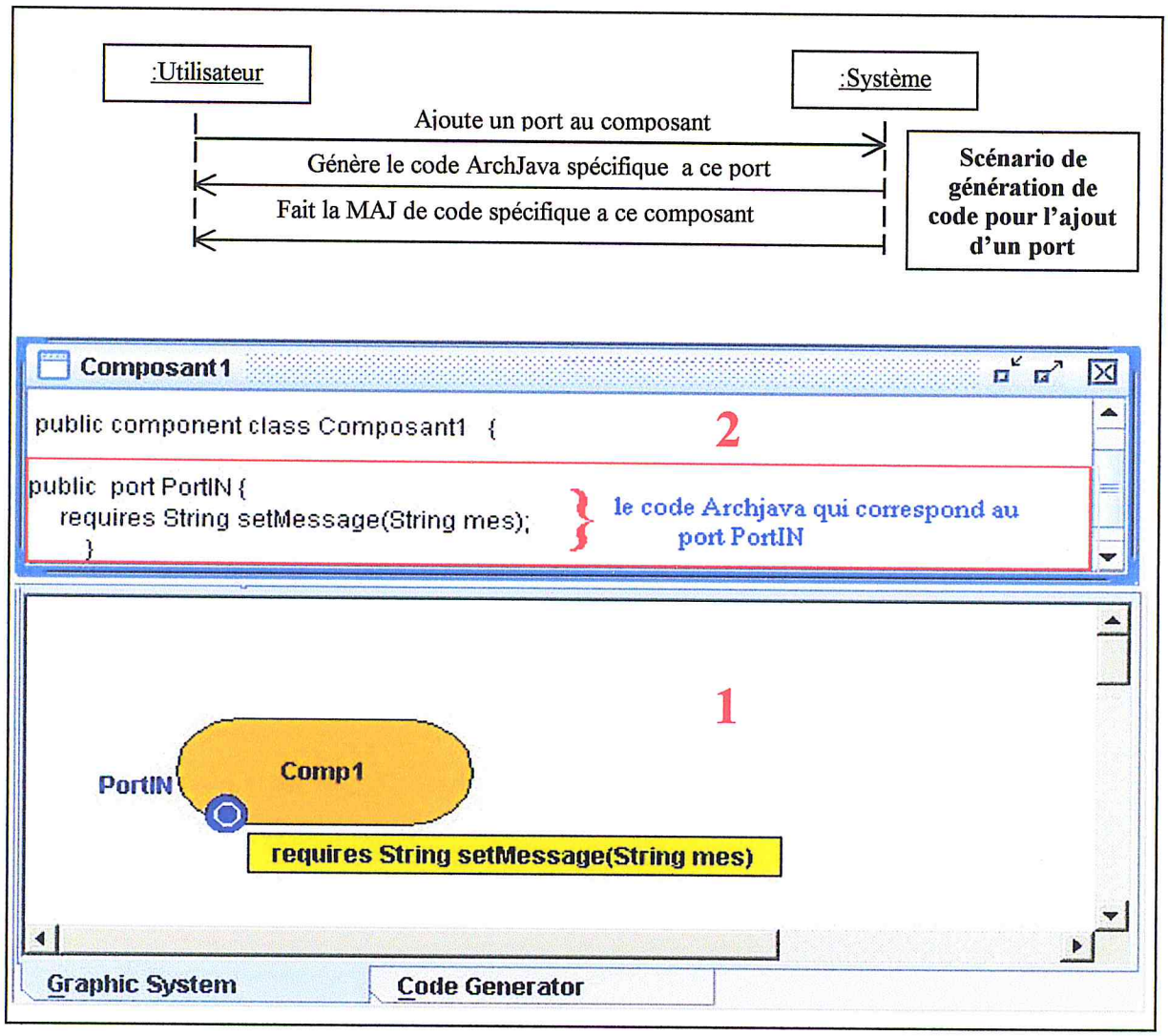

Figure 6.7: Validation de cas d'usage
" Générer le code ArchJava adéquat pour chaque port de communication "

- **Génération de code pour l'ensemble de l'architecture :**

La spécification de l'ensemble de l'architecture logicielle vas être encapsuler dans un composant principal qui englobe l'application, pour des raisons purement techniques de Archjava que nous avons déjà vu (voir Annexe A). Dans le cas de notre application nous avons nommé ce composant Main.archj et qui englobe toute l'architecture. Ce composant présente le code complet qui spécifier cette architecture en ADL Archjava voir figure **fig. 6.8** ci-après.

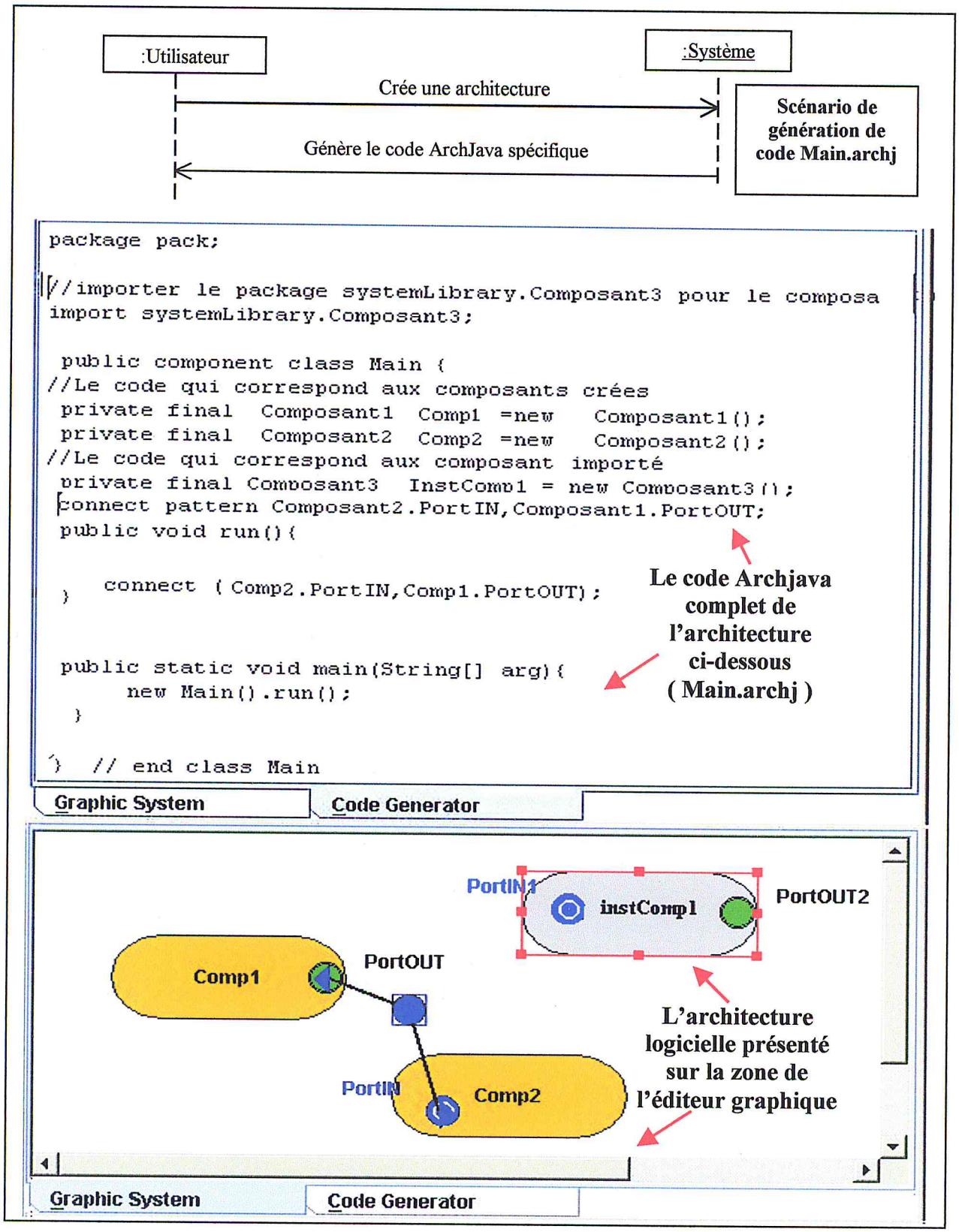

Figure 6.8: Validation de cas d'usage
" Générer le code ArchJava pour toute l'architecture "

VI.2.3- Validation de l'architecture spécifiée:

- **Compilation Archjava de code d'un composant :**

Pour compiler le code Archjava spécifique a chaque composant logicielle crée L'utilisateur clique sur le nœud représentant le code de composant qui veut compiler (le nœud de l'arbre dynamique), ensuite le système lance la compilation. A la fin de compilation le système récupère les résultats de compilation sur le terminal d'affichage de système comme montre la figure **fig. 6.9** ci-après.

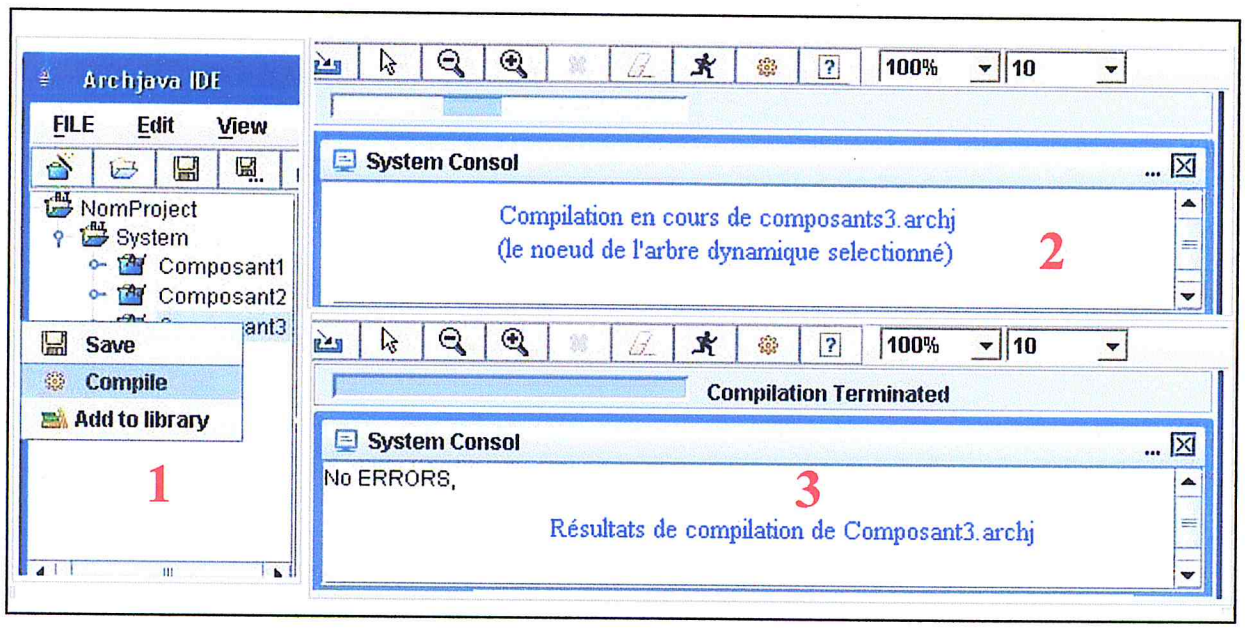

Figure 6.9: Validation de cas d'usage " Compilation de code composant "

N.B : Le composant **Main.archj** est le composant principal qui englobe toute l'architecture logicielle spécifiée, il contient des instances de chaque composant logiciel de cette architecture et les différentes interconnexions établies entres ces composants.

- **Compilation de code Main.archj**

Pour compiler le code Archjava spécifiant toute l'architecture, l'utilisateur demande d'abord au système de compiler le code ArchJava spécifique a son l'architecture (le fichier Main.archj) en cliquant tous simplement sur le bouton adéquat, ensuite le système lance la compilation de Main.archj. A la fin le système récupère les résultats de compilation sur le terminal d'affichage de système, voir figure **fig. 6.10** ci-après.

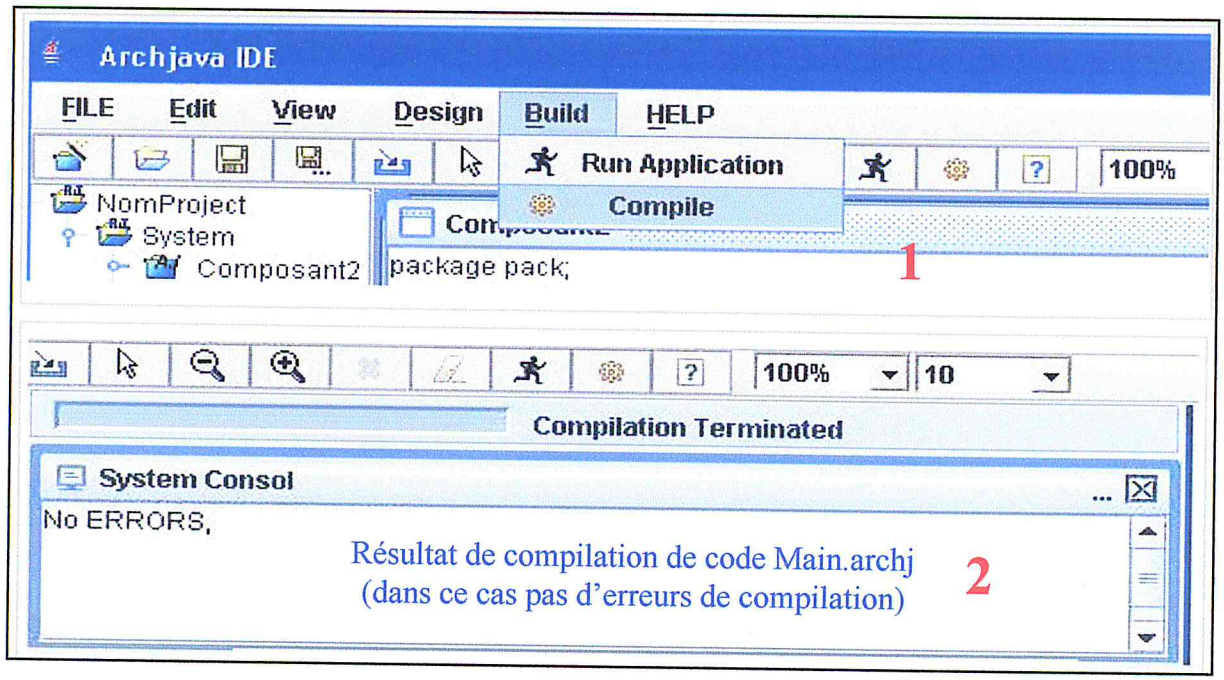

Figure 6.10: Validation de cas d'usage " Compilation de code Main.archj "

- **Exécution de code Main.archj**

Après la compilation de code Main.archj l'utilisateur demande d'exécuter le code ArchJava spécifique a l'architecture conçue, le système va donc lancer l'exécution. A la fin de l'exécution le système affiche les résultats sur le terminal d'affichage de système comme montre la figure **fig. 6.11** ci-dessous.

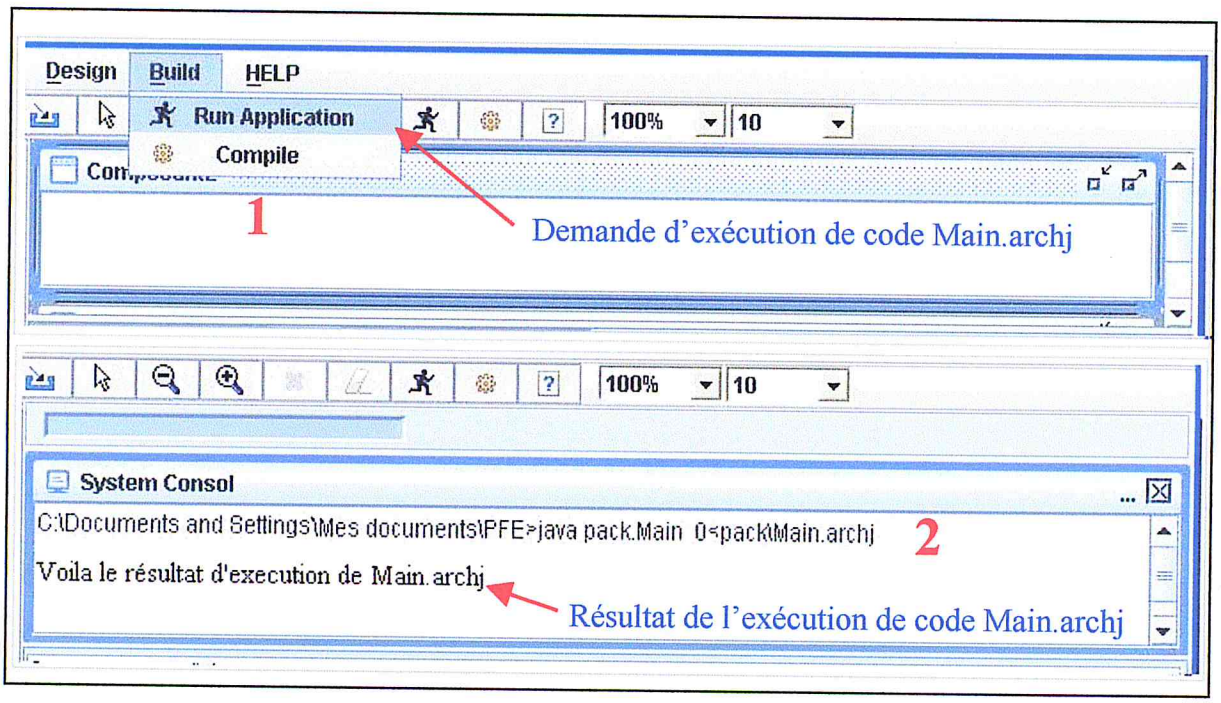

Figure 6.11: Validation de cas d'usage " Exécution de code Main.archj"

CHAPITRE VII :

Conclusions & Perspectives

L'architecture logicielle est un domaine récent de génie logicielle dont l'importance augmente jours après jours, elle est devenue un facteur critique dans le développement des projets logiciels car elle permet de gérer les difficultés que rencontrent les concepteurs de gros systèmes, et facilite aussi la maintenance et l'évolution des logiciels. On peut dire qu'elle offre une vue globale de l'application et la ronde plus maîtrisable a son concepteur.

L'architecture logicielle d'un système doit être spécifier de manière précise et commune, c'est pour cette raison que les ADL sont apparus. Les ADL permettent de spécifier l'architecture d'un système en terme de composants logiciels et définissent de manière explicite les interactions entre ces composants.

Le travail présenté dans ce mémoire consiste à la conception et la réalisation d'un environnement intégré dont l'objectif est de fournir un ensemble d'outils et de services afin de faciliter la spécification des architectures logicielles avec l'ADL *Archjava.*

L'environnement que nous avons développé offre plusieurs facilitées à ces utilisateurs que nous avions mentionnés précédemment. Nous pouvons dire qu'il est le premier IDE dédier à spécifier des architectures logicielles avec l'ADL Archjava. Il est doté d'un :

- Editeur graphique qui un outil de modélisation dont l'objectif est d'offrire aux utilisateurs de système la possibilité de spécifier chacun des éléments de son architecture de manière graphique c'est à dire en terme de plusieurs formes graphiques.

- Un générateur de code qu'est un outil pour générer de code Archjava qui doit correspondre à l'architecture modélisée sur la zone de dessin de l'éditeur graphique.

- D'une plate forme d'exécution dont l'objectif est d'unifier l'architecture avec l'implémentation en vérifiant que l'architecture conçue répond correctement aux contraintes architecturales.

Nous espérons que le travail réalisé sera un pas considérable vers un domaine très vaste et important dans le domaine de développement des systèmes logiciels et nous souhaitons que ce mémoire puisse servir à une référence aux étudiants aimons travailler dans cette voie.

Nous avons comme perspectives d'ajouter des versions améliorant et introduisant des nouvelles techniques de Archjava sachant que le système conçu ne touche pas tous les concepts offerts par l'ADL Archjava, il y'a d'autres concepts que nous avons pas traités pour des raisons strictement temporelles comme par exemple ajouter la possibilité de créer des composants dynamiquement.

A la fin on espère bien que ce modeste travail soit à la hauteur, et reflète bien les efforts Déployés tout au long une année afin d'aboutir à un travail honorable et avantageux.

ANNEXE A:

L'ADL ArchJava

Dans ce présent chapitre, nous allons décrire le langage de description d'architecture logicielle Archjava, présenter comment une architecture est désignée avec cet ADL, comment les différents éléments architecturaux sont spécifiés, les avantages et les inconvénients de cet ADL et on va conclure ce chapitre avec les domaines d'application de ce dernier.

1-Introduction :

L'architecture de logiciel décrit la structure d'un système, permettant une conception plus efficace, une compréhension de programme, et une analyse formelle. Cependant, les approches existantes découplent le code d'implémentation de l'architecture, violant les propriétés architecturales, et empêchant l'évolution de logiciel.

Archjava est une prolongation a java qui unifie l'architecture logicielle avec l'implémentation, en utilisant un système de type pour s'assurer que l'implémentation se conforme aux contraintes architecturales en obéissant principalement a une propriété d'uniformité appelée l'intégrité de communication[1].

Archjava est une amélioration apportée à java qui propose par le bais de l'ajout d'un ensemble de mot clés une solution aux problèmes que connues certains ADL, Archjava est

[1] On dit qu'un système a l'intégrité de communication si les composants implémentés communiquent seulement et directement avec les composants avec qui ils sont connectés dans l'architecture.

aussi un compilateur pour l'ADL Archjava permettant de générer de bytecode à destination de la JVM (Java Virtual Machine).

2-L'architecture logicielle dans l'ADL Archjava :

Cette section a pour but de décrire la conception de langage Archjava,
Chaque sous-section va introduire un concept de ce langage. Nous avons jouté des exemples pour décrire comment employer ces concepts pour spécifier des architectures logicielles.

Pour permettre à des programmeurs de décrire l'architecture de logiciel, ArchJava a ajouté des nouveaux concepts au langage pour supporter les éléments architecturaux qui sont les composants, connexions, ports.

2.1- Composants et ports :

Un composant est un objet spécial qui communique avec d'autres composants d'une manière structurée par le biais des ports[20] pour la réalisation d'une ou plusieurs fonctionnalité bien précise dans une architecture à un certain niveau d'abstraction.

Les composants sont des instances de la classe *component*, comme le composant Parser (Analyseur lexical en français) comme montre la figure **fig. 1** ci-dessous.

Exemple: Le composant Parser.

```
public component class Parser {
public port in {

        provides void setInfo(Token symbol, SymTabEntry e) { ... }
}

public port out {
        requires void compile(AST ast);
       provides SymTabEntry getInfo(Token t);
}


public SymTabEntry getInfo(Token t) { ... }
...
}
```

Figure 1 : Exemple d'un composant en Archjava

Les ports sont des canaux de communication logiques entre composants : aucun message ne peut passer entre deux composants s'il n'est issu d'un port et destiné à autre port. Un composant doit donc posséder des ports pour être inséré dans une architecture.

Les ports sont des objets internes aux composants, fournissant des types de messages entrants et sortants. Ces types de messages sont matérialisés par les méthodes des ports, ce sont des méthodes Java qui permettent donc de passer des objets Java ordinaires.

Les ports déclarent deux types de méthodes spécifiées par les mots clés **requires**, **provides**[21] .

Les méthodes **provided** ou fournies se sont des services fournis par le port, ils doivent être implémenter à l'intérieur de déclaration de port déclarant la méthode (comme la méthode **setInfo** de Parser dans l'exemple vu précédemment) ou a l'intérieur de composant déclarant le port (comme la méthode **getInfo** dans l'exemple ci-dessus)

Les méthodes **required** ou requises se sont les services que fournis un port (comme la méthode **compile** dans l'exemple ci-dessus), les méthodes required dans un port doivent être implémentées par les composants connectés a ce port.

N.B : Les classes de types composant peuvent contenir les déclarations des ports et connexions

2.2 -La Composition et les communications entre composants:

2.2.1- Les sous composants :

Un sous-composant ou (subcomponent en anglais) est une instance de la classe composant[22] qui est instantié à l'intérieur d'une autre classe composant

L'architecture d'Archjava permet les appels entre composants connectés (entre le composant Scanner et Parser dans la **fig.2**) et entre un parent et ses sous-composants immédiats. Elle interdit les appels externes à un sous composant et les appels entre composants non-connectés (entre le composant Scanner et Codegen dans la **fig.2**), elle interdit aussi les appels violant la structure hiérarchique de l'architecture.

Par exemple, Le composant compiler (voir **fig.2** ci-dessous) est le composant principal de l'application, il résulte de l'assemblage de plusieurs composants suivant le modèle du **Pipe-line,** qui sont les sous-composants scanner, parser et codegenerator.

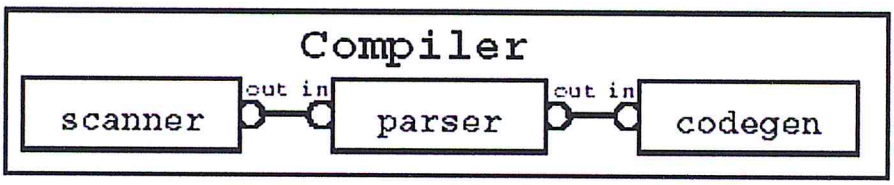

Figure 2: Exemple d'une architecture d'un compilateur simplifié

On a parlé précédemment des **connexions statiques** dont l'interconnexion des deux composants est faite après la compilation. On va maintenant parler d'un autre type de connexion qui est la **connexion dynamique** (voir exemple plus loin fig. 4 ci-dessous), ce type

de connexion se fasse dans l'appel de la méthode run( ) du composant principal qui englobe l'application. Elle est déclarée comme suit :

on va remplacer :

// La déclaration de la connexion statique
connect instComp1.port1, instComp2.port2 ;
Par :
// La déclaration de la connexion dynamique

1. **connect pattern Comp1.port1, Comp2.port2 ;**
2. Méthode (….){
   ….
   connect (instComp1.port1, instComp2.port2) ;
   …..
   }

2.2.2- Les connexions :

La description d'une architecture s'exprime en énonçant les communications entre les composants, et donc par les connexions entre ports.

ArchJava garantit que les communications entre composants respecteront les connexions établies à l'aide de la déclaration **connect** [20] [22] et qu'aucune autre connexion ne sera acceptée. La vérification de cette propriété est réalisée à la compilation. On parle alors d'une connexion statique.

L'architecture d'Archjava permet les appels entre composants connectés et entre un parent et ses sous-composants immédiats. Elle interdit les appels externes à un sous composant et les appels entre composants non-connectés. Elle interdit aussi les appels violant la structure hiérarchique de l'architecture (comme par exemple Compiler1.Scanner appelant Compiler2.Parser voir **fig. 2**).

Archjava permet au développeur de déclarer dans l'architecture leurs connexions qui sont réalisée à l'exécution. La déclaration "**connect** scanner.out, parser.in" dans la **fig.3** ci-dessous d'établir une connexion entre le port out de sous-composant scanner et le port in de sous-composant parser.Une fois les connexions ont été déclarées, une connexion concrète va être établie entre les composants, par exemple le constructeur de composant compiler connecte le port out de sous-composant scanner et le port in de sous-composant parser. Cette connexion lie la méthode fournie (provided) par le port out de scanner avec la méthode requise (required) par le port in de parser avec le même nom et signature.

```
public component class Compiler {

        private final Scanner scanner = ...;
        private final Parser parser = ...;
        private final CodeGen codegen = ...;

        connect scanner.out, parser.in;
        connect parser.out, codegen.in;

        public static void main(String args[]) {
        new Compiler().compile(args);
        }


        public void compile(String args[]) {
        ...parser.parse();...
}
}
```

Figure 3: La spécification avec Archjava de l'architecture de la figure 2

Depuis la structuration en composants jusqu'au traitement des accès aux données partagées entre ceux-ci, en passant par les types de connexions inter-composants, ArchJava permet de spécifier fiablement son application distribuée en facilitant sa maintenance et son évolution.

Exemple d'une connexion dynamique

```
public component class Compiler {

        private final Scanner scanner = ...;
        private final Parer parser = ...;
        private final CodeGen codegen = ...;
....

connect pattern Scanner.out, Parser.in;

        public void compile(String args[]) {
                connect (scanner.out, parser.in);
}
....
}
```

Figure 4: Exemple d'établissement d'une connexion dynamique entre composants

3-Quelles sont les avantages d'Archjava ?

L'ADL Archjava apporte beaucoup d'avantages par rapport aux ADL existants [19][21][22] qui se résument dans:

- Pour la raison d'intégrité de communication qu'assure l'ADL Archjava, le code source de programme reste conforme à l'architecture pendant que le programme évolue.
- L'architecture explicite d'un programme d'Archjava facilite les différentes interactions entre les différentes parties de code source, ce qui aide le concepteur à faire évoluer son logiciel plus pertinemment.
- Le contrôle de d'intégrité de communication d'Archjava aident à améliorer la structure d'un programme par l'élimination de violation d'encapsulation.
- Archjava permet de spécifier de façon fiable les application distribuées en facilitant sa maintenance et son évolution.
- Le code que le programmeur écrit en Archjava correspond directement à sa vision de l'architecture de l'application.
- Tout composant de l'architecture logicielle donne lieu a l'implémentation d'un composant en java.
- Archjava est facile à apprendre surtout pour quelqu'un connaissant déjà java puisque ce dernier est fondé sur celui-ci et qu'il ne lui ajoute que quelques mots clés très simples.
- Archjava permet la spécification de l'architecture logicielle dans le code java.

- Dans Archjava le code et l'architecture évoluent ensemble.

- Archjava vérifie le flux de contrôle de l'architecture excepté pour les casts comme dans java.
- Archjava est flexible car :
  - Il supporte dynamiquement le changement de l'architecture.
  - Il permet des techniques d'implémentation commune.
- Archjava garantie l'intégrité de communication car:
  - Un composant peut communiquer directement qu'avec le composant avec qui il est connecté dans l'architecture.
  - Un composant ne peut pas invoquer les méthodes d'un autre composant s'il n'est pas connecté à ce dernier.

- Archjava permet les objets d'être librement partagé entre les composants.

- Archjava enforce l'intégrité de communication par le flux de contrôle c'est à dire :
  - L'appel de méthode n'est permis d'un composant a l'autre excepté d'un parent à ses sous composants immédiats par des interconnexions dans l'architecture.
  - Archjava interdit :
    1. l'appel externe a des sous composants
    2. les appels entre sous composants non connectés
    3. Les appels qui violent la hiérarchie architecturale.

- Archjava assure un coût de développement réduit en diminuants les nombres de lignes de code.
- Archjava garde l'architecture et le code synchronisé et structure le système a haut nivaux e terme de composants, connecteurs. etc.

4- Quelles sont les limites d'*ArchJava* ?

Certaines limitation sont encore légèrement gênantes et devrant être levées pour assurer une plus grande efficacité et fiabilité pour cet outil dans le domaine d'architecture logicielle, on peut résumer ces limitations [19][20][22] dans :

- L'implémentation actuelle d'Archjava ne permet pas qu'un composant implémente une interface.
- Archjava reste entièrement compatible avec les librairies standard de java, il ne se dote pas de bibliothèques propres a ces composants.
- Les interconnexions inter composant doivent être mises en application par des méthodes (pas d'événements par exemple).
- Que de java.
- Trop concentrer sur l'intégrité de communication.
- Ne pas exprimer toutes les propriétés architecturales comme les données partagées (shared data), les protocoles temporels, quelques styles.

A la fin, nous pouvons dire que Archjava permet à des programmeurs d'exprimer facilement la structure architecturale et unifie cette structure a l'implémentation a chaque étape du cycle de vie de logiciel, ainsi aide les concepteurs à favoriser la conception efficace d'architecture, l'implémentation, la compréhension et l'évolution de programme malgré les quelques limites que comprenne cet ADL, sans oublier les travaux en cours de réalisation par les concepteurs de cet ADL pour éliminer ces limites.

Les informations concernant Archjava fournies dans ce document ne constituent qu'une petite partie de ce qu'offre cet ADL au domaine d'architecture logicielle, il y'a d'autres concepts que nous avons évités de les traiter dans ce document car il demande une explication approfondie comme le concept d'annotation, les classes de connexions, la régulation des accès aux données partagées… etc.

ANNEXE B:

Le langage UML

Ce document a pour but d'éclaircir certaine notion UML que nous avons utilisée pour modéliser et documenter la démarche de développement de système. UML est l'acronyme de Unified Modeling Language que l'on peut traduire par "langage de modélisation unifié ".

1. Introduction a UML :

UML[11] est une notation permettant de modéliser un problème de façon standard. Ce langage est né de la fusion de plusieurs méthodes existant auparavant, et est devenu désormais la référence en terme de modélisation objet

UML est un langage standard conçu pour l'écriture de plans de modélisation de logiciel, il peut être utiliser pour visualiser, spécifier, construire et documenter ce système. UML est adapté à la modélisation des systèmes depuis les systèmes informatiques d'entreprises jusqu'aux application distribuée basée sur le web en passant par les systèmes embarqués, c'est un langage très expressif qui couvre touts les perspectives nécessaires de développement de tels systèmes.

UML n'est qu'un langage et ne constitue qu'une partie d'une méthode de développement logiciel, UML est indépendant des processus de développement, est un formalisme et non une méthode.

2.Les briques de base d'UML :

La terminologie d'UML inclut trois sortes de briques qui sont : des éléments, des relations, des diagrammes. Les éléments sont des abstractions essentielles à un modèle, les relations constituent les liens entre ces éléments et les diagrammes les regroupent en des ensembles dignes d'intérêt.

2.1 -Les éléments :

Il existe quatre types d'éléments dans UML qui sont :

- Les éléments structurels
- Les éléments comportementaux
- Les éléments de regroupement
- Les éléments d'annotation

2.1.1- Les éléments structurels :

Ce sont les parties les plus conceptuels ou physiques, au total il existe sept types d'éléments structurels dont nous allons citer ceux que nous avons utilisés.

1) **Classe:** Représente un ensemble d'éléments qui partagent les même attributs, même opérations, même relations.

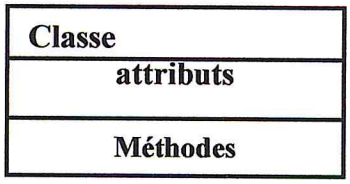

Figure 1 : Exemple d'une classe

2) **Cas d'utilisation:** Est la description d'une séquence d'actions exécutées par un système.

Figure 2 : Exemple d'un cas d'utilisation

3) **Composant:** Est une partie physique remplaçable d'un système, qui se conforme à un ensemble d'interfaces et en permet la réalisation.

Figure 3 : Exemple d'un composant

4) **Interface:** Est un ensemble d'opérations qui définissent la fonction d'une classe ou d'un composant, par conséquent une interface décrit le comportement apparent de cet élément.

Figure 4 : Exemple d'une Interface

2.1.2- Les éléments comportementaux : Représentent les parties dynamiques des modèles UML, ils représentent le comportement dans le temps et dans l'espace.

1) **L'interaction:** C'est un comportement qui comprend un ensemble de massage échangé au sein d'un groupe d'éléments.

Figure 5 : Message

2) **La généralisation:** Est une relation de spécialisation /généralisation selon laquelle les attributs de l'élément spécialisé (enfant) peuvent se substituer aux attributs de l'élément généralisé (parent), de cette manière l'enfant partage la structure et le comportement de parent.

Figure 6: La généralisation

2.1.3- Les éléments d'annotation: Ils représentent les parties explicatives des modèles UML, parmi les il existe l'élément:

1) **Note :** Est simplement un symbole utilisé pour représenter les contraintes et les commentaires rattachés à un élément ou ensemble d'éléments. Ils sont utilisés pour décorer les diagrammes des contraintes ou des commentaires.

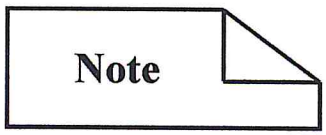

Figure 7: Une note

2.1.4- Les éléments de regroupement: Ils représentent les parties organisationnelles des modèles UML, les paquetages qui permettent de regrouper des éléments

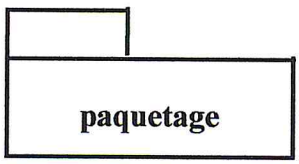

Figure 8: Un paquetage

2.2 -Les relations dans UML:

1) **dépendance :**Est une relation entre deux éléments selon laquelle un changement apporté a l'un (élément source) peut affecter la sémantique de l'autre élément (élément cible).

Figure 9: la dépendance

2) **association :**Est une relation structurelle qui décrit un ensemble de liens (relation entre objets)

2.3-Les diagrammes dans UML :

Un diagramme est la représentation graphique d'un ensemble d'élément qui constitue un système, la plupart du temps il se présente sous la forme d'un graphe connexe où le sommets et les arcs aux relations [11].

UML comprend nefs diagrammes comme montre la figure **fig.10** ci-dessous qui sont:

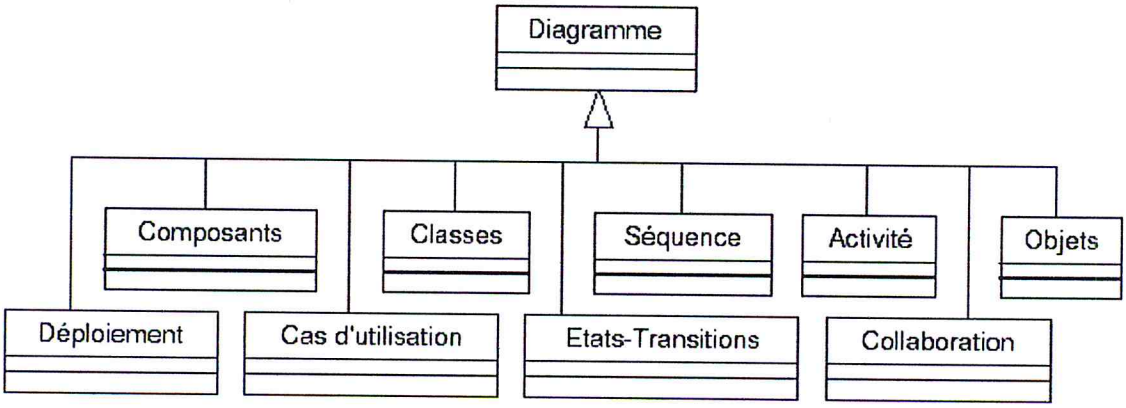

Figure 10 : Les nefs diagrammes d'UML

- Les diagrammes de classes
- Les diagrammes d'objets
- Les diagrammes de cas d'utilisation
- Les diagrammes de séquence
- Les diagrammes de collaborations
- Les diagrammes d'états transition
- Les diagrammes d'activités
- Les diagrammes de composants
- Les diagrammes de déploiement

Parmi ces diagrammes, nous allons citer que ceux que nous avons utilisé pour modéliser la démarche de développement de notre système qui sont les diagrammes de classe, diagrammes de cas d'utilisation, diagrammes d'activités, diagrammes de séquence et le diagramme de composants.

2.3.1- Les diagrammes de classe:

Il représente un ensemble de classes, d'interfaces ainsi leur relations. Ce sont les diagrammes les plus fréquents dans la modélisation des systèmes orientés objet, ils représentent la vue de conception statique d'un système.

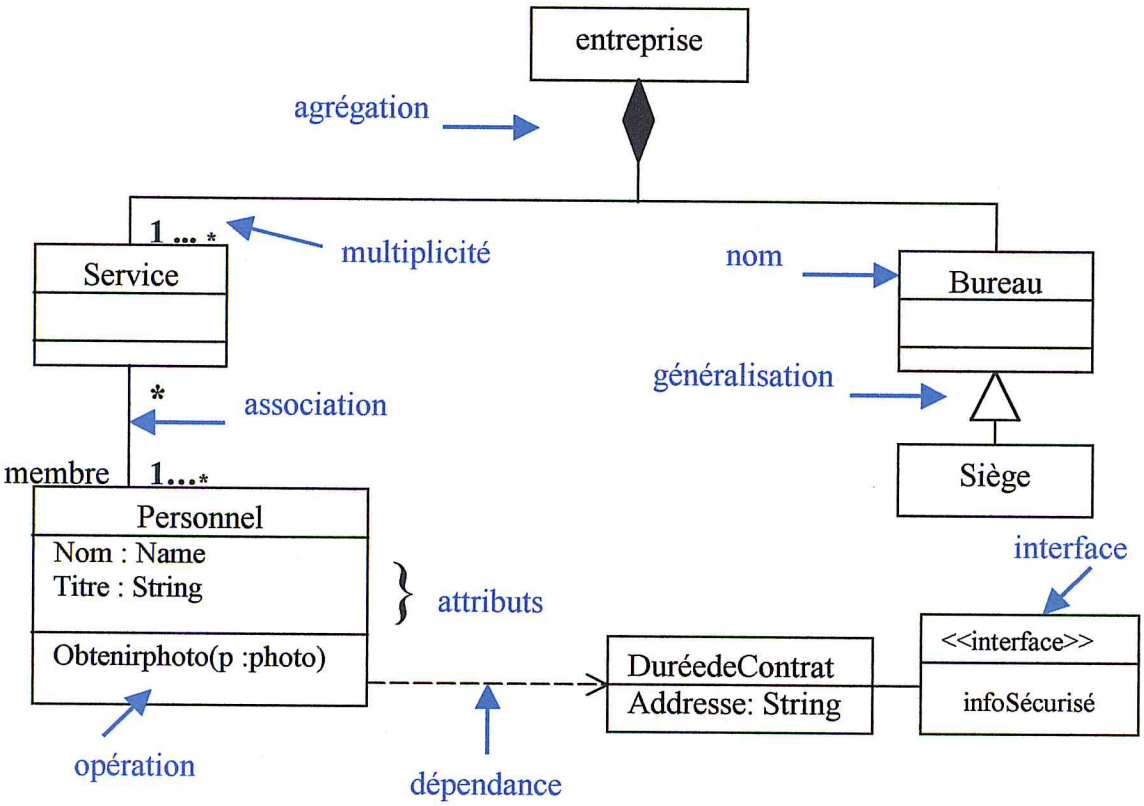

Figure11 : Exemple de diagramme de classe

3-Conclusion :

En résumé, UML permet de documenter l'architecture d'un système dans les moindres détails, fournit un langage pour exprimer les besoins et effectuer des tests. Il faut retenir aussi qu'UML fournit un moyen visuel standard pour spécifier, concevoir et documenter les applications orientées objets. En fin de compte, l'intérêt de la normalisation d'un langage de modélisation tel que UML réside dans sa stabilité et son indépendance vis-à-vis de tout fournisseur d'outil logiciel.

2.3.4- Les diagrammes d'activités:

Ce sont des types particuliers de diagrammes d'états-transitions qui décrivent la succession des activités au sein d'un système. Ils représentent la vue dynamique d'un système, ils sont particulièrement importants dans la modélisation de la fonction d'un système et mettent l'accent sur le flot de contrôle entre les objets.

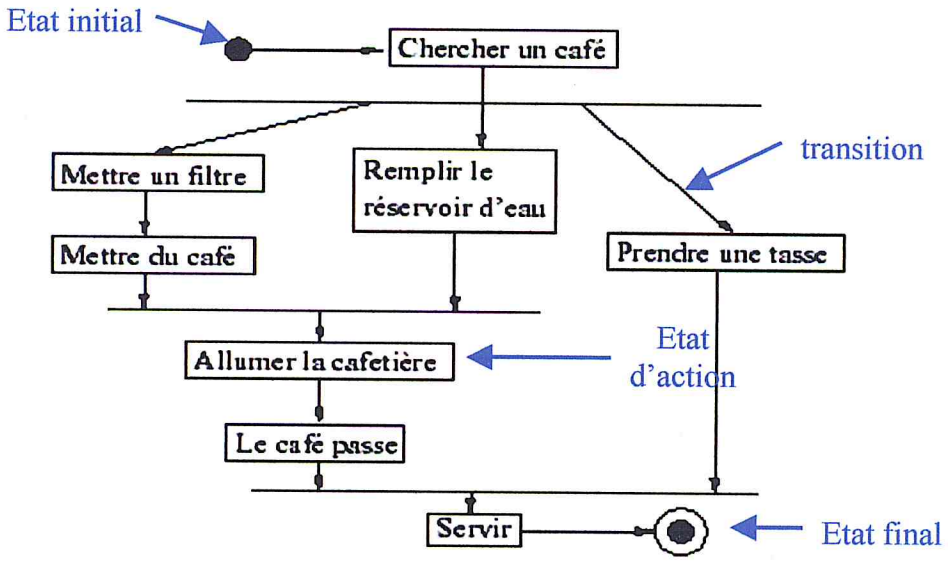

Figure14 : Exemple de diagramme d'activité

2.3.5- Les diagrammes de composant:

Ils représentent l'organisation et les dépendances au sein d'un ensemble de composants, ils représentent la vue d'implémentation statique d'un système.

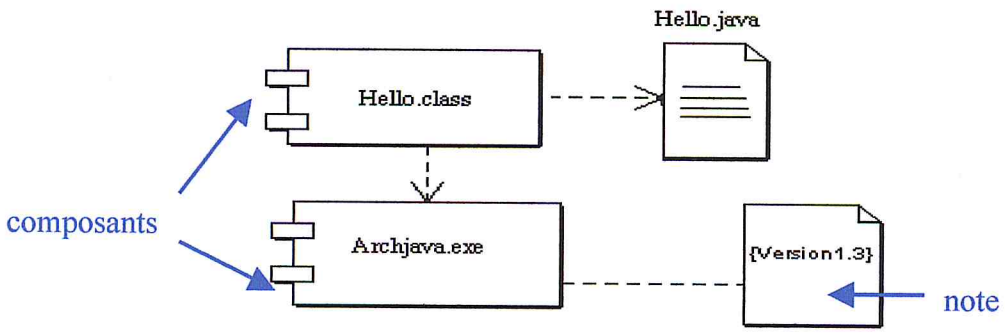

Figure15 : Exemple de diagramme de composants

2.3.2- Les diagrammes de cas d'utilisation :

Ils représentent un ensemble de cas d'utilisation et d'acteurs et leurs relations, ils modélisent le comportement d'un système ainsi qu'ils représentent les fonctions du système de point de vue d'utilisateur.

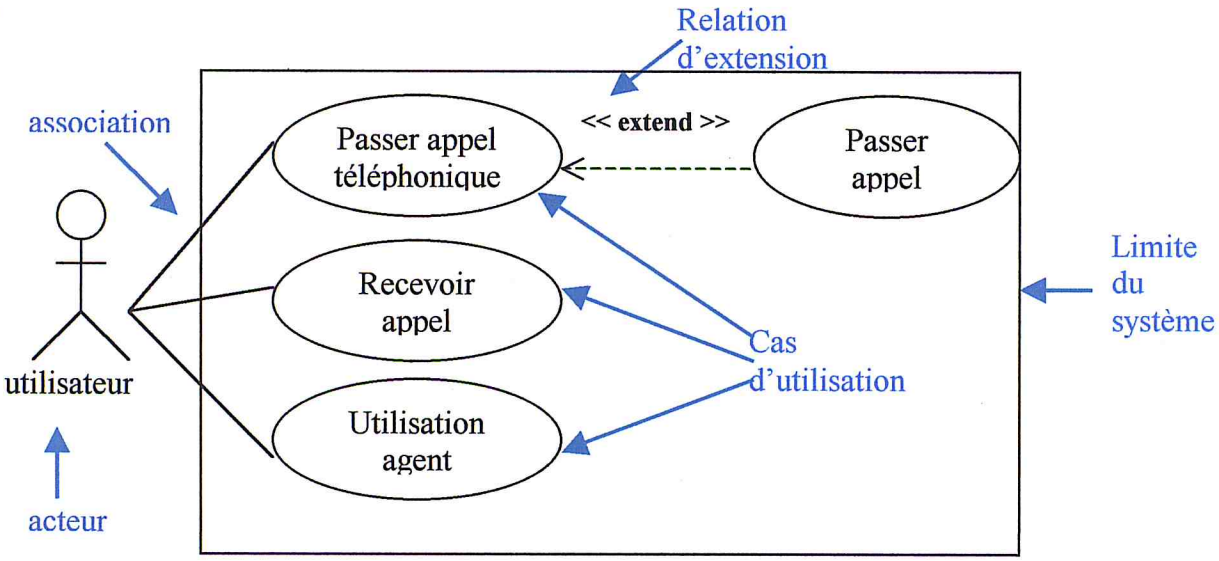

Figure12 : Exemple de diagramme de cas d'utilisation

2.3.3- Les diagrammes de séquence:

Ils sont des diagrammes d'interaction qui mettent l'accent sur le classement chronologique des messages, ils représentent la vue dynamique d'un système.

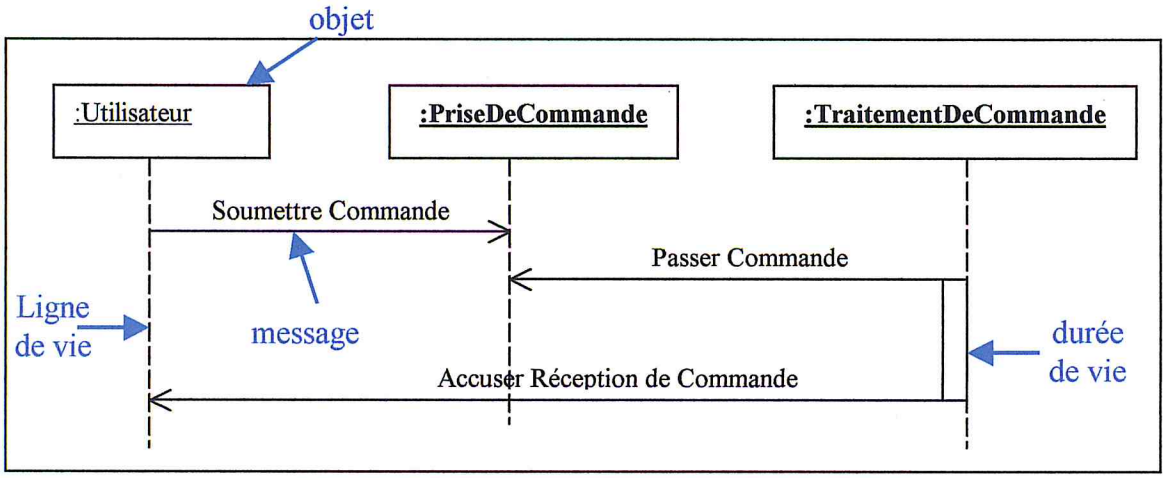

Figure13 : Exemple de diagramme de séquence

La Bibliographie:

[1] 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.

[2] D. Garlan: Software Architecture, Encyclopedia of Software Engineering, John Willey & Son, 2001.

[3] A.Abdurazik, Suitability of the UML as an Architecture Description Language with Applications to Testing, ISE-TR-00-01; Information and Software Engineering, Université de George Mason, février 2000.

[4] 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.

[5] N.Kanaoui, T.Khammaci, M.Oussalah, Les ADLs : une voie prometteuse pour les architectures logicielles, Rapport interne IRIN, Université de Nantes, 2003.

[6] D.Bennouar, Vers la définition d'un modèle de connecteurs logiciels pour la description de systèmes complexes, Université Saad Dahlab, Blida, Algérie, décembre 2004.

[7] D.Bennouar, Vers une mise en œuvre conjointe des ADL ET UML dans le développement de logiciels complexes, Université Saad Dahlab, Blida, Algérie, décembre 2003.

[8] E.Durand, A. M. Deplanche, Y.Trinquet, Langage de configuration, mars 2000.

[9] T.Khammaci, A. Smeda, M. Oussalah, Operational Mechanism for Object Component Architecture, Rapport IRIN, Université de Nantes, France.

[10] J. PRINTZ, Le génie Logiciel, Université de France, mai 2000.

[11] G.Booch, J.Rumbaugh, I.Jacobson, Le guide de l'utilisateur UML, Groupe Eyrolles, Paris 2003.
[12] V. Berthié, J.-B. Briaud, Swing La Synthèse, Développement des interfaces graphices en Java, Dunod, Paris, avril 2003

[13] A.Mirecourt, P.-Y. Saumont, Référence java2, Edition Osman Eyrolles Multimédia, 2003

[14] N. Medvidovic, D.S.Rosenblum, Assessing the suitability of a Standard Design Method for Modeling Software Architectures, Proceeding of the First IFIP Working Conference on Software Architecture, San Antonio, TX, 1999.

[15] D. Garlan, A. J. Kompanek, Reconciling the Needs of Architectural Description With Object Modeling Notation, Proceeding of the Third International Conference on the Unified Modeling Language, Oct. 2000, York, UK.

[16] J. E. Robbin, D. F. Redmiles, D. S. Rosenblum: Integrating C2 with the Unified Modeling Language, Proceedings of the 1997 California Software Symposium, Irvine CA, November 1997.

[17] C. Hofmeister, R.L. Nord, D.Soni: Describing Software Architecture with UML, Proceeding of the First IFIP Working Conference on Software Architecture, San Antonio, TX, 1999.

[18] J. Aldrich, C.Chambers, D.Notkin,Archjava: Connecting Software Architecture to Implementation, Department of Computer Science and Engineering, University of Washington,
ICSE '02 May 2002 Orlando, FL, USA

[19] J. Aldrich, C.Chambers, D.Notkin,Architectural reasoning in Archjava , Department of Computer Science and Engineering , University of Washington,
ICSE '02, Orlando, FL, USA, juin 2003.

[20] J. Aldrich, C.Chambers, V.Kostadinov,Alias Annotations for program understanding, Department of Computer Science and Engineering , University of Washington,
OOPSLA'02, Seattle, Washington, USA, novembre 2002.

[21] J. Aldrich, C.Chambers, V.Kostadinov,Archjava connecting software architecture to implémentation, Department of Computer Science and Engineering , University of Washington, ICSE'02, Seattle, Washington, USA, mai 2002,

[22] D. Bennouar, Intégrer l'architecture d'une application avec Archjava, Université Saad Dahlab, Blida, Algérie, décembre 2003.