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 : Systèmes D'Information.

Sujet :

Conception et réalisation d'un mini éditeur pour UML

Présenté par : Djaouane Nesr Eddine.
Hamlaoui Ibrahim.

Promoteur : Boukhelef Djeloul

Organisme d'accueil :

Soutenue le: 27/12/2006, devant le jury composé de :

- 2005/2006-

بسم الله الرحمن الرحيم

إن الحمد لله ؛ نحمده ونستعينه ونستغفره , ونعوذ بالله من شرور أنفسنا وسيئات أعمالنا , من يهده الله فلا مضل له , ومن يضلل فلا هادي له .

واشهد ان لا اله الا الله – وحدة لا شريك له –.

واشهد ان محمدا عبده ورسوله .

(يا ايها الذين امنوا اتقوا الله حق نقاته ولا تموتن الا وانتم مسلمون).

(يا ايها الناس اتقوا ربكم الذي خلقكم من نفس واحدة وخلق منها زوجها وبث منهما رجالا كثيرا ونساء واتقوا الله الذي تساءلون به والارحام ان الله كان عليكم رقيبا).

(يا ايها الذين امنوا اتقوا الله وقولوا قولا سديدا . يصلح لكم اعمالكم ويغفر لكم ذنوبكم ومن يطلع الله ورسوله فقد فاز فوزا عظيما).

أما بعد:

Remerciement

Nous tenons à remercier tous les gens qui nous ont aidé à réaliser ce travail de prés et de loin.

Nous remercions notre promoteur M. BOUKHELEF pour son aide, sa patience et ses conseils.

Nous remercions tout particulièrement nos chers parents pour leurs patiences, encouragement et leurs soutiens précieux tout au long de notre carrière d'étudiants.

Merci a tous ceux qui de prés ou de loin nous ont aidés et soutenus pour que ce travail soit réalisé.

TABLE DES MATIERES

TABLE DES FIGURES

CHAPITRE I

Introduction générale

UML (Unified Modeling Language) est un langage graphique de modélisation objet résultat des tentative d'unification des méthodes **OOD** (Object Oriented Design) de Booch, **OMT** (Object Modeling Technique) de Rumbaugh et **OOSE** (Object Oriented Software Engineering) de Jacobson.

Les tentatives d'unification de ces méthodologies ont permis la construction de plusieurs versions de la méthode unifiée (version 0.8 de 95 et version 0.91 de 96) sans qu'il y ait eu véritablement consensus.

Le rapprochement en premier d' **OOD** et d' **OMT** à permis de :

a) Garder la puissance d'expression de la sémantique d' **OMT** à travers les concepts d'association entre objets, et l'expression de leurs comportements à travers la notion de diagramme d'état/transition.

b) Sélectionner d' **OOD** les concepts de modules (sous systèmes) et la notion de flots de message.

Avec ces caractéristiques ; les étapes d'analyses (**OMT**) et de construction (**OOD**) étaient prisent en considération dans le cycle de vie du SI.

Le rapprochement en second lieu avec **OOSE** a permis d'utiliser les Use Case (cas d'utilisation) comme moyen d'expression des besoins des utilisateurs dans la phase d'analyse.

UML est avant tout un support de communication performant, qui facilite la représentation et la compréhension de solutions objets :

- Sa notation graphique permis d'exprimer visuellement une solution objet, ce qui facilite la comparaison et l'évaluation des solutions objets.
- L'aspect formel de sa notation, limite les ambiguïtés et incompréhensions.
- Son indépendance par rapport aux langages de programmation et aux domaines d'application, en fait un langage universel.

La notation graphique d'**UML** n'est que le support du langage, la véritable force d'**UML** c'est qu'il repose sur un méta modèle. La puissance et l'intérêt d'**UML**, est qu'il normalise la sémantique des concepts qu'il véhicule.

UML est bien donc plus qu'un simple outil qui permet de dessiner des représentations mentales, il permet de parler un langage commun, normalisé mais accessible, car visuel. Il représente un juste milieu entre langage mathématique et naturel, pas trop complexe mais suffisamment rigoureux, car encore basé sur un méta modèle.

Enfin ; l'absence de consensus sur une méthode d'analyse objet a longtemps freiné l'essor des technologies objet, ce qui a mené à la demande de générer un ou plusieurs outils graphiques qui permettent de visualiser les modèles objets, c'est la première raison qui nous à mener à réaliser notre travail.

Une deuxième raison est qu'il existe une panoplie d'éditeurs de diagrammes **UML** plus ou moins complets, mais on s'intéresse aux logiciels libres pour les raisons suivantes :

- Il est possible de voir le code et de le modifier pour ajouter d'autres fonctionnalités aux fonctions principales du logiciel.
- Notre travail s'annonce comme un point de départ d'un grand projet qui s'articule sur le langage **UML** pour arriver a réaliser un éditeur de diagrammes **UML** dédier aux systèmes distribués (les éditeurs **UML** généralistes trouvent des difficultés de modélisation si on les utilise dans un domaine spécialisé comme les systèmes distribués car ils partent d'une vue globale).

"Parmi les logiciels libres on peut trouver les deux plus aboutis: DIA[1] et ArgoUml[2] .
Le premier souffre du fait d'être un éditeur de diagrammes généraliste, ne faisant pas que des diagrammes UML, le second est parti d'un point de vu trop éloigné des besoins concrets de la modélisation **UML**[5].

Donc, le but de notre travail est de réaliser un outil graphique qui permet aux utilisateurs de modéliser les trois diagrammes (diagramme de classe, cas d'utilisation et de séquence) et de donner aux futurs ingénieurs un point de départ et une plateforme pour continue ce grand projet.

Afin de réaliser ce but, on à organiser notre travail en quatre parties :

Nous allons commencé par le chapitre I qui constitue une introduction générale.
Dans la deuxième partie nous allons décrire dans le chapitre II les concepts généraux qui seront le concept objet et la modélisation objet avec **UML** et dans le chapitre III le processus de développement utilisé.

La troisième partie est constituée des chapitres IV, V, VI et VII qui seront consacrés au processus de développement en commençant par une analyse des besoins dans le chapitre IV, puis la conception dans le chapitre V, l'implémentation dans le chapitre VI et on conclut le processus par un chapitre de test et validation qui sera le chapitre VII.

Dans la quatrième partie qui sera le chapitre VIII nous allons conclure notre travail en rappelant son intérêt et son apport vis-à-vis le langage de modélisation **UML**, ainsi que nous proposons quelques perspectives afin d'enrichir et améliorer ce travail et aider le grand projet a atteindre ses buts.

[1] DIA: A diagram editor. http://dia.sourceforge.net
[2] ArgoUML UML editor. http://argouml.tigris.org

CHAPITRE II

Concepts généraux

SECTION I : LE CONCEPT OBJET

1. Les Problèmes Actuels du Génie Logiciel :

1.1. Introduction :

Les systèmes informatiques d'aujourd'hui sont au cœur des entreprises, ils doivent être a l'image de l'activité de ces entreprises : réactifs, évolutifs, performants.

Or l'industrie informatique est récente comparée a l'industrie métallurgique ou l'industrie de bâtiment. Sa place prépondérante au sein de l'activité économique la contraint a faire preuve de maturité, prise entre les exigences grandissantes des utilisateurs et l'évolution fulgurante des technologies électroniques (processeur, mémoires) sur lesquelles elle repose. Cette maturité doit s'acquérir en adoptant une attitude digne d'un processeur industriel et en mettant un terme aux techniques artisanales du monde informatique. Ainsi dans ce chapitre on va montrer comment la technologie objet et les méthodes objet permettent d'atténuer sinon de résoudre ces problèmes.

1.2. La taille et la complexité d'un logiciel :

- **La complexité fonctionnelle :** parce que le logiciel occupe une position de plus en plus centrale au sein de l'entreprise, il offre de plus en plus de fonctionnalités. Si nous prenons le cas d'un système d'information il doit non seulement être capable de stocker et d'organiser un grand nombre de données, mais également de les traiter de façon intelligente pour aider a la prise de décision (***datawerehouse***).

Ces informations doivent être reparties et mise à disposition des acteurs de l'entreprise selon leur niveau de confidentialité et les droits d'accès de chacun.

- ***La* complexité architecturale:** de plus en plus, les systèmes informatiques utilisent des machines distantes, hétérogènes qui ont des rôles bien précis (client, serveurs de données, serveur d'application,... etc.). Avec les architectures client/serveur, INTRANET, CORBA se posent de nombreux problèmes, pour certains mal connus :
  - Comment distribuer un système ?
  - Comment le système réagit-il a l'augmentation de la charge ?
  - Quelle technologie adopter ?

1.3. La taille croissante des équipes:

- **Des compétences de plus en plus variées:** la fabrication d'un logiciel demande des compétences de plus en plus variées et de plus en plus pointues. En outre, plus les systèmes a développer sont importants, plus la taille des équipes augmente. Ceci pose un certain nombre de problèmes :
  - Gérer les différentes compétences techniques.
  - Coordonner les travaux afin d'assurer a chacun une tache a tout moment.
  - Faire circuler l'information au sein des groupes de travail, autrement dit faire communiquer des gens qui n'ont pas les mêmes compétences ni le même vocabulaire.
  - Gérer le travail en parallèle sur une même tache.

- **Des délais de plus en plus courts :** la taille des équipes augmente également parce que les délais qui sont impartis pour la réalisation d'un système sont de plus en plus courts et ceci pour plusieurs problèmes :
  - L'importance grandissante des applications informatiques dans l'entreprise. Les entreprises ne peuvent attendre longtemps un système qui leur permettra de gagner en productivité ou de développer de nouvelles activités.
  - Les concurrences qui existent entre les sociétés de services en informatique.

1.4. L'évolution rapide des applications :

- **Evolution fonctionnelle et technique :** une fois les spécifications écrites, la réalisation du projet peut commencer. Les choses seraient relativement simples si le travail a réaliser était défini une fois pour toute. Le problème est qu'il peut survenir des changements en court du projet. Ceci est compréhensible Pour des projets s'étalant sur plusieurs mois voire plusieurs années. Ces changements peuvent porter sur plusieurs points.

- **Modification des besoins du client:** au fur et a mesure que le projet avance, on peut se rendre compte des insuffisances, des chois inadaptés qui ont été faits lors des spécifications et de l'analyse. II faudra donc compléter voir modifier le projet pour le recadrer.

- **Modification de l'activité du client :** le client peut aussi vouloir modifier l'orientation fonctionnelle du système, d'autant plus facilement que la spécification aura été floue.

- **Modification de l'environnement technique : dans des longs projets, on peut être amenés a changer de choix techniques, soit parce que l'on se rend compte que ceux-ci sont inadaptés, soit parce que la pérennité du système est remise en cause, soit encore parce qu'un standard de fait s'installe parallèlement au choix qui a été fait. Cette constante évolution constitue un gène permanent. Elle empêche une bonne maturité des technologies, une analyse en profondeur des problèmes.**

2. Origine de l'approche objet : [Bouzghoub 98][1]

L'approche objet est caractérisée par une notion centrale, celle d'objet. Un objet

associe données et traitements dans une même entité en laissant visible que l'interface de l'objet, c'est adire les opérations que l'on peut effectuer dessus. L'approche objet est déjà ancienne, puisqu'elle est née avec le langage SIMULA, l'objectif étant de réaliser un langage de programmation structuré permettant de simuler des processus parallèles. L'aspect abstraction de l'approche objet qui permet de cacher une structure de données par des opérations manipulant cette structure a été formalisé dans les années 70 avec la théorie des **types abstraits**.

L'idée fondamentale était de définir un ensemble de fonctions (par exemple : empiler, dépiler) manipulant un ensemble de sortes (par exemple: des piles et des entiers) et de spécifier formellement ces fonctions et leur relations par des axiomes, sans se soucier de l'implémentation qui pouvait venir plus tard.

En parallèle au formalisme des types abstraits s'est développe le langage Smalltalk qui a implémenté la notion de type abstrait sous forme de classes comme Simula, mais qui a aussi apporte l'envoie de message emprunté au concept d'acteur et la structuration des classes en **hiérarchie de généralisation** avec **héritage**. Smalltalk est la source essentielle de l'inspiration de l'approche objet. Au-delà de Smalltalk, les relations de généralisation avec héritage ont été bien développées en intelligence artificielle dans la représentation des connaissances, plus particulièrement dans le contexte de ***frames*** et des réseaux sémantiques. Les objets peuvent ainsi être relies par des relations de type «est un».

En résumé, trois points de vues ont conduit à la notion d'objet :

- Le point de vue structurel : ou l'objet est perçu comme une instance d'un type de données caractérisé par une structure cachée par des opérations.
- Le point de vue conceptuel : ou l'objet correspond a un concept du monde réel qui peut être spécialisé.
- Le point de vue acteur : ou l'objet est une entité autonome et active qui répond a des messages.

De ces trois points de vues sont nés plusieurs variantes de l'approche objet qui ont influence la plupart des domaines de l'informatique, plus particulièrement

les langages de programmation, les interfaces homme-machine, les systèmes experts, les bases de données et les méthodologies de conception.

Autrement dit, l'approche objet a des origines multiples, plus particulièrement la simulation, la communication homme-machine et l'intelligence artificielle. L'objet vu comme acteur autonome permet de simuler des processus parallèles. L'objet vu comme concept permet la modélisation de connaissances. L'objet vu comme une boite noire permet de dialoguer par icône et souri. En résultat, l'approche objet a une large diversité de mises en œuvre. Il s'agit plus particulièrement d'une nouvelle approche de la programmation et la conception afin de mieux maîtriser la complexité.

3. pourquoi les objets ? [Bouzghoub 98][1]

La conception d'objet est sans doute le premier domaine ou les objets facilitent le travail. Ils permettent en effet de représenter les entités du monde réel et les relations entre ces entités par des graphes disposant de nœuds représentant les classes et d'arcs représentant les liens de généralisation, d'agrégation ou d'associations entre entités, il est possible de visualiser un modèle de l'application. Plusieurs graphismes appropries et souvent clairs ont été proposes pour cela. Une méthodologie d'objet permet même de visualiser les services offerts par les différentes entités. Elle permet aussi de constituer des groupes de classes et de développer des exemples d'objets.

La programmation des objets se distingue de la programmation classique par sa vision des objets comme des entités actives qui exécutent un comportement en réponse à un message. Au lieu de structurer d'un coté les variables contenant les données et de l'autre les fonctions constituent les traitements, on organise les programmes en entités actives composées de structures de données cachées par des fonctions. Un même nom de fonction peut être utilise pour effectuer des actions similaires sur des objets différents, ce qui permet de constituer un langage abstrait permettant d'agir de manière similaire sur des objets a priori différents. De manière plus globale, dans l'approche objet, un programme est un ensemble d'objets échangeant des messages qui déclenchent des opérations faisant évoluer leurs états internes et retournant des paramètres. L'utilisateur apparaît lui-même comme un objet qui dialogue avec les autres objets au biais des icônes, d'une souris, de menus déroulent et des boites de

dialogue.

4. Les concepts objet : [Bouzghoub 98][1],[Lai 99][2]

En programmation : dans la plupart des langages traditionnels, c'est a dire non orientés objet, les données et les traitements sont séparés. Les traitements peuvent donc n'avoir qu'une vue limitée des données sur lesquelles ils s'appliquent.

Le langage C, par exemple, est un langage structure qui privilégie l'efficacité et la rapidité des traitements, laissant au programmeur l'entière responsabilité de cohérence des appels de fonctions. Cela est source de nombreux problèmes. D'une manière générale l'exécution d'une fonction qui n'a pas les moyens de vérifier la cohérence de ces paramètres et a laquelle on fournit des paramètres incorrects, laisse le programme s'exécute jusqu'A ce que les cohérences de l'erreur entrain une autre anomalie qui sera bloquante. La difficulté pour le programmeur provient alors du fait que cette anomalie bloquante n'a souvent pas un rapport direct avec les paramètres incorrects.

L'erreur sera très difficile à expliquer et donc à corriger.

En méthodologie l'approche systématique est la plus utilisée (MERISE, largement répondue, s'appuie sur cette approche) ; elle repose principalement sur deux types de modèles pour décrire une application :

- **Le modèle de données :** qui consiste a établir un inventaire des données de !'Application et a définir leur structure et leur relations.
- **Le modèle de traitements :** qui permet d'avoir un aperçu de la dynamique de l'application, par une représentation du séquencement des traitements et la définition des algorithmes.

Les données et les traitements sont complètement séparés, ce qui entraîne une faible cohérence entre les deux modèles. La difficulté consiste alors à comprendre la sémantique des données sans avoir étudie leurs comportements ; de plus, il est difficile de définir un traitement sans connaître la structure des données et les relations entre ces données.

4.1. Les objets :

Les concepts d'objet et de classe sont indépendants. En effet, un objet appartient à une classe et une classe décrit la structure et le comportement communs d'objets. Nous définissons d'abord le concept d'objet :

4.1.1. Définition :

Un objet est une abstraction d'une donnée du monde réel caractérisée ainsi :

Objet=identité +état + comportement.

L'identité est représentée par un identifiant unique et invariant qui permet de référencer l'objet indépendamment des autres objets. L'état d'un objet est une valeur qui peut être simple, par exemple, une littérale, ou structurée, par exemple, une liste. Le comportement d'un objet est défini par l'ensemble des opérations applicables à l'objet et définies dans sa classe d'appartenance.

4.1.2. Liens entre objet :

Grâce à l'identité d'objet, les liens entre objets peuvent être représentés directement par des références. De même que les objets sont des instances de classes, les liens sont des instances d'associations entre classes. Deux types de liens sont a considérer : un lien mono value entre un objet (source) et un seul autre objet (cible), et le lien multi value entre un objet source et un ensemble d'objets cibles.

4.2. Les classes:

Les objets de même nature ont en générale les mêmes structures et le même comportement, la classe factorise les caractéristiques communes de ces objets et, comme sont nom le suggère, permet de les classifier. Nous donnons maintenant une définition de concept de classe :

4.2.1. Définition :

Comme pour l'objet, la classe est caractérisée par trois aspects essentiels :

Classe = instanciation + attributs + opérations.

La classe fournie d'abord un mécanisme d'instanciation qui permet de créer un objet (l'opération *new),* c'est à dire d'allouer un nouvel identifiant et une structure d'objet correspondante. L'objet ainsi est une instance de la classe. Toutes les instances d'une classe constituent l'extension de la classe.

Les attributs, appelés aussi les variables d'instance, ont un nom et un type qui est soit un type de base (simple ou construit), soit une classe. Dans ce dernier cas, l'attribut référencie un objet d'une classe (qui peut être la même ou une autre classe).

La structure implicite d'un objet est donc le n-uplet dont chaque attribut est une valeur ou un identifiant d'objet. Les opérations sont les opérations applicables à un objet de la classe. L'opération peut changer tout ou partie de l'état d'un objet et retourner des valeurs calculées à partir de cet état. On appelle **propriétés** de la classe ses attributs et ses opérations.

4.2.2 Graphe de classe :

Une classe joue plusieurs rôles qui sont souvent des sources de confusion. D'abord, c'est un type abstrait de données. Ensuite, c'est un générateur d'instances :

Une classe permet de créer tous les objets qui en sont des instances. Enfin, une classe permet de représenter d'une façon abstraite les références à des objets d'autres classes. Afin de distinguer ces différents rôles, il est commode de les représenter graphiquement par différents graphes en fonction de la relation interclasse représentée.

4.2.2.1. Graphe de généralisation :

Comme les types abstracts qui offrent la relation de sous type, les classes peuvent être organisées selon la relation de généralisation ou relation est un *(is a).* On parle

alors de graphe de classes ou de graphe d'héritage, car cette relation est la base d'héritage. Issue des modèles de données sémantiques, la généralisation est utile pour classifier les objets en fonction de leurs points communs et de leur spécificité par un même graphe. Elle peut être définie ainsi :

Généralisation : fonction qui, à une classe origine, dite sous-classe, fait correspondre une classe plus générale, dite superclasse.

La fonction inverse de la généralisation, qui correspondre a une classe toutes les sous-classes dont elle est la superclasse, est appelée spécialisation. Elle permet d'ajouter des propriétés spécifiques à une classe pour obtenir une sous-classe.

La représentation graphique faite correspondre par un arc oriente la sous-classe à la superclasse. Le lien de généralisation traduit le fait que les instances de la super-classe, c'est à dire que les objets qui appartiennent à la sous-classe possèdent non seulement les propriétés de la sous-classe mais aussi celle de la superclasse.

4.2.2.2 Graphe d'instanciation :

L'instanciation relie les instances de la classe (les objets) a leurs classes génératrices et peut être définie comme suit :

Instanciation : relation entre un objet et sa classe d'appartenance qui a permis de le créer.

La dichotomie entre objet et classe est maintenant par la plupart des langages compilés. Cependant, elle oblige le programmeur traiter les classes comme des types ; la modification d'une classe doit alors se faire en modifiant sa définition. Pour pouvoir manipuler les classes comme des objets, en particulier, pouvoir créer des classes durant l'exécution des classes du programme, il faut pouvoir générer les classes comme des objets, c'est-à-dire des instances de classe. Le concept de méta classe répond à ce besoin :

Meta classe : classe générateur de classes (ces instances), regroupant en particulier les opérations applicables aux classes et les structures de données communes a routes les classes.

Les opérations applicables aux classes sont la création et la destruction d'instances, et les opérations d'évolution des structures comme l'ajout ou la suppression d'attributs. Pour éviter une régression a l'infini pour créer le méta classe, celle-ci s'admet comme sa propre instance.

4.2.2.3. Graphe d'association et d'agrégation :

L'association est un concept essentiel dans la modélisation des données. Transposée dans le monde d'objet, l'association permet de définir au niveau des classes les liens connectant les objets :

Association : relation entre plusieurs classes, caractérise par un verbe, décrivant conceptuellement les liens entre les objets de ces classes. Une caractéristique importante d'une association est ça cardinalité, qui permet de contraindre le nombre d'objets maximum connectés a chaque objet.

Pour une association binaire entre Cl et C2, les cardinalités peuvent être les suivantes :

1-1 : chaque objet de Cl est connecté a au plus un objet de C2, et chaque objet de C2 est connecté a au plus un objet de Cl.

1-N : chaque objet de Cl est connecté a un ou plusieurs objets de C2, et chaque objet de C2 est connecté a au plus un objet de Cl,

N-M : chaque objet de Cl est connecte a plusieurs objets de C2, et chaque objet de C2 est connecte a plusieurs de Cl.

L'association binaire est représentée graphiquement par un trait, étiqueté par le nom de l'association et sa cardinalité, qui relie les deux classes.

Un cas particulier d'association est l'agrégation qui représente la relation partie-de ***(part of)***, cette association est utile pour modéliser les objets complexes, typiquement composés d'autres objets :

Agrégation : relation entre deux classes spécifiant que des objets d'autres classes (classe cible) sont les composants de l'autre classe (classe source).

4.3. Héritage :

L'héritage est un des apports essentiels des langages a objets. II peut être simple ou multiple et permet de supporter le polymorphisme des opérations. L'héritage fournit trois avantages majeurs. D'abord, en capturant la relation de généralisation, il facilite la modélisation des objets et la modularité du code. Ensuite, il favorise la réutilisation du code. En effet, les attributs et les opérations d'une classe sont réutilisables directement par les sous classes. Avec l'héritage multiple, la redondance du code peut 'être grandement réduite. Enfin, la liaison dynamique réduit le besoin de maintenance des programmes utilisateurs.

4.3.1. Héritage multiple :

Lorsque toute classe ne possède qu'une super-classe directe, le graphe est réduit à un arbre, l'héritage est dit simple. Cependant, lorsque les classes indépendantes ont les mêmes propriétés, celle-ci doit être dupliquée. L'héritage multiple résout ce problème en permettant à une classe de posséder plusieurs superclasses directes et d'hériter des propriétés de toutes ses superclasses.

Héritage multiple : mécanisme par lequel une sous-classe hérite les propriétés de plus d'une superclasse.

4.4. Le polymorphisme :

Le polymorphisme est un apport essentiel des technologies objet, qui améliore encore l'évolution et la maintenance d'une application. Le polymorphisme est la possibilité pour un même message, de déclencher des traitements différents, suivant les objets auxquels il est adressé. Le mécanisme de polymorphisme permet donc de donner le même nom à des traitements différents. L'intérêt fondamental du polymorphisme est de minimiser le coût de l'évolution du logiciel. Ainsi, il permet le choix dynamique d'un traitement en fonction de l'objet auquel il est appliqué.

D'une manière générale, lorsque pour toutes les sous-classes d'une classe C, une méthode m peut être décrite de la même façon, la définition de m dans la classe C assure le polymorphisme au niveau des sous-classes.

De plus, si la méthode *m* de C fait appel a une méthode *n* qui n'est pas définie dans C, car elle ne peut être décrite par un même énoncé dans les sous-classes de C, il faut définir la méthode *n* dans toutes les sous-classes afin d'assurer le polymorphisme de *m*. ainsi le polymorphisme de la méthode *n* permettra d'assurer le polymorphisme de la méthode *m*.

Lorsqu'un objet reçoit un message non polymorphe, le choix de la méthode à exécuter peut être déterminé sans ambiguïté. Dés la phase de compilation : cette technique est totalement statique. Au contraire, lorsqu'un objet reçoit un message polymorphe, un appel statique ne peut être réalisé. En effet, en phase de compilation, si on ne connaît pas explicitement et sans aucune ambiguïté l'identité de l'objet qui reçoit le message, on ne peut pas déterminer la méthode qui doit être effectivement appelée lors de l'envoi du message, en phase d'exécution. La détermination de la méthode a appeler ne peut être réalise qu'a l'exécution : cet appel de méthode est donc dynamique.

5. Conclusion :

L'approche objet se caractérise par une structure des programmes en classes d'objets. Tous les domaines qui utilisent cette approche nécessite le développement de logiciels complexes, manipulant de grandes quantités de connaissances. Pour résoudre cette complexité quelques souhaits sont souvent formulés :

- Possibilité de représenter directement les entités du monde réel dans les environnements informatiques, sans nécessite de les &former ou de les décomposer.
- Possibilité de réutiliser ou d'étendre les logiciels existants.
- Travail avec des environnements de développements riches, notamment pour la création des interfaces et la trace des exécutions.
- Disponibilité d'outils interactifs permettant la création rapide d'interfaces homme-machine graphiques de grande qualité et capables de réagir a tout événement extérieur.
- Facilité de prototypes rapides des applications, notamment des interfaces

homme-machine et de la logique générale des traitements, sans qu'il soit nécessaire de tout coder.

- Facilite d'exploitation de parallélisme tors de l'implémentation sur des machines multiprocesseurs et/ou distribuées.

SECTION II : Modélisation objet avec UML

1. Introduction :

UML est né de la fusion des trois méthodes qui ont influencées beaucoup la modélisation objet au milieu des années 90 : OMT, OOD et OOSE.

Issue du terrain et fruit d'un travail d'experts reconnus, UML est le résultat d'un large consensus. De très nombreux acteurs industriels de renom ont adopte UML et participent a son développement.

En l'espace d'une poignée d'années seulement, UML est devenu un standard incontournable. Utiliser les technologies objets sans UML relève de l'hérésie. Lorsqu'on possède un esprit un tant soit peu critique, on est en droit de s'interroger sur les raisons qui explique un engouement si soudain et massif ! UML est-il révolutionnaire ?

Il y a déjà longtemps que l'approche objet est devenue une réalité, les concepts de base de l'approche objet sont stables et largement éprouvés. De nos jours, programmer objet, c'est bénéficier d'une panoplie d'outils et de langage performant. L'approche objet est une solution technologique incontournable. Ce n'est plus une mode, mais réflexe quasi-automatique dés lors qu'on cherche à concevoir des logiciels complexes qui doivent résister à des évolutions incessantes.

Mais l'approche objet est moins intuitive que l'approche fonctionnelle. Malgré les apparences, il est plus naturel pour l'esprit humain de décomposer un problème informatique sous forme d'une hiérarchie de fonctions atomiques et de données, qu'en terme d'objet et d'interaction entre ces objets.
Autre problème critique : l'application des concepts objets nécessite une très grande rigueur. Le vocabulaire précis est un facteur d'échec important dans la mise en oeuvre d'une approche objet (risque d'ambiguïté et d'incompréhension). Beaucoup de programmeurs ne pensent souvent objet qu'à travers un langage de programmation, or les langages orientes objet ne sont que des outils qui proposent une manière particulière d'implémenter certains concepts objets. Ils ne valident en rien l'utilisation de ces moyens techniques pour concevoir un système conforme à la philosophie objet.

Pour remédier à ces inconvénients majeurs de l'approche objet il nous faut donc :

- ❖ Un langage qui doit permettre de :
  - Représenter des concepts abstraits (graphiquement par exemple).
  - Limiter les ambiguïtés (parler d'un langage commun, au vocabulaire précis, indépendant des langages orientes objets).
  - Faciliter l'analyse (simplifier la comparaison et réévaluation des solutions).

- ❖ Une démarche d'analyse et de conception objet pour :
  - Ne pas effectuer une analyse fonctionnelle et se contenter d'une implémentation objet, mais penser objet dés le départ.
  - Définir les vues qui permettent de décrire tous les aspects d'un système avec des concepts objets.

En d'autres termes, il faut disposer d'un outil qui donne une dimension méthodologique à l'approche objet et qui permette de mieux maîtriser sa richesse.

2. Historique :

Les deuxièmes moutures (versions) des méthodes OOD et OMT, appelées respectivement Booch'93 et OMT-2, se sont rapprochées tant et si bien qu'elles sont devenues plus ressemblantes que différentes. Après cette première phase d'unification par osmose, les variations subsistantes sont minimes et concentrées principalement dans la terminologie et la notation. Booch'93 s'inspire d'OMT et adopte les associations, les diagrammes d'**Harel**, les traces d'événements..., OMT-2 s'inspire de Booch et introduit les flots de message, les modèles hiérarchiques et les sous-systèmes, les composants modèles, et surtout retirent du modèle fonctionnel les diagrammes de flots de données, hérité d'un passé fonctionnel et peu intégrés avec la forme générale d'OMT. A ce stade là, les deux méthodes offrent une couverture complète du cycle de vie, avec toutefois une différence notable dans l'éclairage. Booch-93 insiste plus sur la construction, alors qu'OMT-2 se concentre sur l'analyse et l'abstraction. Néanmoins, il n'existe entre les deux méthodes aucune incompatibilité majeure.

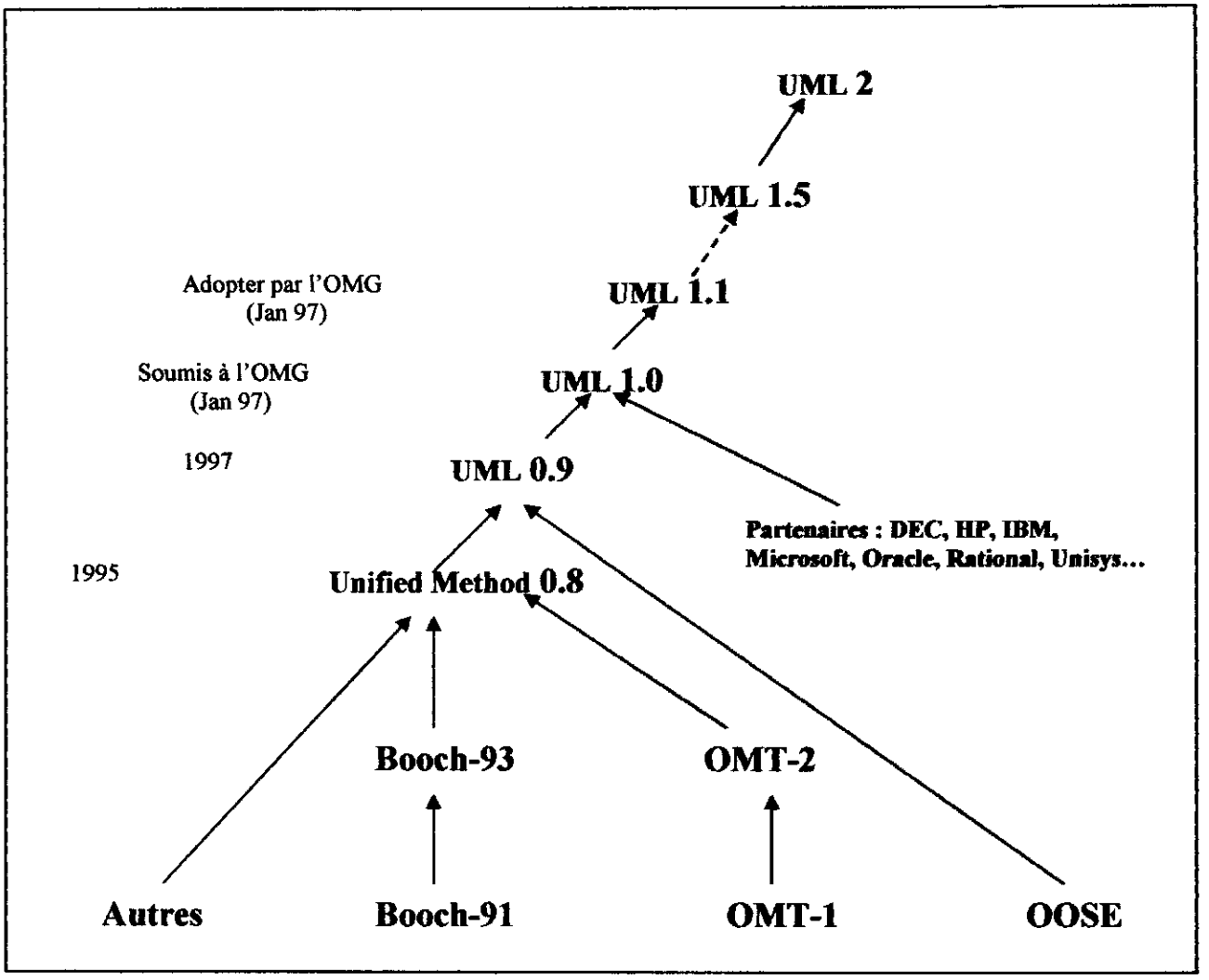


Figure 1 : **Principales étapes de la définition d'UML.**

La première description de la méthode unifiée a été présentée en Octobre 1995, dans un document intitulé « **Unified Method V0.8** ». Ce document a été largement diffusé, et les auteurs ont recueilli plus de 1000 commentaires détaillés, de la part de la communauté des utilisateurs. Ces commentaires ont été pris en compte dans la version 0.9 qui a paru en Juin 96, mais c'est surtout la version 0.91 disponible fin Septembre 96 qui permet de se rendre compte de l'évolution de la méthode unifiée. Les principales modifications consistent dans l'enrichissement de la notation, et surtout dans la réorientation de la portée de l'effort, d'abord vers la définition d'un langage universel unifie pour la modélisation objet et plus tard vers la standardisation du processus de développement objet. Ceci se traduit par un changement de nom. La méthode unifiée se

transforme en UML, la description d'UML version 0.1 a été remise à l'OMG en Janvier 1997 en vue de sa standardisation.

A ce jour, les créateurs d'UML insistent tout particulièrement sur le fait que **la notation UML est un langage de modélisation objet et non pas une méthode objet**. La notation UML est conçue pour servir de langage de modélisation objet indépendamment de la méthode mise en œuvre.

3. Concepts:[Bouzghoub 98][1],[Lai 99][2],[Muller 97][3],[Fannader 99][4]

3.1. Cas d'usage et Scénario :

La définition des besoins peut partir des informations gérées dans le domaine ou des activités réalisées, les deux approches sont complémentaires et doivent être menées simultanément. En introduisant la notion de cas d'usage ***(use case)***, UML fournit cependant un point d'entrée pour définir un périmètre fonctionnel.

Cette méthode est directement issue des méthodes d'ingénierie des processus (**Business Process Engineering**); elle correspond à l'ensemble des actions à entreprendre en réponse à une sollicitation significative d'un acteur. En fait, le cas d'usage peut prendre des formes diverses selon la nature du domaine considéré: contrôle industriel, gestion des stocks, contrôle financier... etc.

Un même cas d'usage peut se dérouler de différentes manières, chacune modélisée par un scénario.

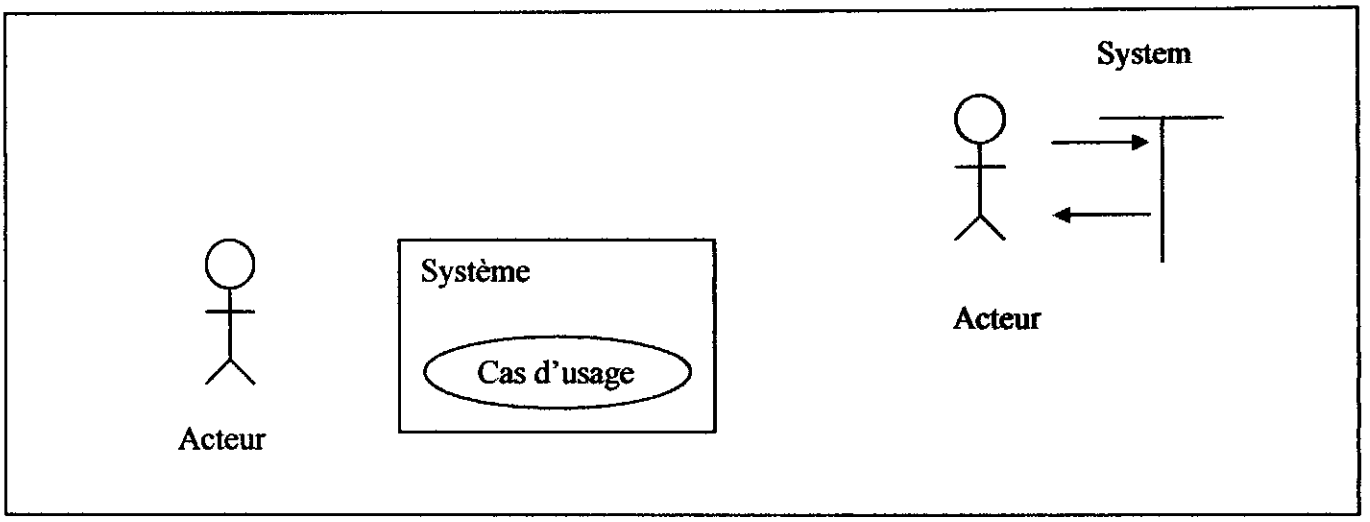


Figure 2 : **Acteurs, Cas d'usage, Scénario.**

3.2. Objets, Types, Classes et Interfaces :

La modélisation des objets avec UML supporte la distinction entre les objets du domaine et les objets du système, ainsi entre les types (description des comportements) et les classes (implémentation des types).

UML peut ainsi supporter une approche classique (modèle entité/relation avec ou sans sous-types) ou une méthode objet (types caractérisés par des opérations). Le choix méthodologique se fera avec l'analyse des interactions, puisque alors les entités verront attribuer des responsabilités, ce qui en fera des objets dons le comportement sera décrit par des types d'objets, eux-mêmes implémentes par des classes. La notion d'interface est utilisée pour représenter les types non implémentés.

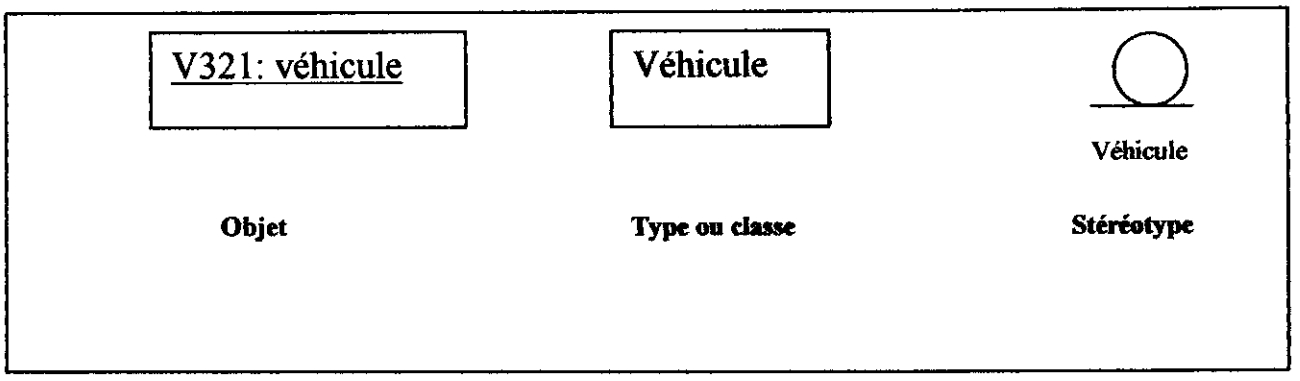


Figure 3 : **objet, type ou classe et stéréotype.**

3.3 Relation, association et composition :

De même que les objets sont représentés par des types (ou classes), les liens entre les objets sont représentés par des relations. Lorsque la relation est caractérisée par des informations, on parle des associations. Lorsque les relations sont structurelles, on parle des compositions. Comme dans les méthodes classiques, les relations sont décrites par des rôles et par des cardinalités minimales et maximales.

3.4. Messages et Opérations :

Comme nous l'avons vu, l'activité du système se réduit à l'activité des objets, qui se coordonne par échange de ***messages***. Pour qu'un objet traite un message, il faut que celui-ci corresponde à une opération connue de l'objet, c'est à dire qu'elle soit définie par l'interface de la classe. A noter cependant que les messages et les opérations sont définis indépendamment les uns des autres, a charge pour l'analyse d'assurer la consolidation.

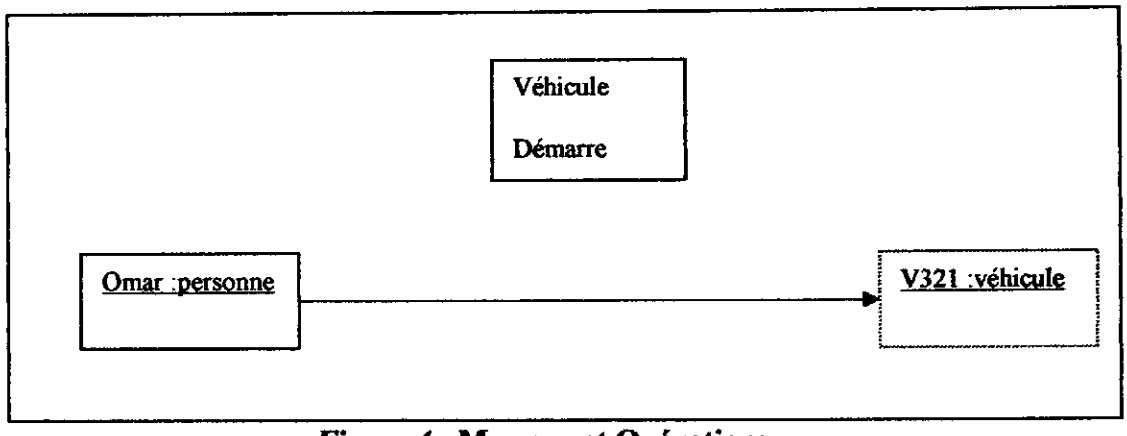


Figure 4 : **Message et Opérations**

3.5. Etat, Evènement et Transition :

Pour modéliser la dynamique des états, UML reprend les concepts utilisés par les méthodes classiques : lorsque le comportement d'un objet est déterminé par un ensemble fini de situations, Celles-ci sont représentées par des ***états*** ; ces états peuvent être déterminés de deux manières :

- En termes d'activités (modèle de **Moore**) : les états sont associés aux opérations exécutées par les objets, et les transitions associées aux ***événements.***
- En terme d'objets (modèle de **Meaty**) les états sont définis en terme de conditions sur les attributs, et les transitions associées aux opérations.

Les deux approches peuvent être combinées sous réserve que la sémantique utilisée soit parfaitement claire.

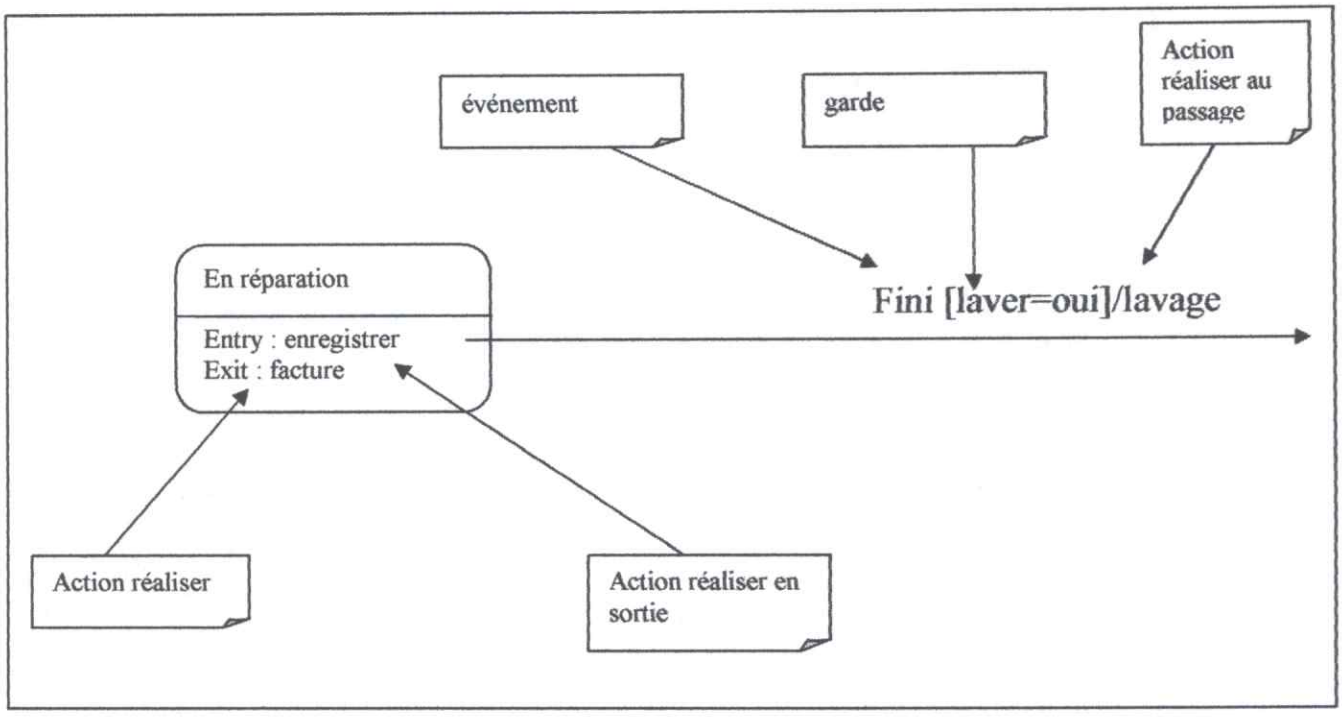


Figure 5 : Etats et Transitions.

Par ailleurs, UML reprend le modèle des *StateChart* pour représenter les comportements sur différents niveaux en terme de disjonction et de conjonction :

- Les états peuvent être détaillés en sous-états exclusifs, les transitions étant alors fractionnées en fonction d'une condition (garde).
- Les états peuvent être détaillés en sous états simultanés, les transitions deviennent elles aussi simultanées.

3.6 Action, activité et opération :

La notion de temps (durée) est définie par domaine : c'est l'intervalle entre deux événements. Une *action* peut être instantanée pour un domaine et avoir une durée pour un autre (deux événements différents pour marquer le début et la fin). C'est généralement le cas lorsque l'on passe d'un domaine applicatif au domaine physique.

Pour un domaine donné, on parlera *d'opération* pour une action instantanée, *d'activité* dans le cas générale. La décomposition des états et des transitions joue

un rôle essentiel pour modéliser les différents niveaux d' abstraction : une action sera représentée par une *transition* dans un domaine (action instantanée) et par un état dans le contexte de l'objet qui l'implémente (activité avec une durée).

3.7. Paquetage et Composant :

Un paquetage regroupe un ensemble d'objets logiques solidaires du point de vue développement. Un composant est un élément physique du système qui implémente un ensemble d'interfaces. Un composant est l'équivalent physique d'une classe (ou de plusieurs).

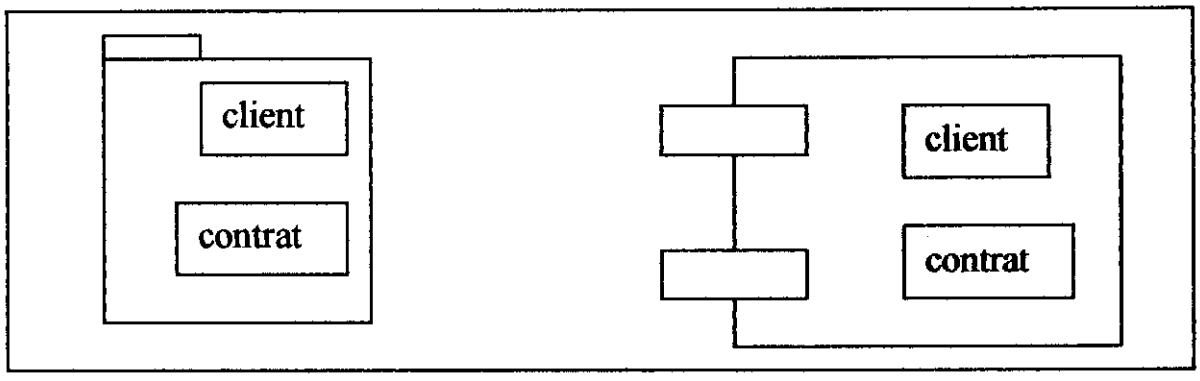


Figure 6 : **Paquetage et Composant.**

4. Les diagrammes d'UML :

Comme son nom l'indique, UML est fait pour construire des modèles. Pour se faire, on dispose de certain nombre d'outils : ***les diagrammes***. Les différents types de diagrammes sont :

Vue statique du système :

- Diagrammes de cas d'utilisation.
- Diagrammes d'objets.
- Diagrammes de classes.
- Diagrammes de composants.
- Diagrammes de déploiement.

Vue dynamique du système :

- Diagrammes de collaboration.
- Diagrammes de séquence.
- Diagrammes d'état-transition.
- Diagrammes d'activités.

4.1 Vue statique du système :

4.1.1. Diagramme de cas d'utilisation :

Un cas d'utilisation (**use case**) permet de clarifier et organiser les besoins en délimitant le système étudie. Ils peuvent être utilisés tout au long du processus du développement. Dans un premier temps, il s'agit de définir de manière informelle les exigences auxquelles devra répondre le système. On peut ensuite préciser les variantes (scénarios) avec les opérateurs de composition et de spécialisation. On peut par ailleurs utiliser les cas d'usages pour formaliser les comportements et les synchronisations des actions (pré et post-condition).

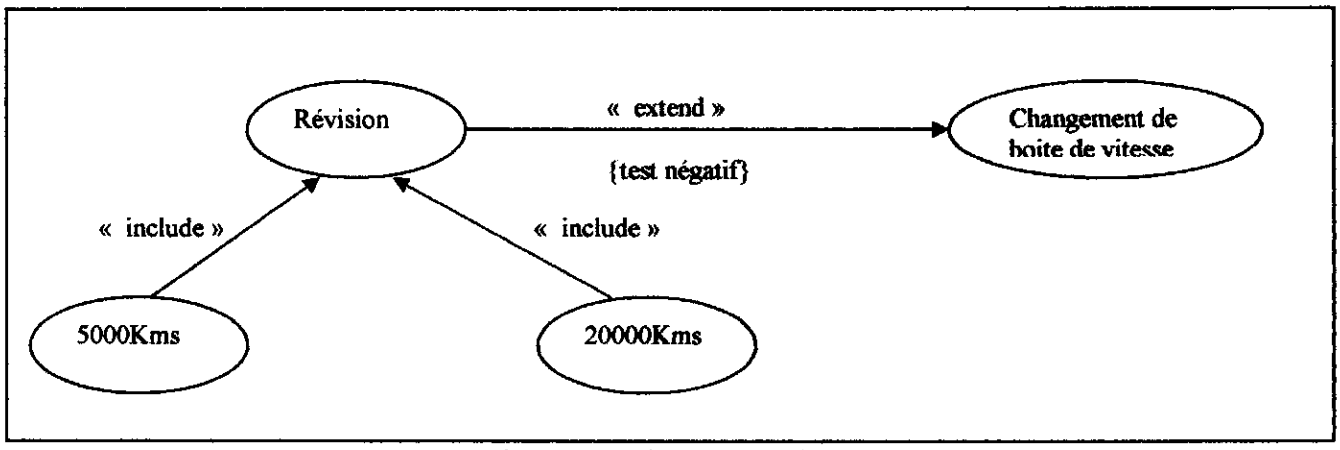


Figure 1 : Diagramme de cas d'usage.

Les éléments de base de cas d'utilisation sont :

- **Acteur :** c'est une entité externe qui agit sur le système, il peut 'être soit : un opérateur, un autre système, ... etc. Les acteurs sont hiérarchisés en deux types :

- **Acteurs principaux :** sont les acteurs qui interagissent directement sur le système (comme les opérateurs de saisie).
- **Acteurs secondaires :** se sont les acteurs qui ont un rôle de contrôle.

En UML, le système est simplement représenté par un rectangle, contenant les différents cas d'utilisation, autour duquel figurent les différents acteurs. Un cas d'utilisation peut être décrit avec une représentation textuelle. Il est formé d'un scénario de base qui décrit le cas le plus courant et des scénarios d'exceptions qui décrivent les cas particuliers du scénario de base.

- **Association d'extension :** permet de structurer et de relier des cas d'utilisation.

4.1.2. Diagramme de classes :

Le diagramme de classes correspond à une vue statique du système. Les outils gèrent différents niveaux de détail, depuis les stéréotypes jusqu'au détail des interfaces. Ces possibilités peuvent être utilisées différemment selon l'avancement du projet et la méthodologie. L'on dispose des types d'entités, des types ou classes d'objets et de l'implémentation des classes. Les objets du domaine sont représentés par des types d'entités, ces types sont statiques et peuvent être spécialisés en sous-types. L'attribution de responsabilités aux types représente un choix méthodologique une nouvelle étape.

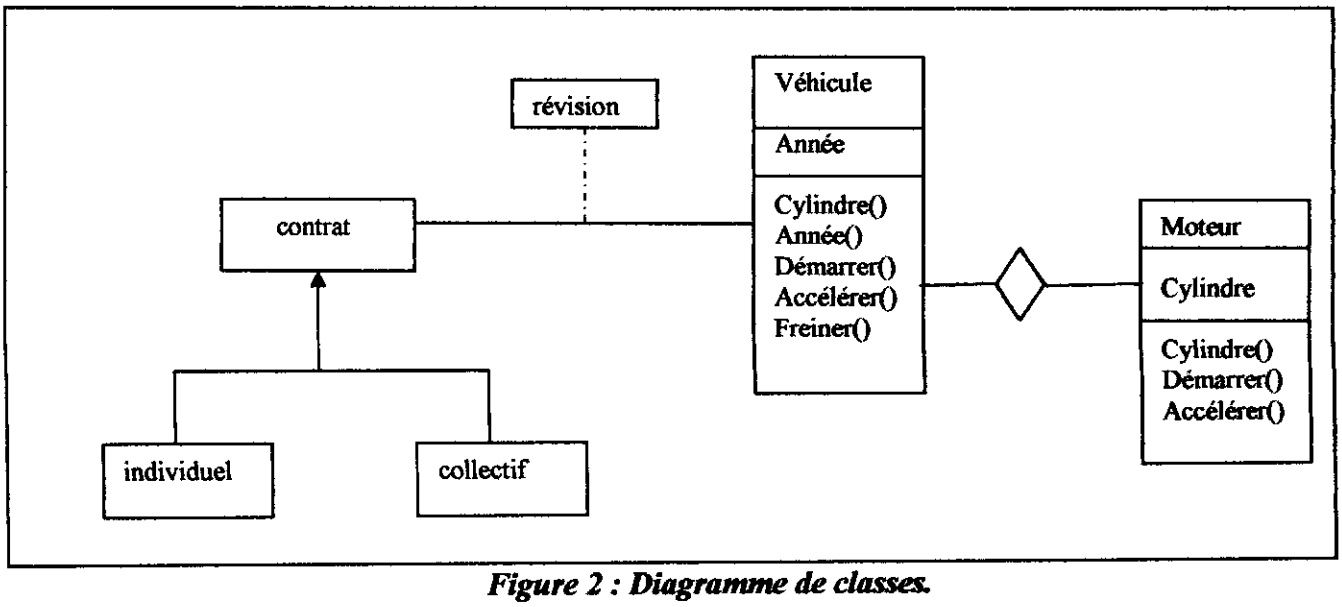


Figure 2 : Diagramme de classes.

4.1.3. Diagramme d'objets :

Ce type de diagramme UML montre les objets (instances de classes dans un état particulier) et des liens (relations sémantiques) entre les objets. Les diagrammes d'objets représentent un cas particulier, une situation concrète a instant donné; il exprime a la fois la structure statique et dynamique.

4.1.4. Diagramme des composants :

Les diagrammes de composants permettent de décrire l'architecture physique et statique d'une application en terme de modules: Fichiers sources, librairie, exécutables,...etc. Il existe trois types de modules : la spécification (les interfaces de classe), les corps (la réalisation des classes) et les spécifications génériques (classes paramétrables).

4.1.5. Diagramme de déploiement :

Les diagrammes de déploiement montrent la disposition physique des matériels qui component le système la répartition des composants sur ces matériels. Les ressources sont représentées sous forme de nœud. Ces nœuds sont connectés entre eux, à l'aide d'un support de communication. La nature des lignes de communication et leurs caractéristiques peuvent être précises.

4.2. Vue dynamique du système :

4.2.1. Diagramme de collaboration :

Ce diagramme représente les objets participants au scénario, les canaux de communication entre eux et les messages échangés. Il s'agit en faite d'une version simplifiée du diagramme de séquences, et on peut donc utiliser ce dernier directement pour modéliser les interactions. Par contre, le formalisme du diagramme de collaboration est très proche du diagramme de classes (il s'agit en faite d'un diagramme d'objets), ce qui en fait un outil pédagogique.

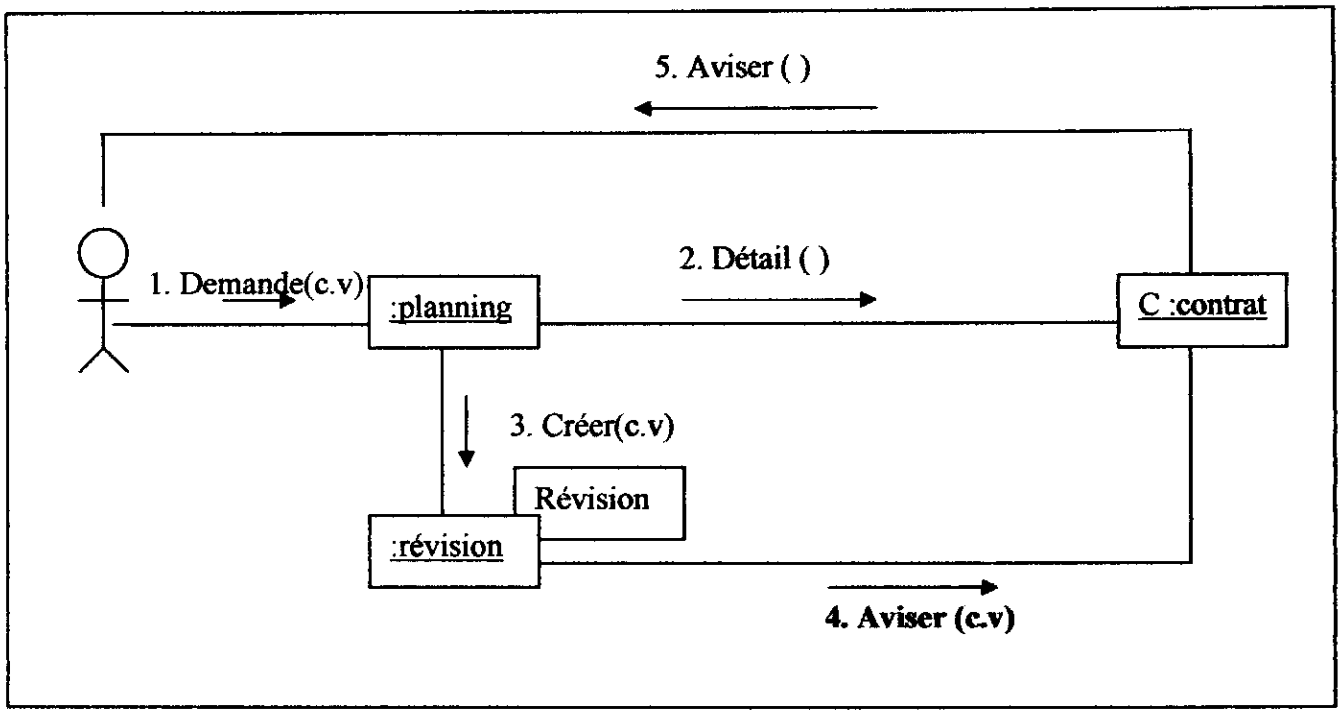


Figure 3 : diagramme de collaboration.

4.2.2. Diagramme de séquence :

Le diagramme de séquence fournit l'axe central de toute la modélisation des interactions :

- Il reprend les éléments du diagramme de collaboration avec un formalisme plus rigoureux, en particulier pour la représentation du temps.
- Sa sémantique suppose aussi bien les méthodes classiques que les méthodes objets.
- Il est utilisable a différents niveaux : système, composants, objets,... etc.
- Il supporte une démarche itérative par enrichissement successif :

1. Participants et responsabilités.
2. Séquences des messages.
3. Structures des traitements.
4. Contraintes d'exécution.

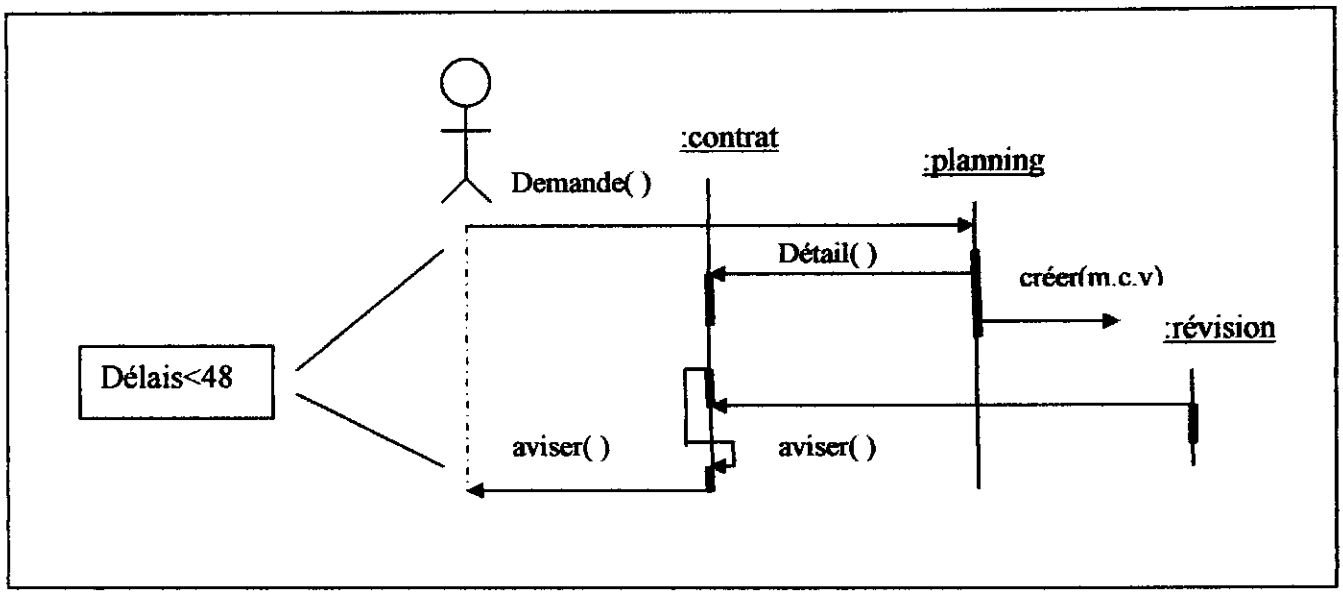


Figure 4 : Diagramme de séquences.

4.2.3. Diagramme d'état/transition :

Pour traiter des problèmes de synchronisation, UML a repris un outil depuis longtemps utilisé dans le domaine du temps réel : les ***StateCharts***. Ce formalisme supporte la décomposition d'automates d'états finis sur plusieurs niveaux en termes d'états et des transitions simultanées ou successives. Les interactions du diagramme de séquences (événements) peuvent ainsi être associées à des changements d'états (transitions), parfois conditionnes par des gardes. Une action peut être associée au franchissement d'une transition. Activités et attentes d'événements représentent des états stéréotypés.

Ce type d'outils peut être utilise dans plusieurs contextes :

- ✓ Dans sa version simplifiée (sans décomposition) pour formaliser des comportements pilotés par les états ainsi pour modéliser le déroulement des activités dans leurs contextes organisationnel et technique.
- ✓ Dans sa version complexe pour spécifier la synchronisation des sous-systèmes imbriqués.

Les diagrammes d'états peuvent être reformulés pour représenter les activités et leurs synchronisations. On peut combiner les deux aspects (objet et activité) dans un même diagramme et y ajouter les acteurs pour représenter l'ensemble dans son contexte organisationnel ou technique.

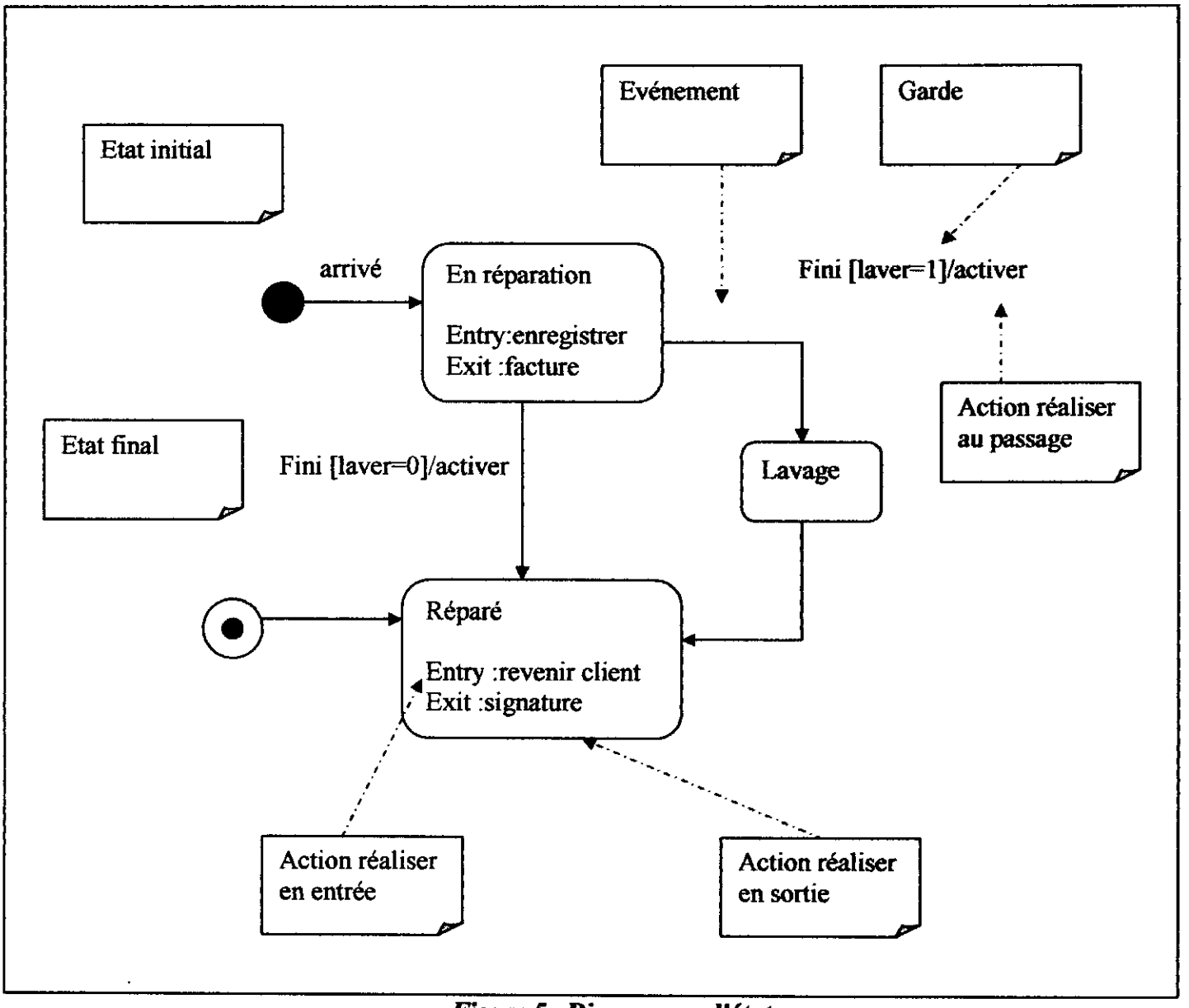


Figure 5 : **Diagramme d'états.**

4.2.4. Diagramme d'activités :

Les diagrammes d'activités sont les représentations des comportements d'une opération ou d'un cas d'utilisation en terme d'action.

Le diagramme d'activité représente l'état d'exécution d'une méthode, sous la forme d'un déroulement d'étapes. Une activité est un automate à deux états dont le franchissement de la transition entre ces derniers est conditionné par la fin d'activité.

Dans le diagramme d'activité, il existe deux types de transitions : ***la transition automatique*** (flèche simple) qui est franchie lorsque l'activité précédente est finie; et la t***ransition gardée*** (flèche avec une condition entre (crochets)) qui est franchie si la condition est vérifiée.

Les diagrammes d'activité permettent aussi de visualiser les activités qui s'exécutent en parallèle au moyen de barre de synchronisation. Ils peuvent être découpés en couloirs d'activités pour visualiser la responsabilité des objets au sein du système à modéliser.

On peut clairement visualiser les objets dans le diagramme d'activité. A chaque objet est associe une ligne de vie et ses activités. La manipulation des objets permet aussi la modification de leurs états. On peut alors indiquer cet état de l'objet dans le symbole de l'objet a l'aide de crochets. Il est possible de visualiser les objets qui ont été crées par les activités ; pour cela, on relie par une flèche en pointillé l'objet et l'activité qui la crée.

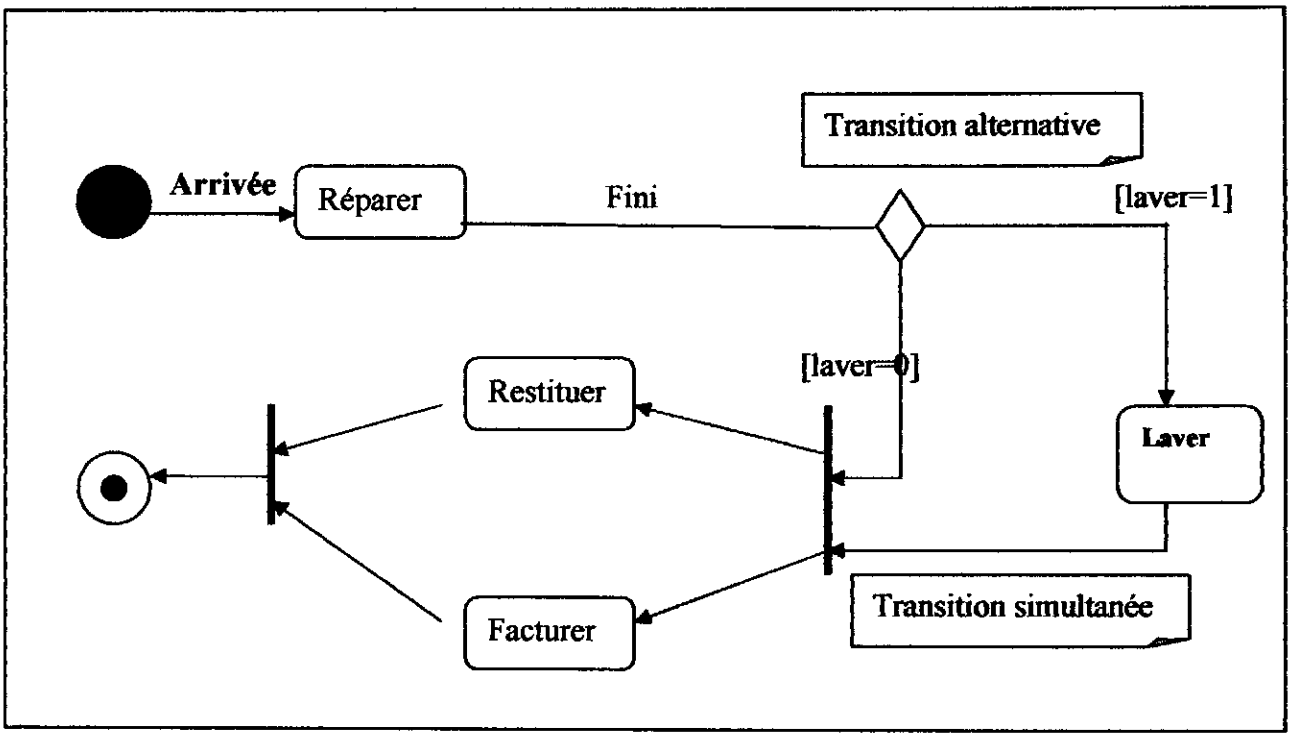


Figure 6 : **Diagramme d'activités.**

5. Redondance des diagrammes :

Les diagrammes d'interaction, d'états et d'activités sont partiellement redondants et sont rarement utilisées simultanément et systématiquement. Bien que la présence des diagrammes ne soit pas identique dans les différents environnements commercialisés, les principes d'organisation des diagrammes sont toujours les mêmes :

- Le diagramme de collaboration traite propriétairement des objets et des messages; il peut aussi représenter les synchronisation, mail sans les activités.
- Le diagramme de séquence offre les mêmes possibilités que le diagramme de collaboration, a quoi s'ajoutent les activités et les contraintes de synchronisation. On peut y ajouter le contrôle des activités (alternatives et boucles), mais pour cela le diagramme d'activité est mieux adapte.

- Le diagramme d'activité est un sous-ensemble du diagramme d'états qui reformule les scénarios pour y ajouter les conditions de déclenchement et les contrôles (alternatives et boucles).

- Le diagramme d'états traite aussi bien de l'état des activités que des objets.

- Le diagramme de flux combine les diagrammes d'états et d'activités en y ajoutant l'environnement (organisation et équipement, représentés par les couloirs).

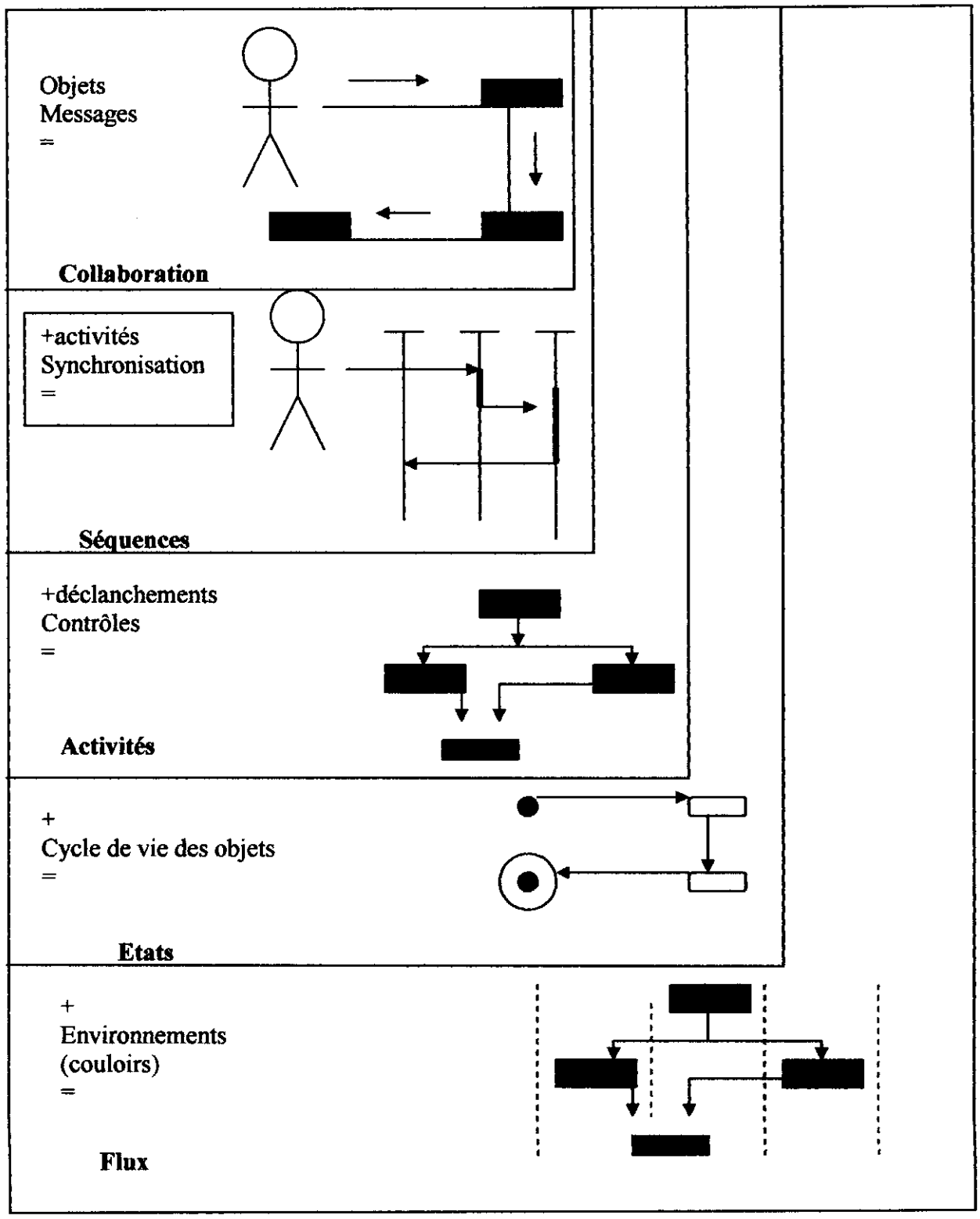
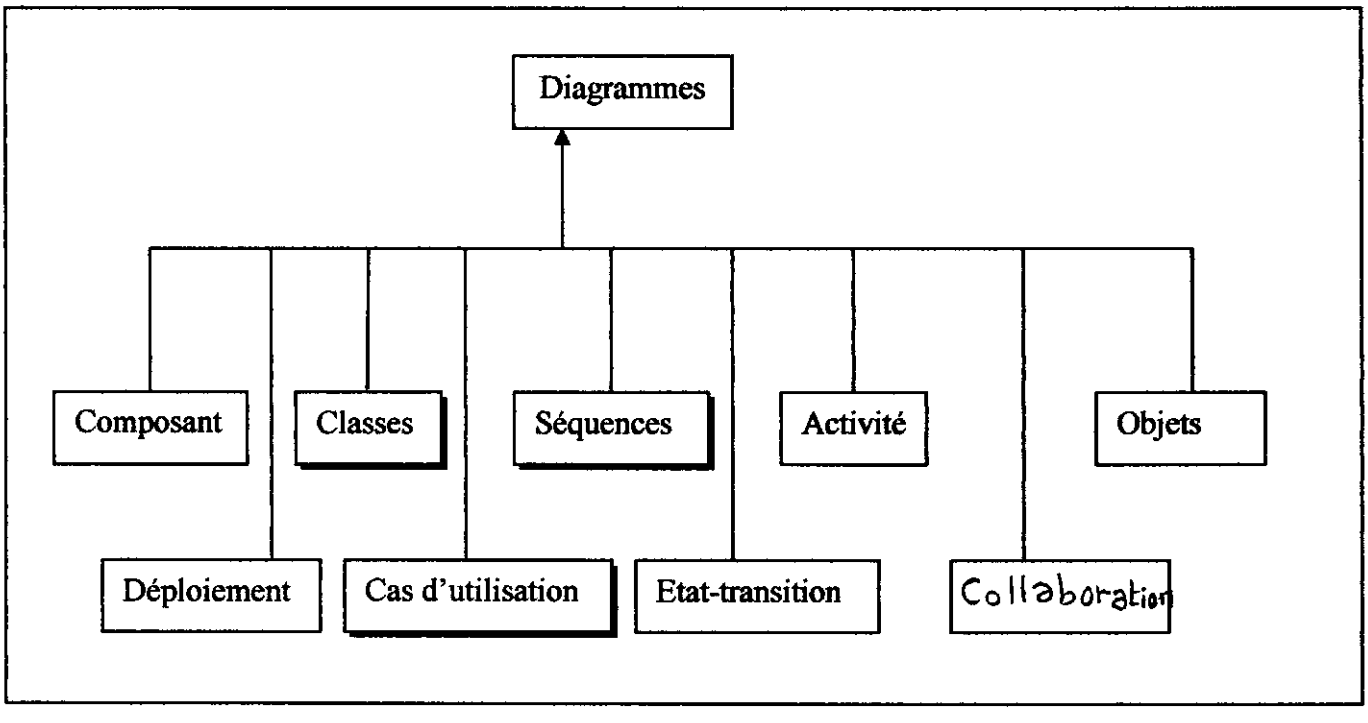


Figure 7 : Redondance des diagrammes d'interaction.

6. conclusion :

Dans cette section, on a fait une description générale de la méthode UML, en commençant par les différents concepts de cette méthode, ensuite les neufs diagrammes, et enfin on a indiqué quelques critiques concernant ces diagrammes et leur développement.

Les neufs modèles d'UML sont :



NB : les diagrammes représentes par des rectangles ombres sont les diagrammes de conception qui seront pris en compte lors de notre travail.

Chapitre III

Le processus de développement :

1. Introduction :

Pour créer des bons logiciels capables de satisfaire des besoins bien précis et pour un 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 quelles le développement doit passer, les ressources qu'il doit consommer et les objectifs auxquelles doit répondre les systèmes conçus.

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

UML est un langage de modélisation objet et une norme dans le domaine. Entre autre, UML n'est pas une méthode (on va le voir dans le chapitre qui vient, section: Modélisation objet avec UML) car il ne décrit pas une démarche ou un processus précis pour le développement logiciel.

UML est un langage qui permet de représenter des modèles mais il ne définit pas le processus d'élaboration de ces modèles or une méthode propose aussi un processus qui régit l'enchaînement des activités de production d'une entreprise. Les auteurs d'UML sont tout à fait conscient de l'importance du processus ; mais comment prendre en compte toutes les organisations et culture de l'entreprise ?

Un processus est adapté (donc très lié) au domaine d'activité de production d'une entreprise. Même s'il constitue un cadre général, il faut l'adapter au contexte de l'entreprise. Les auteurs d'UML préconisent d'utiliser une démarche : guidée par les besoins de l'utilisateur, centré sur l'architecture, itérative et incrémentale.

2. Le processus : [Fannader 99][4]

Le processus proposé (ou la démarche suivie) consiste à définir une démarche de développement logiciel utilisant la plupart des diagrammes d'UML depuis la phase d'analyse (expression des besoins) jusqu'à la phase finale qui est le test du logiciel, donc notre processus de développement comprend les étapes suivantes :

2.1. Expression des besoins :

C'est le point de départ de toute modélisation informatique et si on veut avoir une modélisation juste, l'expression des besoins doit être claire et bien précise parce que c'est d'elle qu'on va servir comme référence tout au long de la réalisation du projet.

Comme son nom l'indique, dans l'expression des besoins on cite les exigences auxquelles doit le système répondre et les conditions qu'il doit vérifier.

2.2 Conception :

La conception à 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.[8]

La partie de conception débute a partir de la détermination de l'architecture globale du système a développé, c'est-à-dire l'élaboration de ses structures statiques et dynamiques.

2.3. Réalisation (implémentation) :

Elle correspond à la programmation proprement dite des fonctions sur la base des informations venant de la phase conception. [8]

2.4. Test :

Dans cette partie, on doit ; par l'exécution du logiciel ; s'assurer que le système conçu satisfait à ces objectifs fonctionnels spécifier en phase d'expression des besoins en testant quelques cas d'utilisation.

Dans notre projet on a trouvé une bonne idée pour tester notre logiciel :

Comme on va utiliser le langage UML comme langage de modélisation et comme notre projet consiste a la « Conception et réalisation d'un mini éditeur de diagrammes UML », pourquoi donc, après la réalisation du logiciel, ne pas l'utilisé pour générer les diagrammes modélisés tout au long du processus de développement et ça sera donc, un vrai test du logiciel dans un projet réel. C'est ça notre vision pour la partie test du logiciel.

CHAPITRE IV

Expression des besoins

1. Introduction :

Une bonne modélisation informatique part d'une bonne expression des besoins qui soit claire et précise et qui servira comme référence tout au long de la réalisation d'un projet, et dans le cas de notre projet notre vision est la suivante :

- Nous pensons qu'un logiciel simple et efficace sera plus utile qu'un logiciel complexe, qui ne couvre pas les besoins de base de la modélisation UML.

Un éditeur de diagrammes UML doit en effet permettre de répondre au mieux aux besoins essentiels de gestion d'un projet, sans pour autant offrir trop de fonctionnalités. Les concepts directeurs sur lesquels nous nous basons sont **la *simplicité*** et **la *flexibilité*** :

❖ ***La simplicité*** se caractérise par la présence de l'essentiel des fonctionnalités exigibles d'un éditeur de diagrammes UML simple, en éliminant au début les fonctionnalités superflues.
Simple veut aussi dire ergonomique, l'éditeur privilégiera l'efficacité dans le cadre d'une gestion de projet et une interface utilisateur claire. Notre philosophie est donc de ne pas implémenter de fonctionnalités superflues mais un nombre minimal de fonctionnalités répondant aux besoins primaires de la modélisation.

❖ ***La flexibilité*** se caractérise par la possibilité d'étendre simplement l'éditeur. Notre approche se veut de mettre l'accent sur la *modularité* de l'éditeur, lui permettant d'accroître ses fonctionnalités de manière fluide.

- Notre logiciel doit aussi être facile à utiliser : il doit permettre une prise en main rapide pour la spécification des différents diagrammes.
- L'éditeur doit aussi aider l'utilisateur dans la création des diagrammes en lui permettant la couper/coller des composants pour faciliter la tâche.

2. L'architecture générale du système :

Notre système est basé sur un seul élément primordial qui est : **l'éditeur graphique**.

2.1 Présentation générale de l'éditeur graphique :

L'éditeur de diagrammes UML est un outil pour la modélisation graphique des diagrammes du langage UML.

Cet outil offre dans notre cas à l'utilisateur du système la possibilité de dessiner (modéliser) chacun des trois diagrammes de manière graphique en lui offrant les outils nécessaires qui sont :

1. Un menu qui guidera l'utilisateur durant la conception de diagrammes, ce menu l'aidera à créer des diagrammes de classes, cas d'utilisation et de séquences et lier ces diagrammes a un projet.

2. Une palette de dessin qui contient tous les boutons pour dessiner des classes, paquetage, commentaires, cas d'utilisation, objet...etc, suivant le diagrammes choisi et les règles du langage UML.

3. Une zone de dessin pour que l'utilisateur place les composants de ses diagrammes dans celle la et crée des relations entre ses composants.

Si les composants sont définis, il faudrait que toutes les liaisons soit entre des composants qui sont connectables, sinon l'outil n'accepte pas de réaliser la liaison.

Les objectifs de l'éditeur graphique sont :

- La modélisation graphique de l'ensemble des trois diagrammes avec tous leurs composants et les relations entre ces composants.
- Offrir toutes les facilités d'utilisations habituelles que l'on trouve dans la plupart des applications graphiques, par exemple copier, coller, supprimer,...etc.

3. Modélisation des besoins :

Pour exprimer les besoins on utilise les diagrammes des cas d'utilisation (use case) qui nous permettent de voir le comportement attendu du système en cours de développement, donc on doit passer par deux étapes :

- Déterminer les acteurs du système.
- Déterminer les principaux cas d'utilisation (use case).

Remarque :

Tous les diagrammes de cas d'utilisation, de séquence et de classes que nous allons présentés dans ce chapitre et les chapitres qui viennent sont réalisés en utilisant notre éditeur de diagrammes UML.

Cette utilisation du logiciel dans la génération des diagrammes représente donc la partie Test et Validation du processus de développement de notre logiciel.

3.1. Les acteurs :

Notre éditeur de diagrammes UML s'adresse principalement à des personnes désirant modéliser des projets en utilisant le formalisme UML, tout en leur permettant d'étendre celui-ci selon leur besoin. Il s'adresse donc à :

- ✓ Des développeurs.
- ✓ Des architectes travaillant à la modélisation d'un projet informatique.
- ✓ Des étudiants qui s'intéressent a la modélisation avec le langage UML.
- ✓ Toute personne désirant utiliser l'un des diagrammes du langage UML pour modéliser un problème car UML ne s'adresse pas seulement à un informaticien, mais a tout le monde donc on peut avoir une personne non informaticienne qui veut utiliser UML, en plus cette personne peut être non initié.

3.2. Les principaux cas d'utilisation :

Les principaux cas d'utilisation auxquels doit répondre notre système sont les suivants :

1) La création d'un projet.
2) La création d'un diagramme de classe.

3) La création d'un diagramme des cas d'utilisation.
4) La création d'un diagramme de séquence.

Ces cas d'utilisation sont modélisés par le diagramme des cas d'utilisation comme montre la **figure1**.

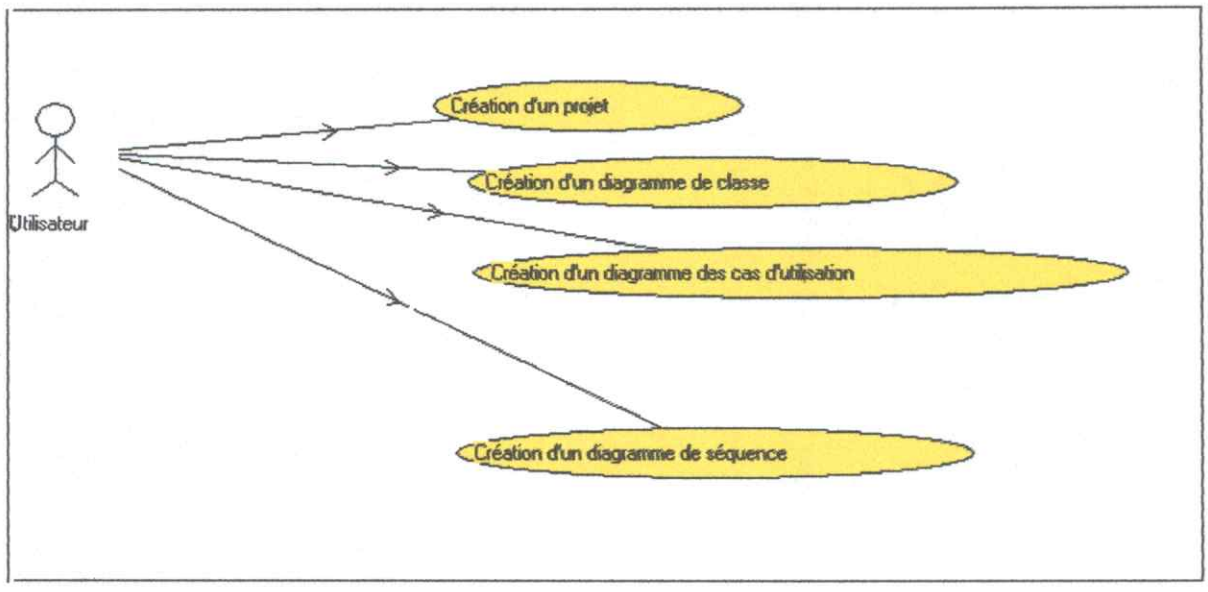


Figure 1 : Diagramme des cas d'utilisation pour les cas principaux du système.

Chaque cas de ces cas d'utilisation principaux sera ensuite raffiné et représenter avec des diagrammes des cas d'utilisation de second niveau. Donc pour mieux comprendre le diagramme principale et pour une grande clarification on décompose chaque cas d'utilisation en sous cas.

Comme nous l'avant vue dans le chapitre précédent les diagrammes des cas d'utilisation représentent les fonctions principales du système du point de vue des utilisateurs, mais ils ne modélisent pas ces fonctions de manière plus détaillée, c'est pour cette raison que nous documentons ces diagrammes avec des diagrammes de séquence pour modéliser de manière temporelle les déférentes interactions entre les utilisateurs du système et le système lui-même.

Cas d'utilisation «Création d'un Projet» :

Ce cas d'utilisation est l'union de trois autres cas d'utilisation qui sont : la création d'une vue de diagrammes de classes, la création d'une vue de diagrammes de cas d'utilisation, la création d'une vue de diagrammes de séquences. Chaque cas de ces trois cas nous mène a la création de l'un des diagrammes que notre éditeur supporte (il faut toujours garder à l'esprit qu'un projet est une collection de vues qui sont eux aussi une collection de diagrammes de même type (donc pour gérer un projet il faut gérer l'ensemble des vues qui le forme et pour gérer une vue il faut faire le même chose avec ces diagrammes)).

Le cas d'utilisation <<Création de projet>> est représenté par la **figure2.**

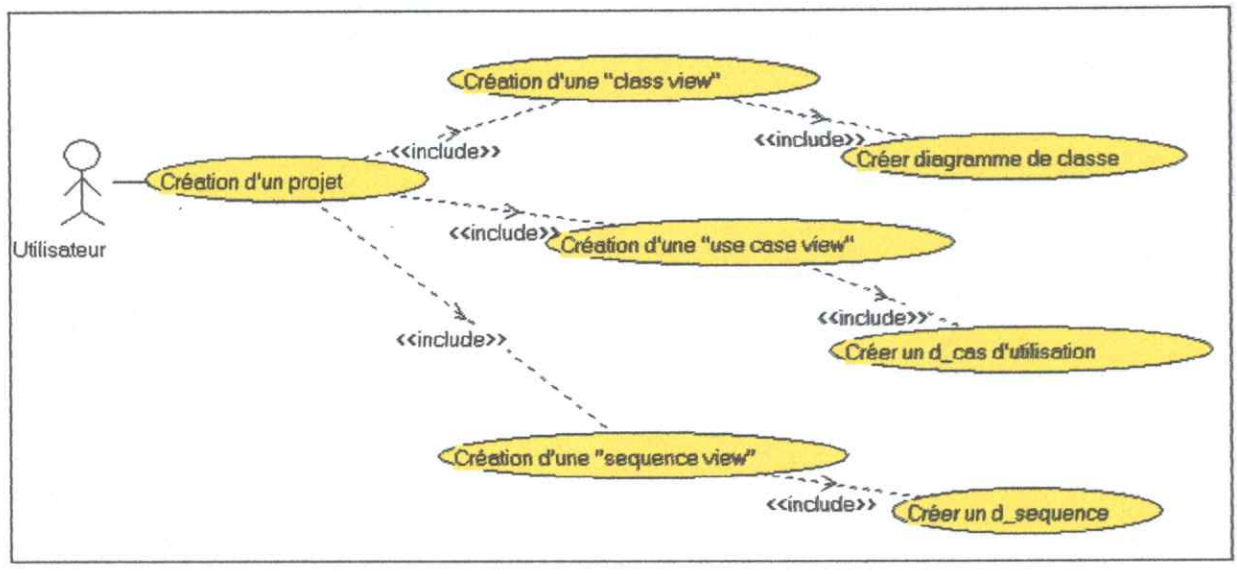


Figure 2 : Diagramme des cas d'utilisation pour le cas «création d'un projet».

Cas d'utilisation «Créer une vue de diagrammes» :

Dans notre projet une vue joue le rôle d'un dossier par rapport a un fichier : elle rassemble des diagrammes de même type pour organiser le projet, facilité sa compréhension et permettre de le géré facilement.

Donc les trois type de vue ont le même comportement a l'exception du type de diagramme a géré. On peut donc mettre ces trois cas dans un même cas d'utilisation qui sera le suivant :

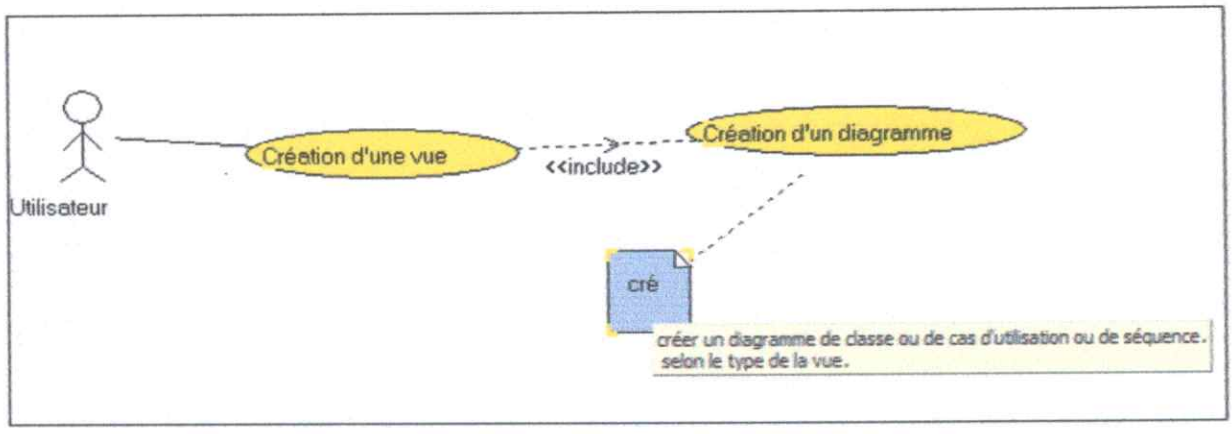


Figure 3 : Diagramme des cas d'utilisation pour le cas «création d'une vue».

Cas d'utilisation «Créer un Diagramme de Classe» et «Créer un Diagramme de Cas d'Utilisation» :

Le cas d'utilisation Créer un Diagramme de Classe apparaît à deux moments différents (appelés par l'utilisateur ou après la création d'un projet), mais il a le même comportement. Même chose pour Le cas d'utilisation Créer un Diagramme de Cas d'Utilisation.

Ces deux cas ont le même comportement à l'exception que les composants à gérer sont différents et aussi sont les relations entre composants.

La création d'un diagramme de classe ou d'un diagramme de cas d'utilisation comporte plusieurs opérations, comme créer des nouveaux composants (classes, paquetage, note pour le diagramme de classe et acteur, cas d'utilisation, paquetage, note pour le diagramme de cas d'utilisation), importer des composants a partir d'autres diagrammes de même type que le diagramme destination et ensuite ajouter des relations entre ces composants et enfin définir ces relations en spécifiant leurs caractéristiques.

Le cas d'utilisation « Création d'un diagramme de classe ou de cas d'utilisation » est présenté dans la **figure4.**

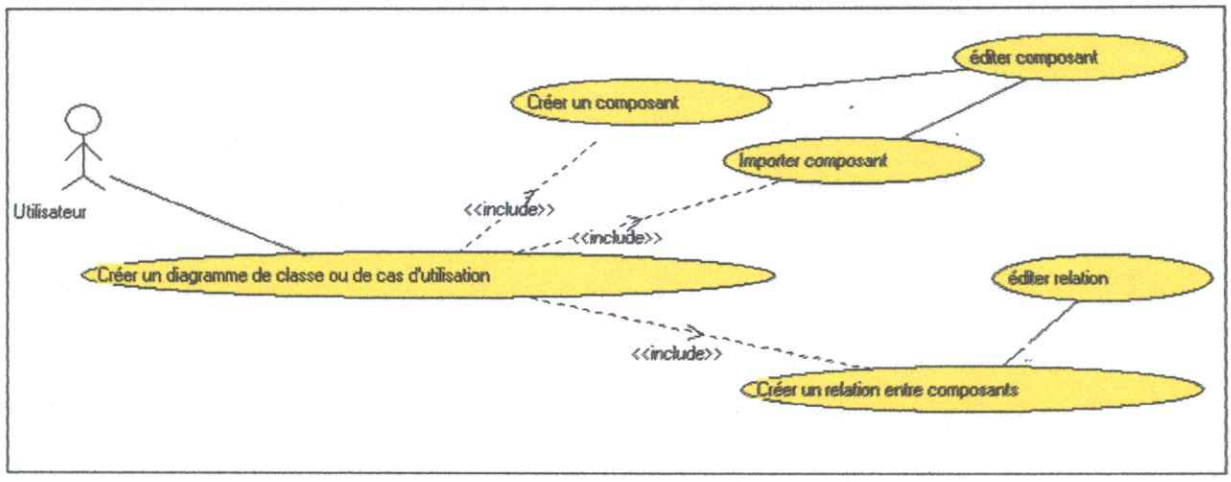


Figure 4 : Diagramme des cas d'utilisation pour le cas créer un diagramme de classe ou de cas d'utilisation.

Cas d'utilisation «Créer un Composant» :

1) Scénario :

1. L'utilisateur clique sur le bouton correspondant au composant à dessiner.
2. Déplace la souris sur la zone de dessin.
3. Relâche la souris sur l'endroit ou il veut placer le composant.
4. Le système demande le nom du composant (sauf les notes sans nom).
5. L'utilisateur entre le nom du composant.
6. Le système dessine le composant.

2) Diagramme de séquence pour le cas « Créer un Composant » :

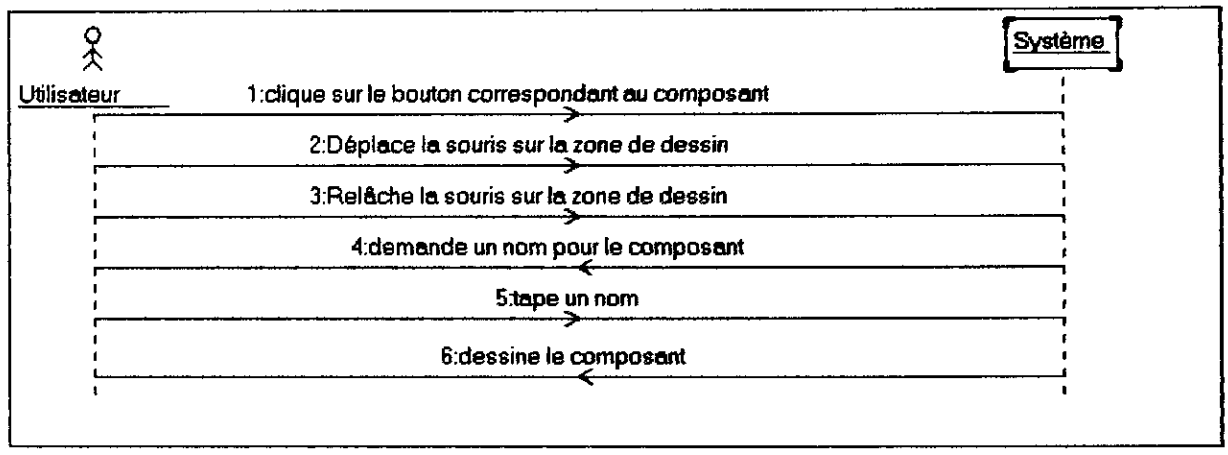


Figure 5 : Diagramme de séquence pour le **cas d'utilisation « Créer un Composant ».**

Cas d'utilisation «Importer un Composant» :

1) Scénario :

1. L'utilisateur choisit le composant a importé (le sélectionne).
2. Demande de copier le composant.
3. Choisit le diagramme cible.
4. Demande de coller le composant.
5. Le système importe le composant.

2) Diagramme de séquence pour le cas « Importer un Composant »:

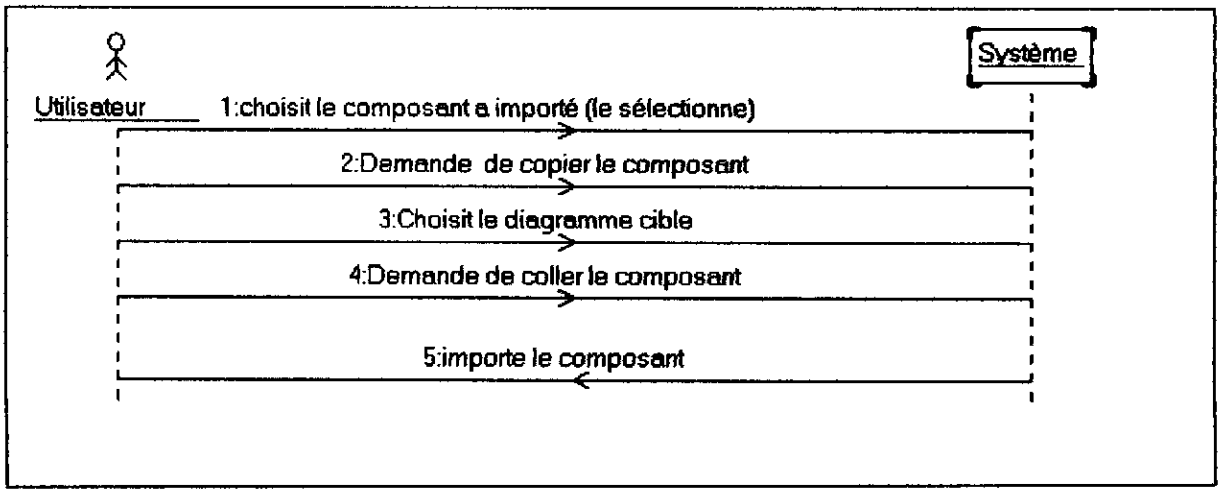


Figure 6 : Diagramme de séquence pour le **cas d'utilisation « Importer un Composant ».**

Cas d'utilisation «Créer une Relation entre Composants» :

1) **Scénario :**

1. L'utilisateur clique sur le bouton correspondant a la relation a crée.
2. Choisit le composant source de la relation.
3. Choisit le composant destination de la relation.
4. Le système demande de donner un nom à la relation.
5. L'utilisateur définit un nom pour la relation.
6. Le système crée la relation.

2) **Diagramme de séquence pour le cas «Créer une Relation entre Composants» :**

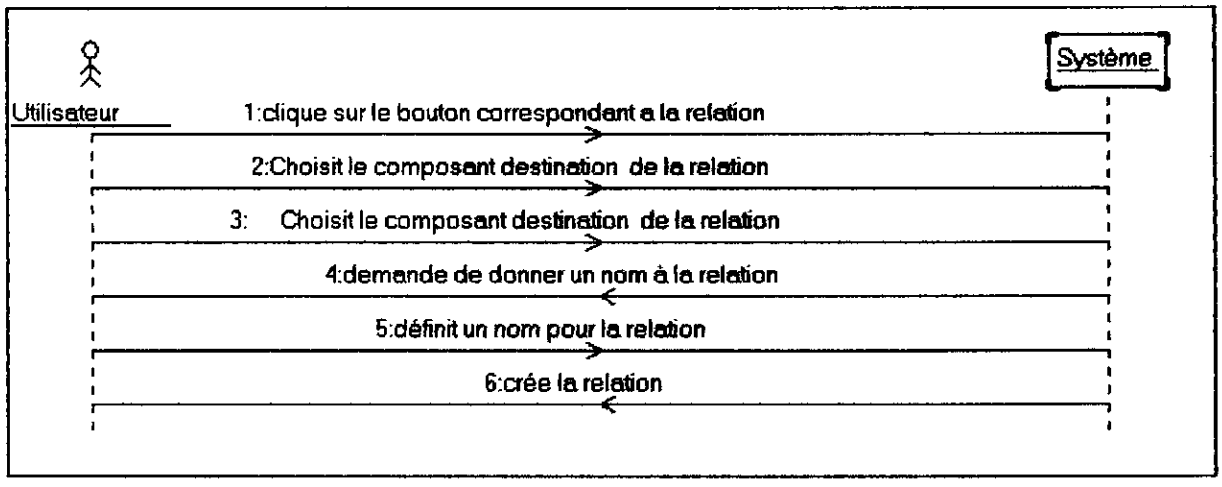


Figure 7 : Diagramme de séquence pour le **cas d'utilisation «Créer une Relation entre Composants».**

Cas d'utilisation «Editer Composant» :

1) **Scénario :**

1. L'utilisateur sélectionne le composant à éditer.
2. Demande d'éditer le composant.
3. Le système affiche la boite de dialogue «éditer composant».
4. L'utilisateur définit les propriétés du composant.
5. Le système vérifier la justesse des propriétés.
6. Le système ajoute ces propriétés au composant.

2) **Diagramme de séquence pour le cas «Editer Composant» :**

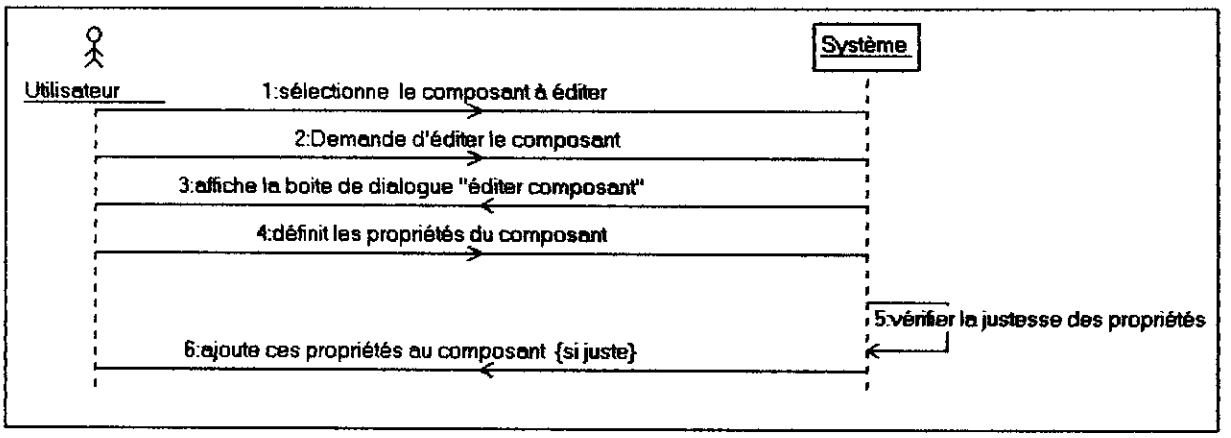


***Figure 8 : Diagramme de séquence pour le* cas d'utilisation «Editer Composant».**

Cas d'utilisation «Editer Relation» :

1) **Scénario :**

1. L'utilisateur sélectionne la relation à éditer.
2. Demande d'éditer la relation.
3. Le système affiche la boite de dialogue «éditer relation».
4. L'utilisateur définit les propriétés de la relation.
5. Le système vérifier la justesse des propriétés.
6. Le système ajoute ces propriétés à la relation.

2) **Diagramme de séquence pour le cas «éditer Relation» :**

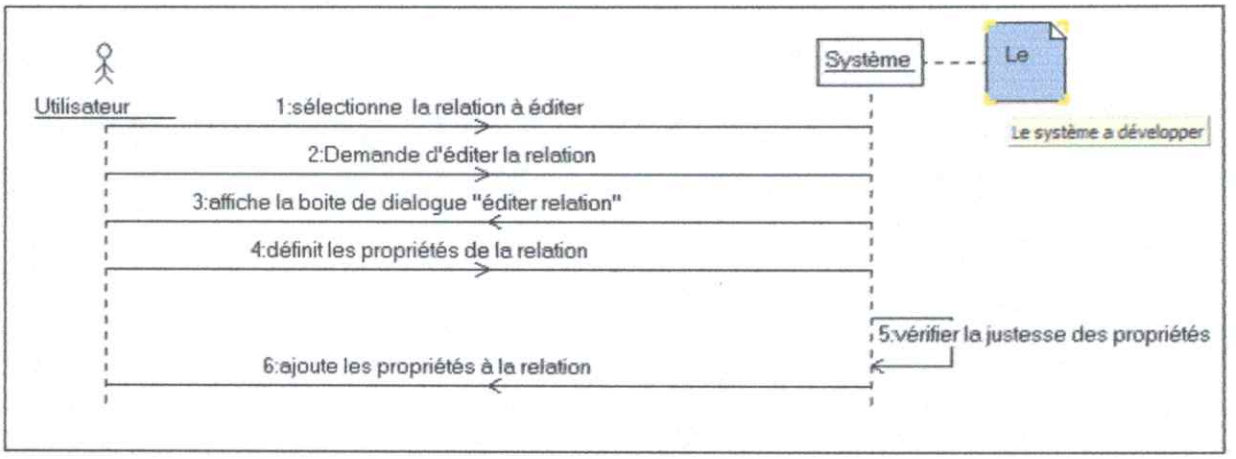


Figure 9 : Diagramme de séquence pour le **cas d'utilisation «éditer Relation».**

Cas d'utilisation «Créer un Diagramme de Séquence» :

La création d'un diagramme de séquence comporte les opérations suivantes : créer des objets, créer des acteurs, créer un message entre composants, ajouter une note.

Le cas d'utilisation «Créer un diagramme de séquence» est présenté dans la **figure4.**

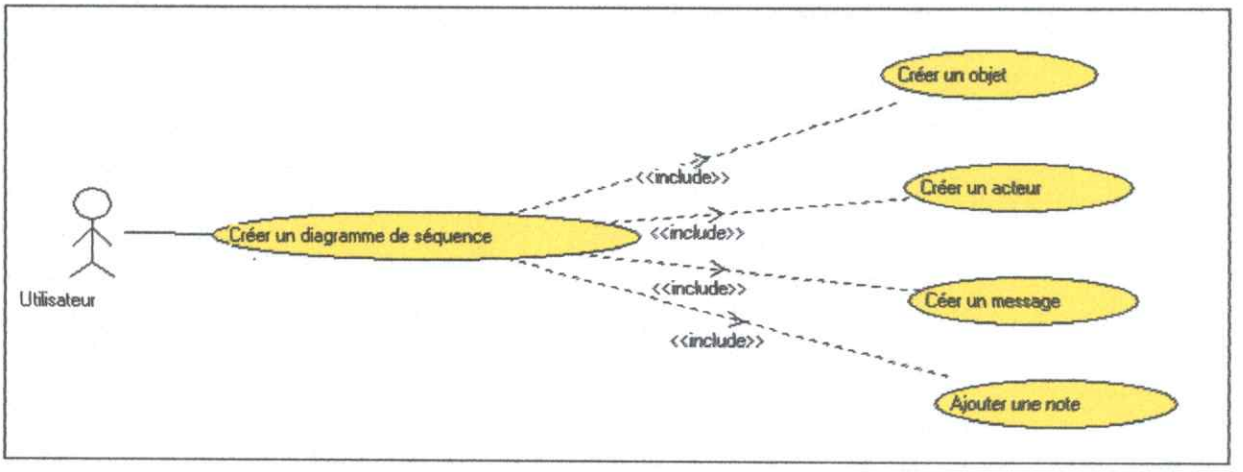


Figure 10: Diagramme des cas d'utilisation pour le cas créer un diagramme de séquence.

Cas d'utilisation «Créer un Objet» :

1) **Scénario :**

1. L'utilisateur clique sur le bouton : créer objet.
2. Déplace la souris sur la zone de dessin.
3. Relâche la souris sur l'endroit ou il veut placer l'objet.
4. Le système demande un nom pour l'objet.
5. L'utilisateur tape le nom de l'objet.
6. Le système dessine l'objet.

2) Diagramme de séquence pour le cas «Créer un Objet» :

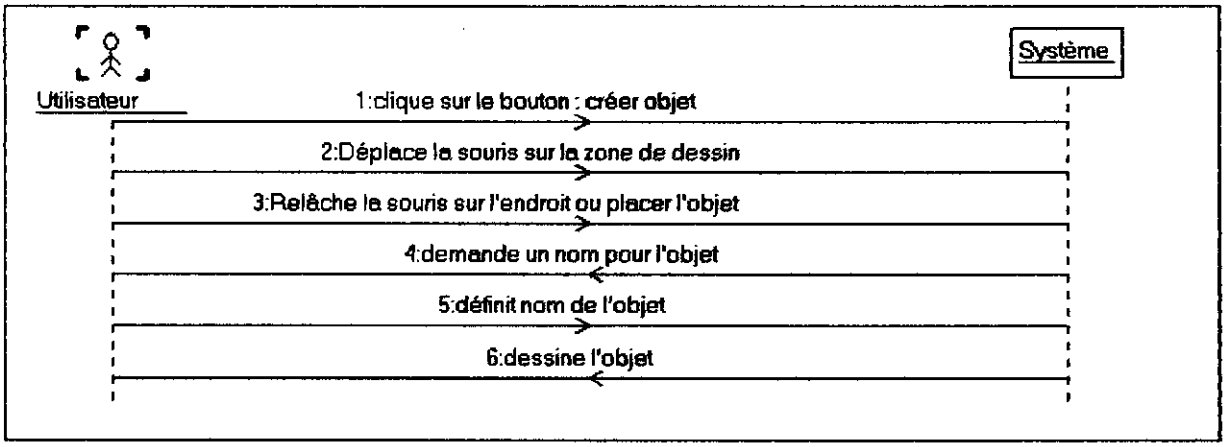


***Figure 11 : Diagramme de séquence pour le* cas d'utilisation «Créer un Objet».**

Cas d'utilisation «Créer un Acteur» :

2) Scénario :

1. L'utilisateur clique sur le bouton : créer acteur.
2. Déplace la souris sur la zone de dessin.
3. Relâche la souris sur l'endroit ou il veut placer l'acteur.
4. Le système demande un nom pour l'acteur.
5. L'utilisateur tape le nom de l'acteur.
6. Le système dessine l'acteur.

2) Diagramme de séquence pour le cas «Créer un Acteur» :

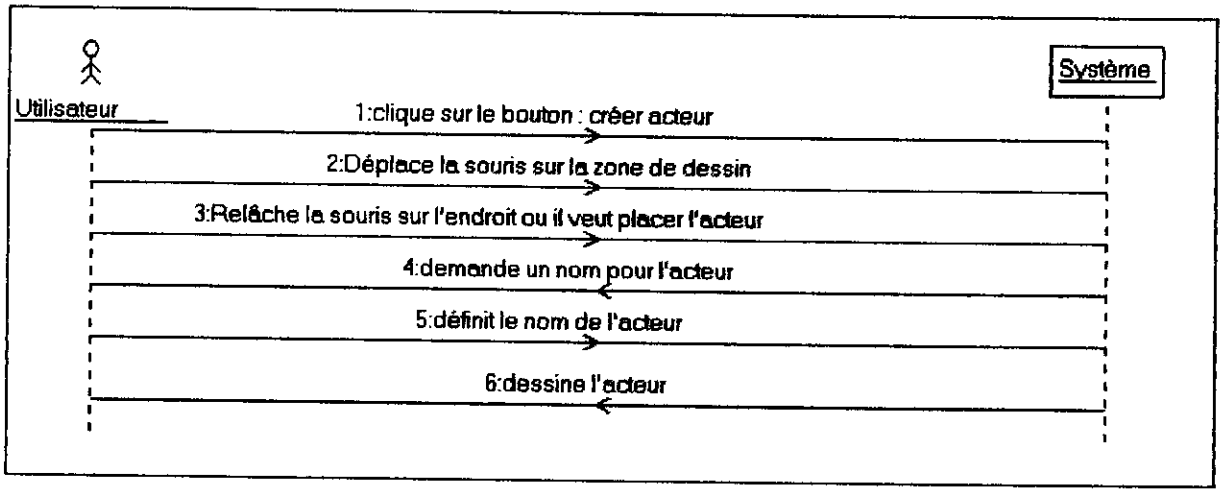


Figure 12 : Diagramme de séquence pour le **cas d'utilisation «Créer un Acteur».**

Cas d'utilisation «Créer un Message» :

1) **Scénario :**

1. L'utilisateur clique sur le bouton correspondant au message.
2. Choisit le composant (objet ou acteur) source du message.
3. Choisit le composant destination du message.
4. Le système demande le message à afficher.
5. L'utilisateur tape le message.
6. Le système crée le message.

2) **Diagramme de séquence pour le cas «Créer un Message» :**

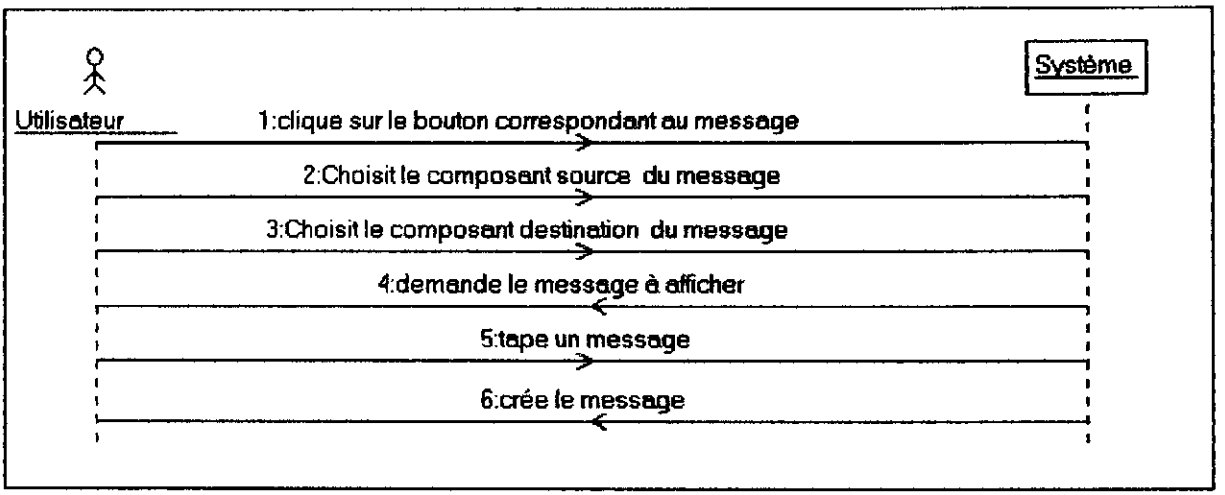


Figure 13 : Diagramme de séquence pour le **cas d'utilisation «Créer un Message».**

Cas d'utilisation «Ajouter une Note» :

1) Scénario :

1. L'utilisateur clique sur le bouton : créer note.
2. Déplace la souris sur la zone de dessin.
3. Relâche la souris sur l'endroit ou il veut placer la note.
4. Le système crée la note.

2) Diagramme de séquence pour le cas «Ajouter une Note» :

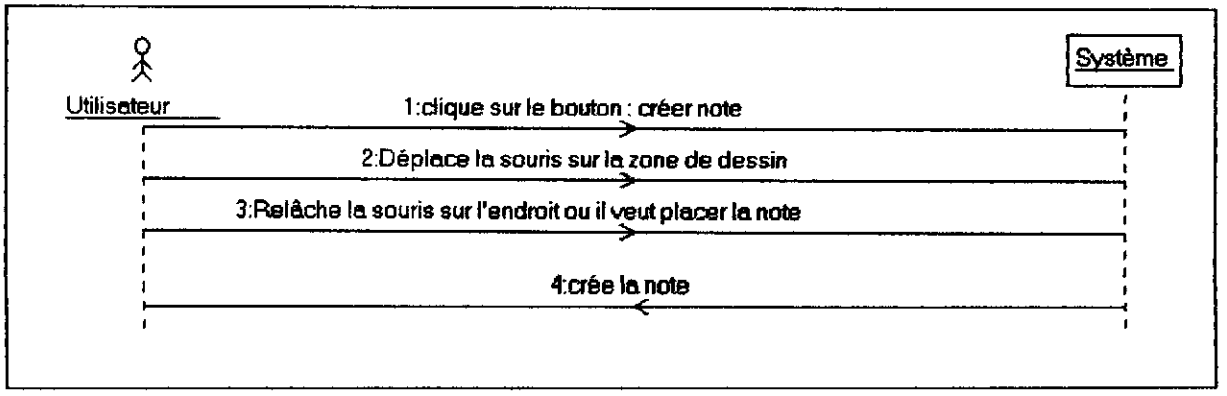


Figure 14 : Diagramme de séquence pour le **cas d'utilisation «Ajouter une Note».**

4. Conclusion :

Dans ce chapitre, on a définit les besoins de l'utilisateur et tout ce qu'il attend du système, et a partir de cette définition des besoins que nous allons commencer la conception en essayons de concevoir un système qui répond a ces besoins.

CHAPITRE V

La Conception

1. Introduction :

1.1. Interface graphique :

Avec le développement des micro-ordinateurs et des stations de travail, le nombre d'utilisateurs de l'informatique s'est considérablement accru. Le développement et la génération des écrans graphiques et les environnements de développements d'interfaces conviviales ont permis de répondre à ces nouveaux besoins.

Les outils d'interaction directe, principalement la souris et l'écran tactile, ont permis un nouveau type de communication avec l'ordinateur appelé : **« le desktop metaphor »**. Bien que ce concept ait été inventé au débuts des années 70 en même temps que SMALTALK, il n'a été généralisé qu'au début des années 80 avec l'explosion du bureautique, ce concept consiste justement a interagir avec un système non seulement a l'aide question-réponse ou de menu de choix multiples, mais aussi a l'aide des icônes graphiques représentant des métaphores des objets quotidiens. Ces métaphores ont été particulièrement développées dans le domaine de la bureautique ou les objets courant de bureau, comme les dossiers, les documents, le téléphone, l'horloge, la calculette et autres ciseaux et gommes ont été matérialisés par des formes graphiques évocatrices de la réalité, permettant une exploration et une utilisation intuitive des services associés.

1.2. Conception et ergonomie des interfaces :

Malgré l'existence d'outils graphiques de haut niveau. La réalisation d'interfaces conviviales n'est pas garantie sans le respect de quelques règles d'ergonomie. En effet, si les boites a outils fournissent un grand choix d'objets graphiques. L'agencement de ces objets, leur disposition à l'écran et le nombre d'éléments utilisés est laissé à la charge du développeur d'interfaces. Les interfaces ne doivent pas êtres conçus de façon indépendante des utilisateurs a qui elles sont destinées. Une interface de bureautique destinée à une secrétaire ne tien pas en compte les mêmes considérations de réalisation d'une interface de CAO destinée à un ingénieur.

Le profil de l'utilisateur est fondamental. Certains échecs célèbres ont montré qu'un système dont les interfaces sont mal adaptées peut entraîner non seulement le rejet du système d'information mais des conséquences commerciales et économiques pour l'entreprise.

Quelques règles élémentaires simples doivent guider la conception de toute interface :

1. Ne pas surcharger l'écran de données inutiles ou périmés (dont la raison d'être a l'écran ne se justifier pas). Par exemple, certains permettent l'ouverture d'un grand nombre de fenêtres. Or a un moment donné seules quelque fenêtres sont utilisables. La non fermeture des autres fenêtres « pollue » l'écran et empêche la focalisation sur les fenêtres essentielles.

2. Ne pas utiliser la mémoire de l'utilisateur comme composant de l'interface, il faut entre autre éviter a l'utilisateur le souci de se souvenir d'une commande ou d'une touche de fonction. La non standardisation des rôles des boutons de la souris est également un élément déroutant dans la mise en œuvre de celle-ci.

Donc, le concept de métaphore doit être judicieusement utilisé pour rendre l'interface intuitive et prévisible, son enrichissement avec des couleurs et des motifs ronds plus facile la mémorisation et plus immédiate l'interaction. Sur le plan méthodologique, il est important de séparer l'interface de l'application et de mieux contrôler la communication entre ces deux composants d'un système d'information.

2. L'architecture générale du système :

L'architecture logicielle d'un système est un ensemble de : décisions d'organisation du système, choix des éléments structurels qui composent le système et comportement de ces éléments structurels.

Dans le cadre de notre projet, l'architecture du système est composée de deux modules essentiels qui sont :

- Module 1 : L'éditeur graphique.
- Module 2 : L'IHM (l'interface utilisateur).

2.1. Module 1 : L'éditeur graphique :

L'éditeur graphique est le composant primordial, car il constitue le cœur de toute opération de modélisation de diagramme, il offre à l'utilisateur du système la possibilité spécifier tous les vues de son projet et pour chaque vue tous les diagrammes qu'elle contient et pour chaque diagramme tous ces composants rapidement et de manière graphique en lui offrant les outils nécessaires qui sont :

- Une palette de dessin qui contient un ensemble de boutons pour dessiner des classes, paquetages, cas d'utilisation, objets, acteurs, notes, relations de différents types,... etc.
- Une zone de dessin sur laquelle l'utilisateur peut placer les éléments à dessiner.

L'éditeur graphique offre à l'utilisateur du système plusieurs fonctionnalités qui lui facilitent la tâche, parmi ces fonctionnalités on peut citer :

- ✓ La possibilité de déplacer les composants créer tout au long de la zone de dessin.
- ✓ La possibilité de supprimer des composants et les relations entre composants existants.
- ✓ La possibilité d'importer des composants à partir d'autres diagrammes de même type que le diagramme destination dans le but de les réutiliser directement sans perdre de temps dans leur redéfinition (c'est une opération de copier/coller mais entre diagrammes).
- ✓ La possibilité de créer des relations entre composants graphiquement et en cas de fausses relations effectuer une correction automatique qui consiste a la suppression des ces relations.
- ✓ La possibilité de visualiser une relation choisit dans le cas d'avoir plusieurs relations entre deux composants.

2.1.1. Modélisation du module « éditeur graphique » :

1) Le comportement statique :

Pour visionner les parties statiques de ce module nous allons utilisés l'un des diagrammes structurels du langage UML qui serve à visualiser, spécifier et documenter les aspects statiques qui est le diagramme de classe.

Les diagrammes de classes représentent un ensemble de classes et d'interfaces ainsi que leurs relations [6], ce sont des diagrammes que l'on utilise pour illustrer la vue conception statique d'un système.

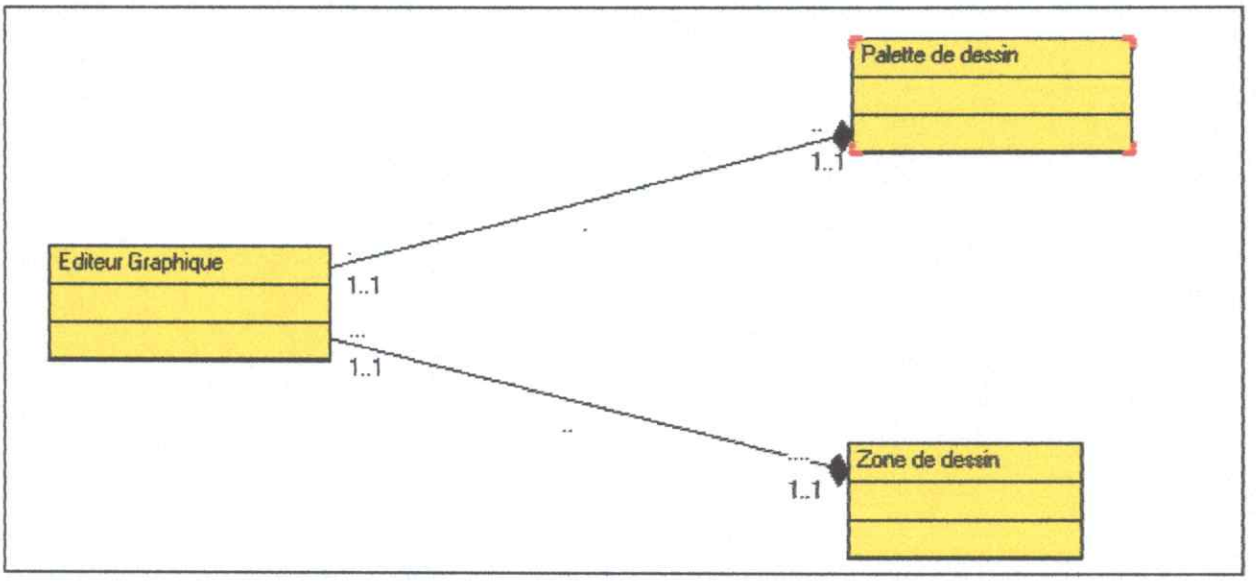


Figure 1: les éléments du module éditeur graphique.

2) Le comportement dynamique :

Pour modéliser les aspects dynamiques du module éditeur graphique nous allons utilisés un diagramme comportementaux du langage UML qui est le diagramme de séquence.

Le diagramme de séquence est un diagramme d'interaction qui met en évidence l'ordre chronologique des messages [6], il représente un ensemble d'objet et d'acteurs ainsi que les messages envoyés et reçus par ces objets et acteurs, ce diagramme est utilisé pour modéliser la vue conception dynamique d'un système.

Les principales opérations réalisées par le module « éditeur graphique » sont :

- **Création d'une vue :**

  La création d'une vue passe par les étapes suivantes :

  1. L'utilisateur clique sur le nom du projet (avec le bouton droit).
  2. Le système affiche le PopUpMenu projet.

3. L'utilisateur demande de créer une nouvelle vue.
4. Le système ajoute la vue.

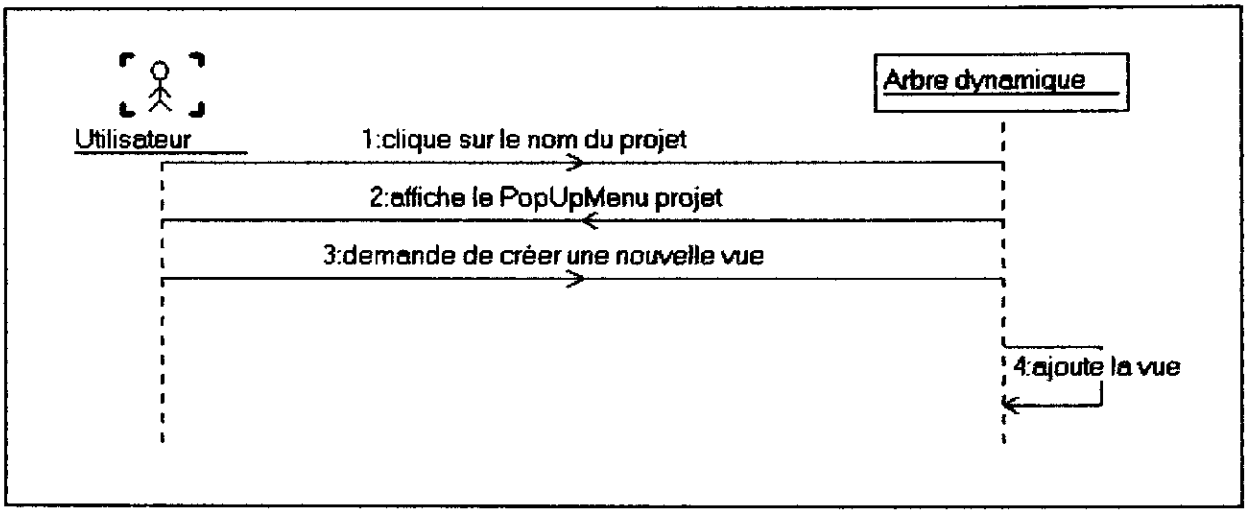


Figure 2 : Diagramme de séquence pour l'opération «Création d'un vue».

- **Création d'un diagramme :**

La création d'un diagramme se passe comme suit :

1. L'utilisateur clique sur le nom d'une vue (avec le bouton droit).
2. Le système affiche le PopUpMenu vue.
3. L'utilisateur demande de créer un nouveau diagramme.
4. Le système ajoute le diagramme et l'affiche.

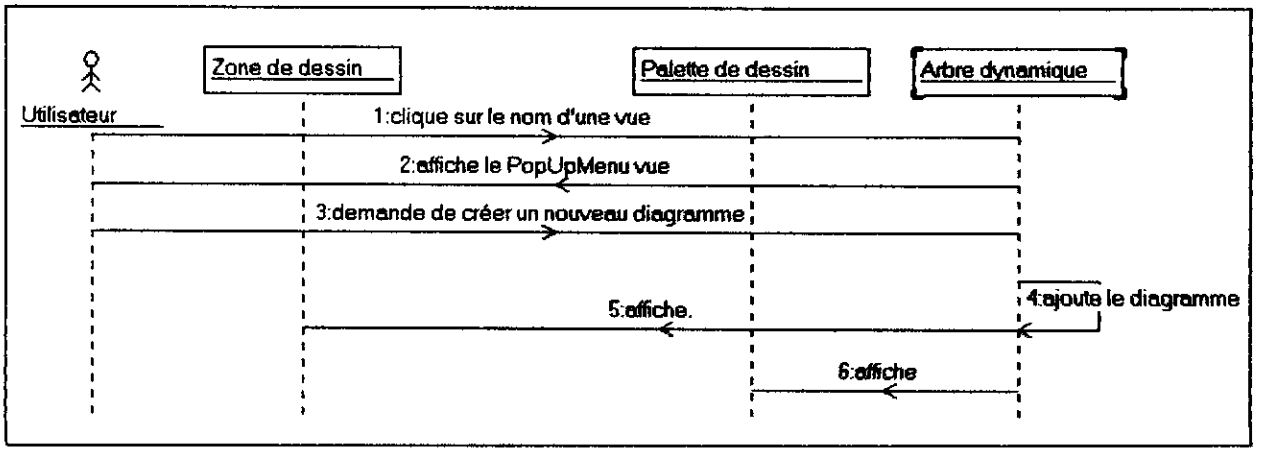


Figure 3 : Diagramme de séquence pour l'opération «Création d'un diagramme».

- **Création d'un Composant :**

La création de composant se fait graphiquement de la manière suivante :

1. L'utilisateur choisit le bouton correspondant au composant à dessiner.
2. Déplace la souris sur la zone de dessin.
3. Clique avec la souris sur l'endroit où il veut placer le composant.
4. Le système dessine le composant.

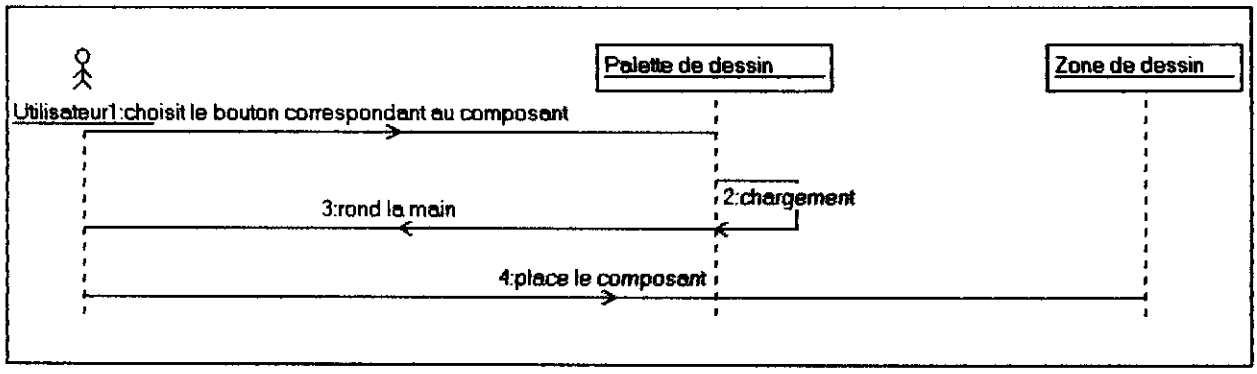


Figure 4 : Diagramme de séquence pour l'opération «Création d'un Composant».

- **Création d'une Relation entre Composants :**

La création d'une relation entre composants est une opération très importante, car la relation est l'interface de communication entre composants. La création se fait de la manière suivante :

1. L'utilisateur choisit le bouton correspondant a la relation a crée.
2. Choisit le composant source de la relation.
3. Choisit le composant destination de la relation.
4. Le système vérifier la justesse de la relation.
5. Si fausse, le système annule cette relation.
6. Si juste, le système demande de donner un nom à la relation.
7. L'utilisateur définit le nom de la relation.
8. Le système crée la relation.

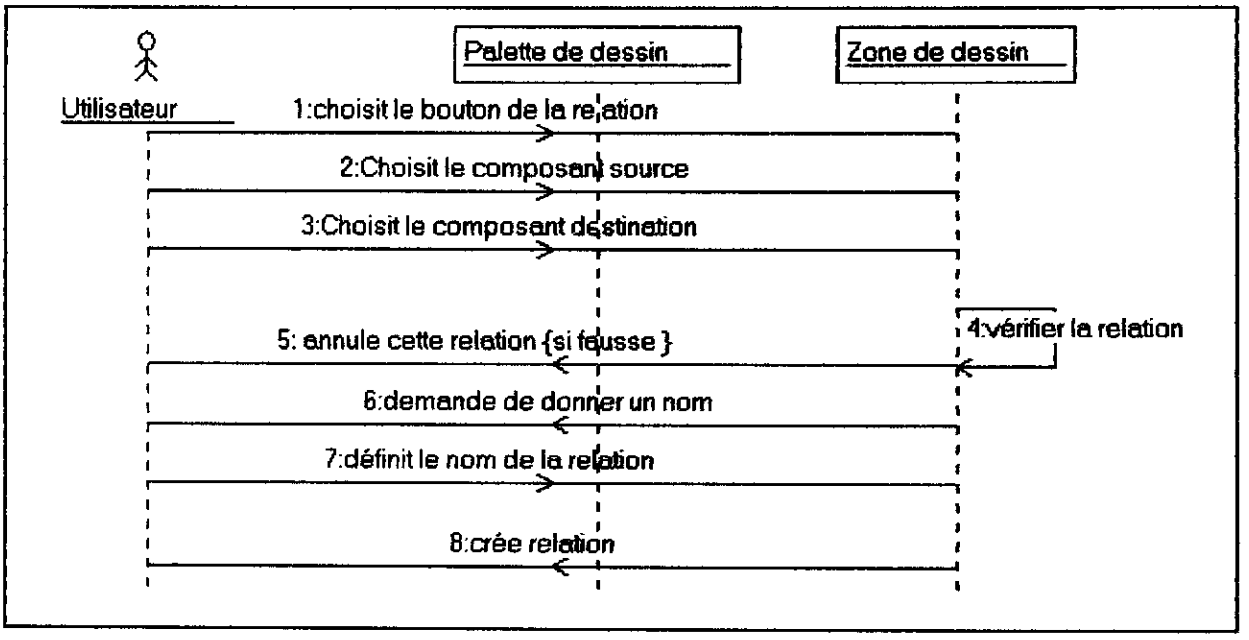


Figure 5: Diagramme de séquence pour l'opération «création d'une relation entre composants».

- **Edition d'un Composant :**

  L'édition d'un composant consiste à définir les propriétés du composant. Ces propriétés diffèrent suivant le type du composant, par exemple :

  Les propriétés d'une classe sont : son nom, ses attributs et ses méthodes, par contre pour un paquetage, un cas d'utilisation ou un acteur la seule propriété est le nom.

  L'édition d'un composant s'effectue de la manière suivante :

  1. L'utilisateur clique avec la souris sur un composant (avec le bouton droite).
  2. Le système affiche le PopUpMenu composant.
  3. L'utilisateur demande d'éditer le composant.
  4. Le système affiche la boite de dialogue «éditer composant» (suivant le type du composant).
  5. L'utilisateur définit les propriétés du composant.
  6. Le système vérifier la justesse des propriétés.
  7. Si faussent demande de les corriger.
  8. Le système ajoute ces propriétés au composant.

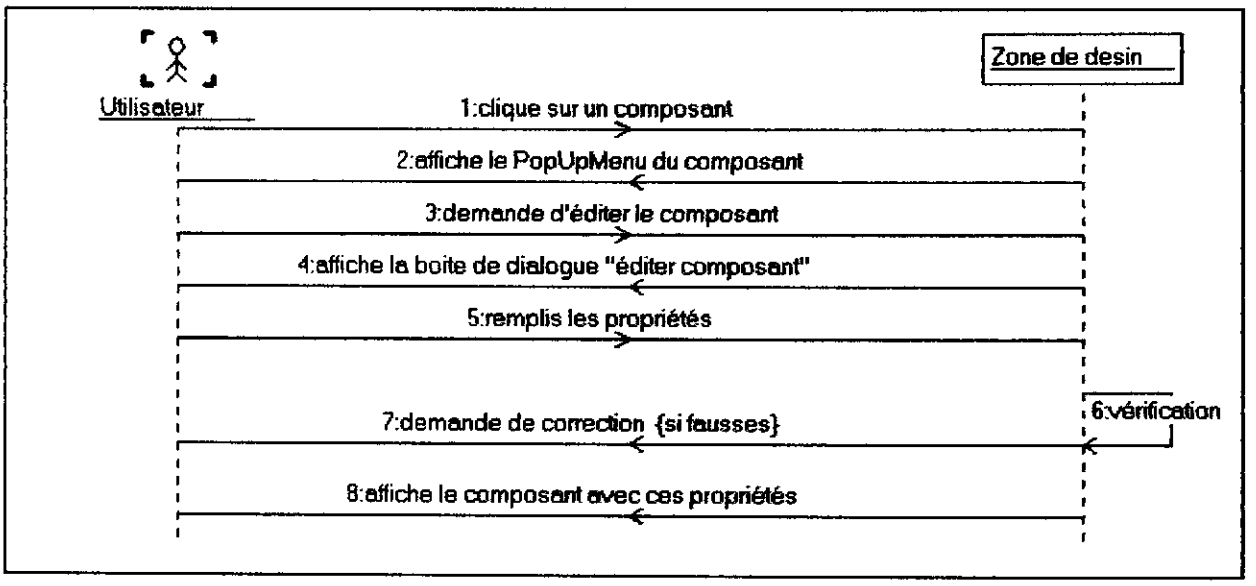


Figure 6: Diagramme de séquence pour l'opération «édition d'un composant».

- **Edition d'une relation :**

L'édition d'une relation consiste à définir les propriétés de la relation qui sont : son nom, les cardinalités des deux cotés de la relation, les rôles de deux individus qui composent la relation et un commentaire pour expliquer la signification de la relation si on veut.

L'édition d'une relation se passe de la manière suivante :

1) L'utilisateur clique avec la souris sur une relation (avec le bouton droite).
2) Le système affiche le PopUpMenu relation.
3) L'utilisateur demande d'éditer la relation.
4) Le système affiche la boite de dialogue «éditer relation».
5) L'utilisateur définit les propriétés de la relation.
6) Le système vérifier la justesse des propriétés.
7) Si faussent demande de les corriger.
8) Le système ajoute ces propriétés à la relation.

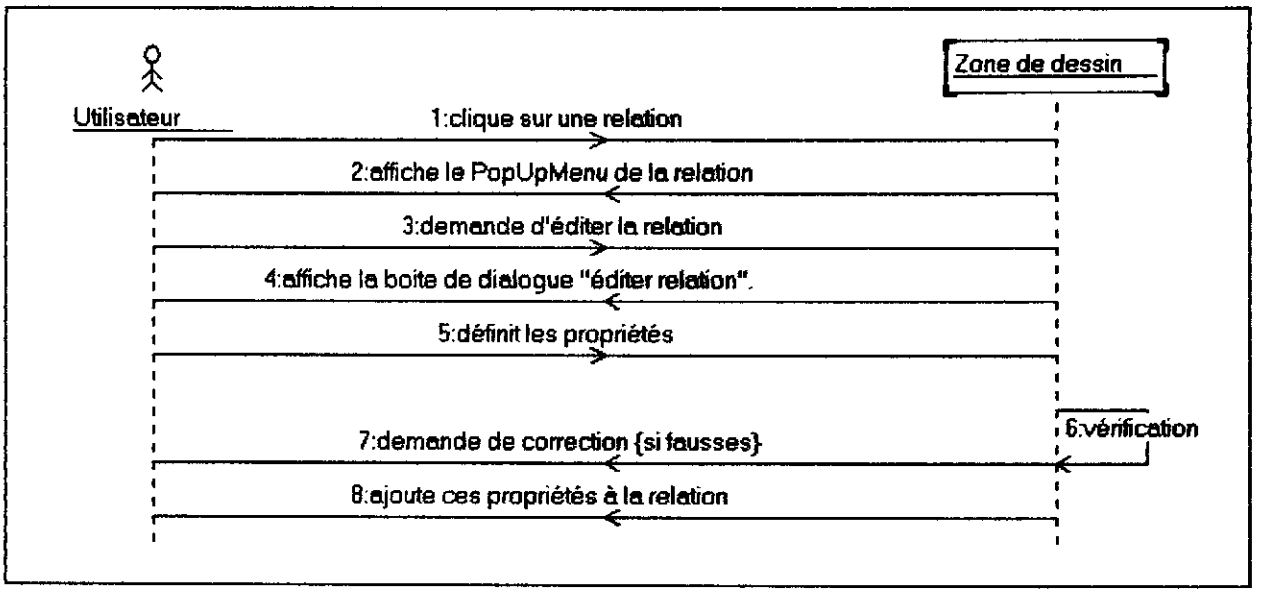


Figure 7: Diagramme de séquence pour l'opération «édition d'une relation».

- **Création d'un message :**

Les messages sont les liens qui reliés les objets et les acteurs dans un diagramme de séquence.

La création d'un message se passe de la manière suivante :

1) L'utilisateur clique sur le bouton correspondant au message (il y a deux types de messages).
2) Choisit l'objet ou l'acteur source du message.
3) Choisit l'objet ou l'acteur destinateur du message.
4) Le système demande de donner le message à afficher.
5) L'utilisateur tape le message.
6) Le système crée le message.

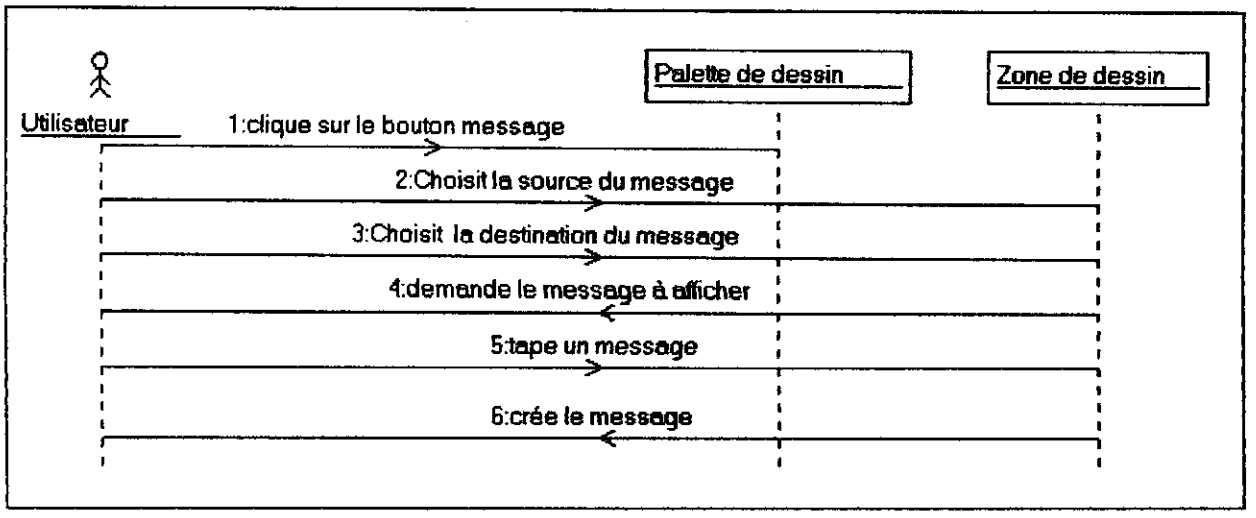


Figure 8: Diagramme de séquence pour l'opération «création d'un message».

- **Suppression d'un composant :**

La suppression d'un composant est une opération importante dans notre éditeur car souvent on fait des erreurs de création (par exemple créer une classe ou un objet jamais utilisé), alors on est obliger de supprimer ce composant inutile.

La procédure de suppression d'un composant se passe de deux manières comme suit :

Soit après : sélection du composant et clique sur le bouton suppression dans la palette de dessin ou clique sur le composant dans la zone de dessin (avec le bouton droit) et choix dans le PopUpMenu composant de la suppression.

Ces deux manières de suppression passent par la même séquence, à l'exception du membre déclencheur de la suppression. Donc nous allons donner un diagramme de séquence pour un de ces cas qui est le premier, pour l'autre le changement est minimal.

Les étapes de la suppression de composant sont :

1. L'utilisateur sélectionne le composant à supprimer sur la zone de dessin.
2. Clique sur le bouton supprimer dans la palette de dessin.
3. Le système supprime les relations dans lesquelles le composant est membre.
4. Le système supprime le composant de l'arbre dynamique.
5. Le système supprime le composant.

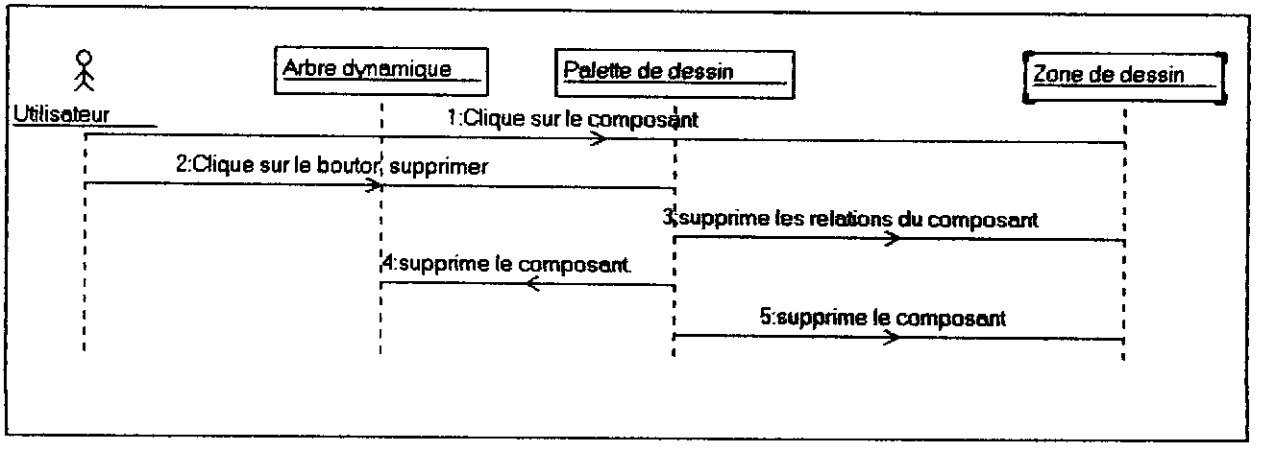


Figure 9: Diagramme de séquence pour l'opération «suppression d'un composant».

- **Suppression d'une relation :**

Comme la suppression d'un composant, la suppression d'une relation est importante.

Les étapes de la suppression d'une relation sont :

1. L'utilisateur clique sur la relation à supprimer (avec le bouton droit) sur la zone de dessin.
2. Le système affiche le PopUpMenu relation.
3. L'utilisateur demande de supprimer la relation.
4. Le système supprime la relation de la zone de dessin.

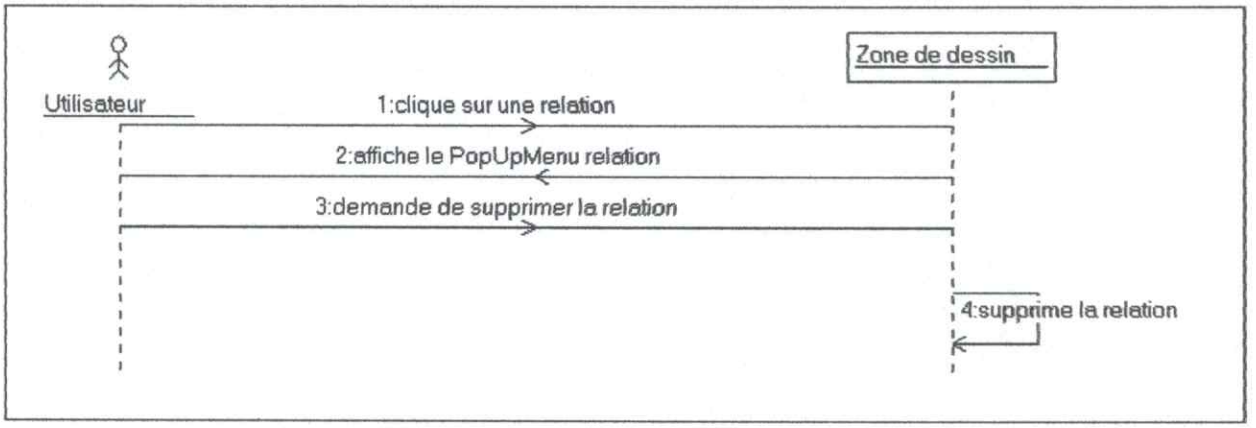


Figure 10: Diagramme de séquence pour l'opération «suppression d'une relation».

- **Sauvegarde de projet ou diagramme :**

L'opération de sauvegarde est une opération très importante dans le cas de notre logiciel, elle permet de stocker les différents projets et diagrammes dans des fichiers avec l'extension :

- '**.prj**' pour les projets.
- '**.cla**' pour les diagrammes de classes.
- '**.cut**' pour les diagrammes de cas d'utilisation.
- '**.seq**' pour les diagrammes de séquences.

Cette opération s'effectue comme suit :

1. L'utilisateur demande de sauvegarder un projet ou un diagramme.
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 projet ou le document.

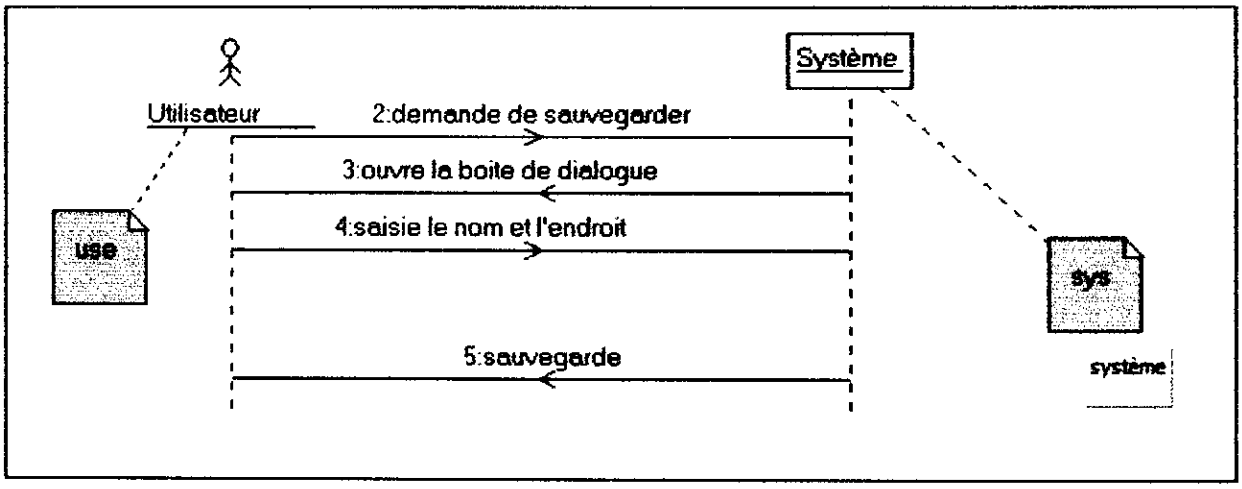


Figure 11: Diagramme de séquence pour l'opération «sauvegarde de projet ou diagramme».

2.2. Module 2 : L'IHM (interface utilisateur) :

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

- **L'arbre dynamique :**

C'est une zone qui décrit l'arborescence des vues et des diagrammes et tous les composants que l'utilisateur à crée durant le processus de développement de son projet (elle permet de voir la hiérarchie du projet).

Cet arbre sert d'une part à donner un aperçu de toutes les vues du projet, de tous les diagrammes d'une vue et tous les composants d'un diagramme. D'autre part, chaque nœud de cette arborescence sert comme interface entre l'utilisateur et le système pour lui offrir l'essentiel des fonctionnalités telles que :

- L'ajout et la suppression d'une vue du projet.
- L'ajout et la suppression d'un diagramme de la vue.

- La suppression d'un composant d'un diagramme.
- Affichage et modification des propriétés d'un composant.

- **Menu :**

C'est une barre de menus qui contient des raccourcis pour réaliser quelques opérations tel que : ouvrir projet, enregistrer, enregistrer sous, organiser les fenêtres, couper, coller, copier...etc.

- **Barre d'outils :**

Cette barre d'outils porte plusieurs boutons, ces boutons sont répartis selon leurs fonctions en trois catégories :

- Les boutons de la première catégorie offrent des traitements sur les fichiers projet et diagrammes (ouverture, enregistrement...etc.)
- Les boutons de la deuxième catégorie offrent des opérations sur le projet et le diagramme eux mêmes (copier composant, coller, couper...).
- Les boutons de la troisième catégorie sont spécifiques à chaque type de diagramme et ils apparaissent avec le diagramme et se cache avec aussi. C'est avec ces boutons qu'on peut dessiner les composants et les liens entre composants de chaque type de diagramme (exemple : les boutons classes, agrégation apparaissent avec les diagrammes de classes et les boutons objet, message avec le diagramme de séquence,...).

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

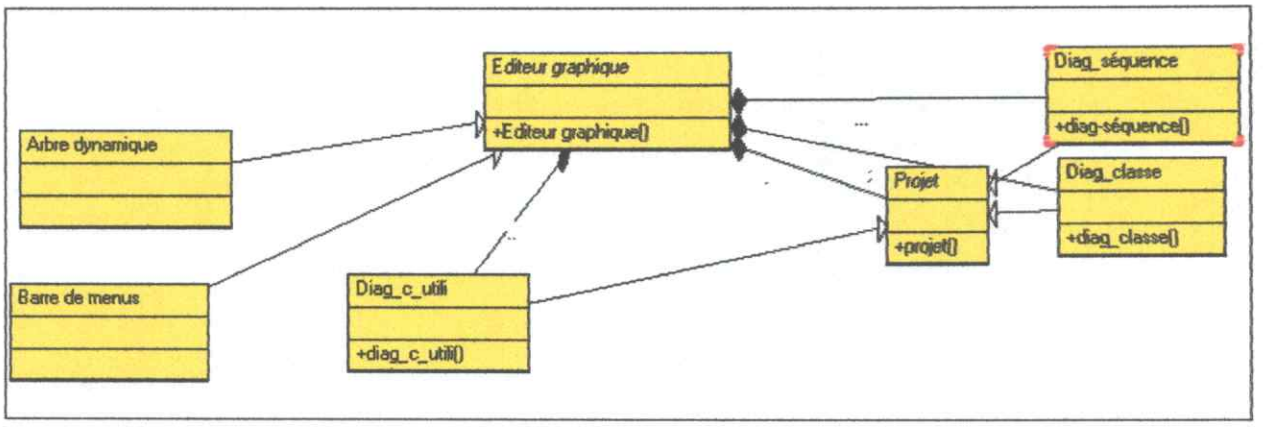


Figure 11: Diagramme de classe générale (vue générale sans attributs).

4. Diagrammes d'activités générales :

Les diagrammes d'activité représentent le flot d'activité à l'intérieur d'un système, ils sont importants dans la modélisation des fonctions du système [6], c'est pour cette raison que nous l'utiliserons pour modéliser les différents flots d'activités au sein de notre système développé.

Chaque activité est placée dans une travée afin de diviser les activités en groupes 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 :**

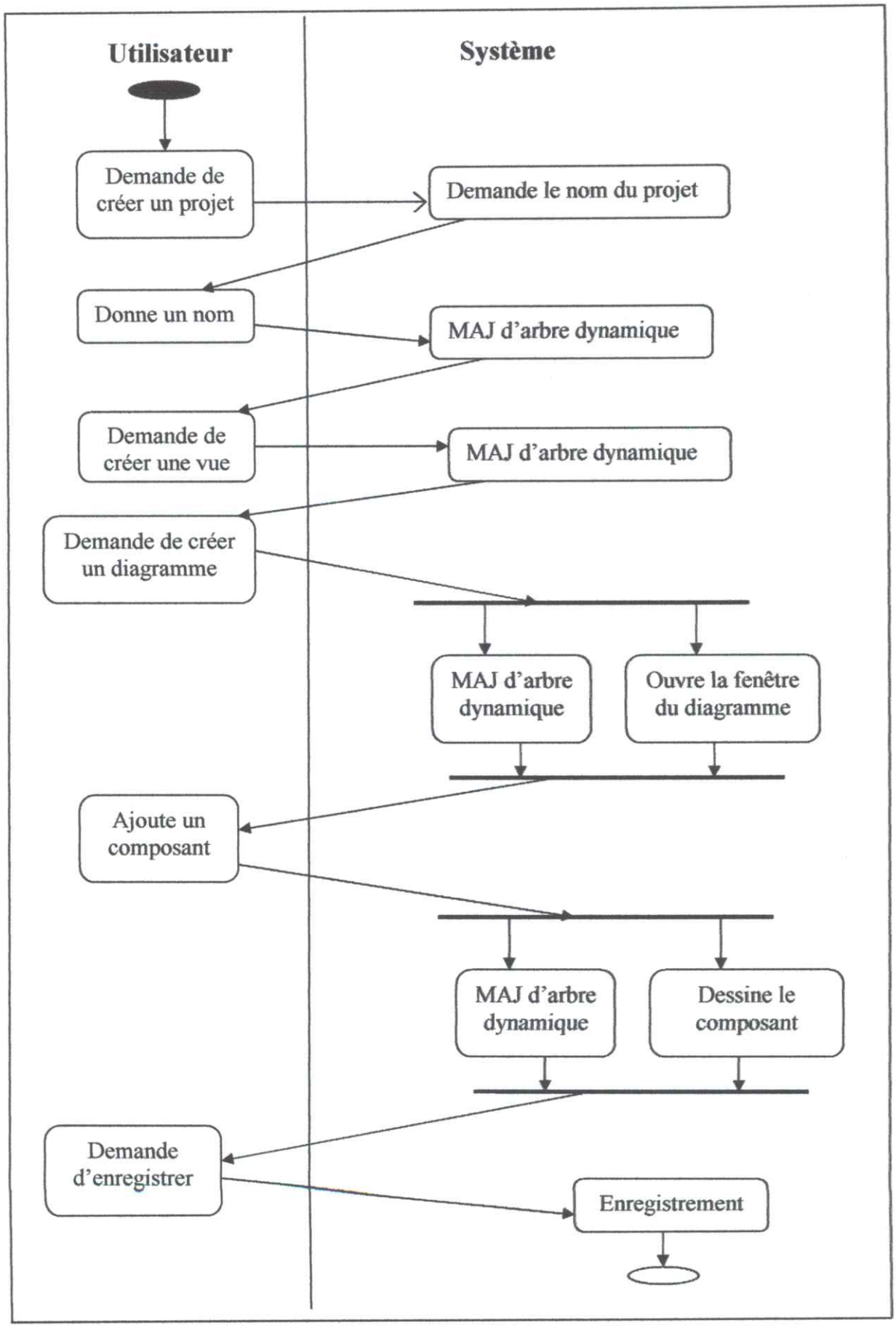


Figure 12: Diagramme de d'activité pour l'opération «Création d'un composant».

- **Créer une Relation entre Composants :**

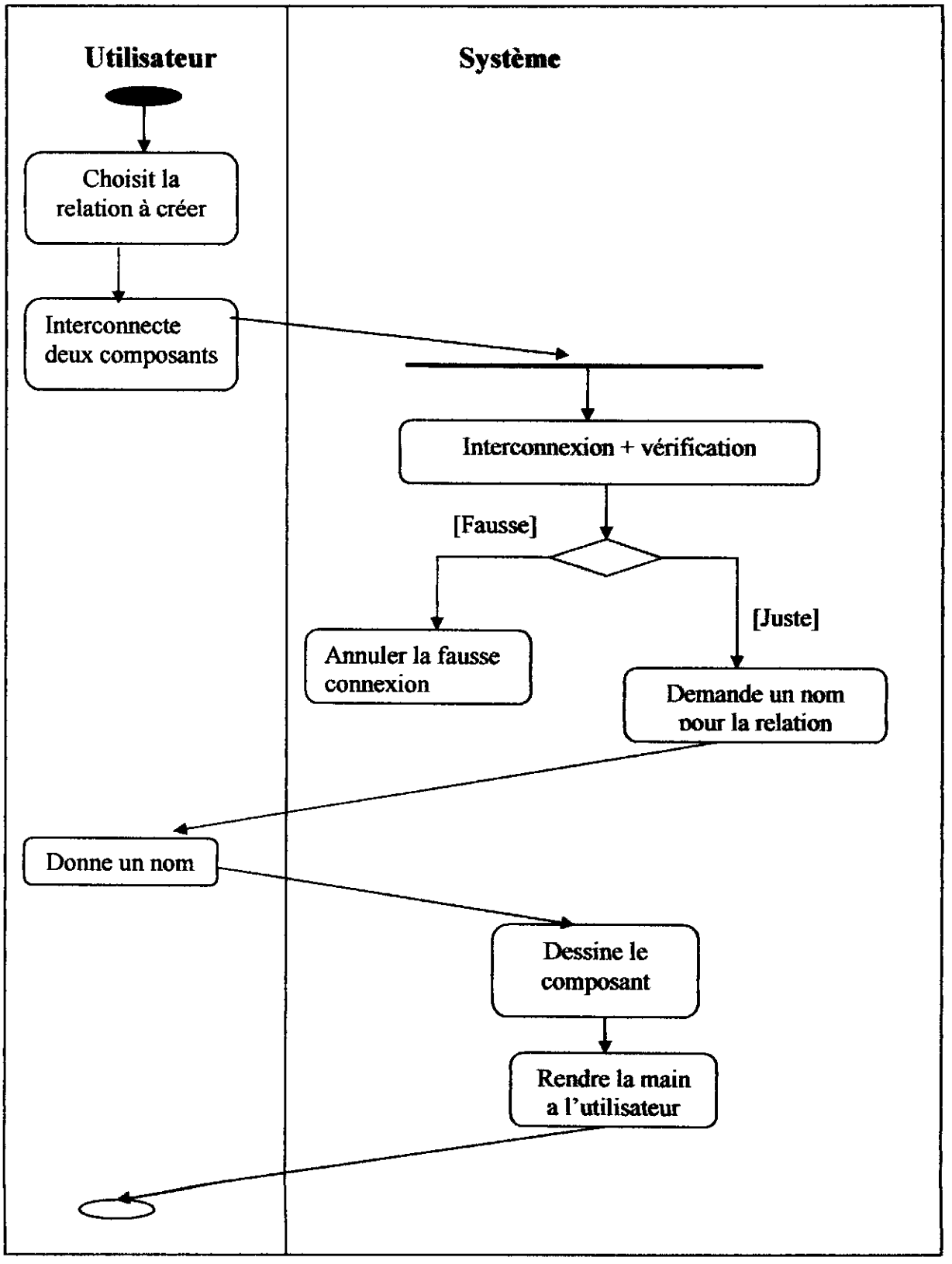


Figure13 : Diagramme de d'activité pour l'opération «Création d'une relation entre composants».

- **Editer un composant :**

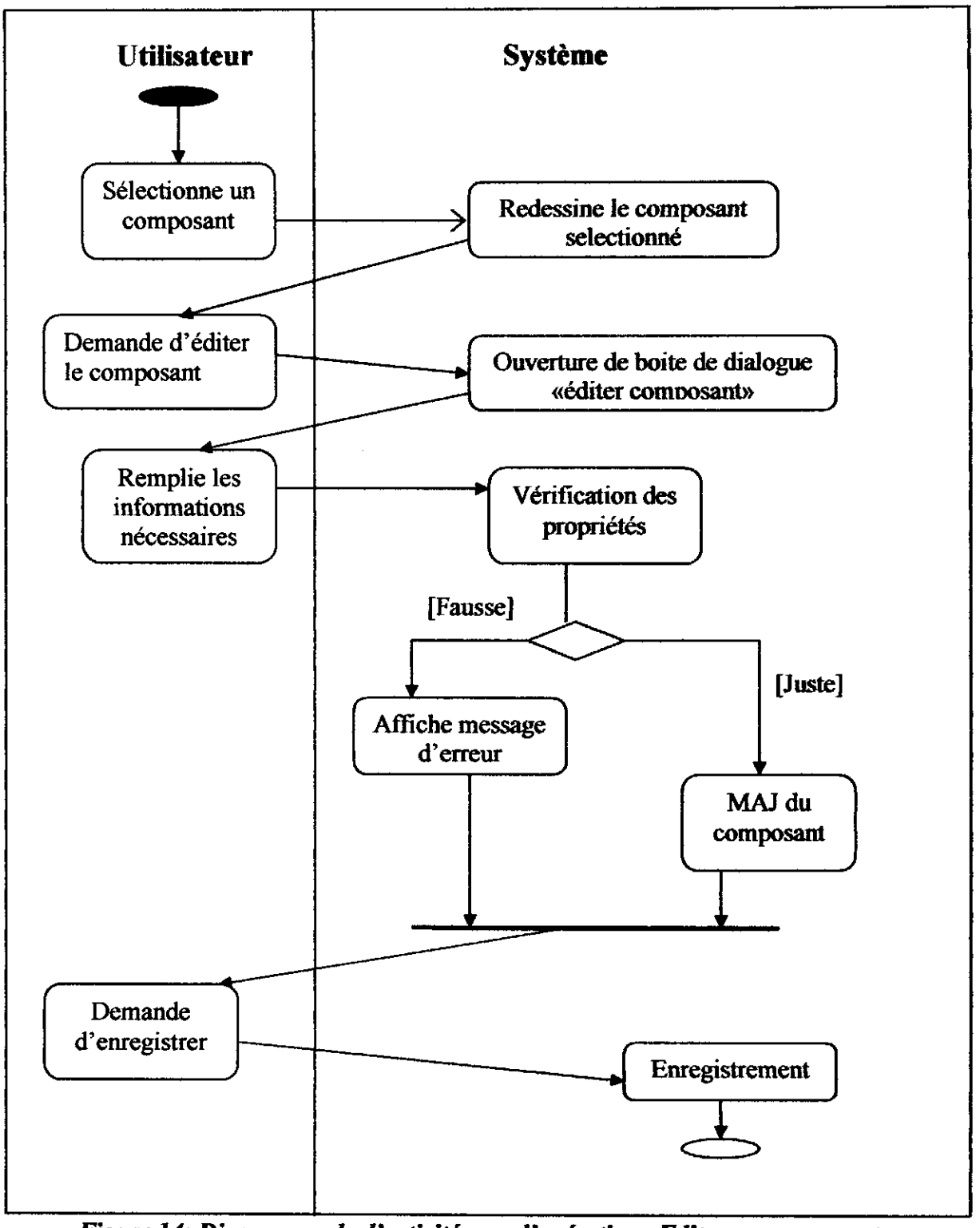


Figure 14: Diagramme de d'activité pour l'opération «Editer un composant».

- **Editer une relation :**

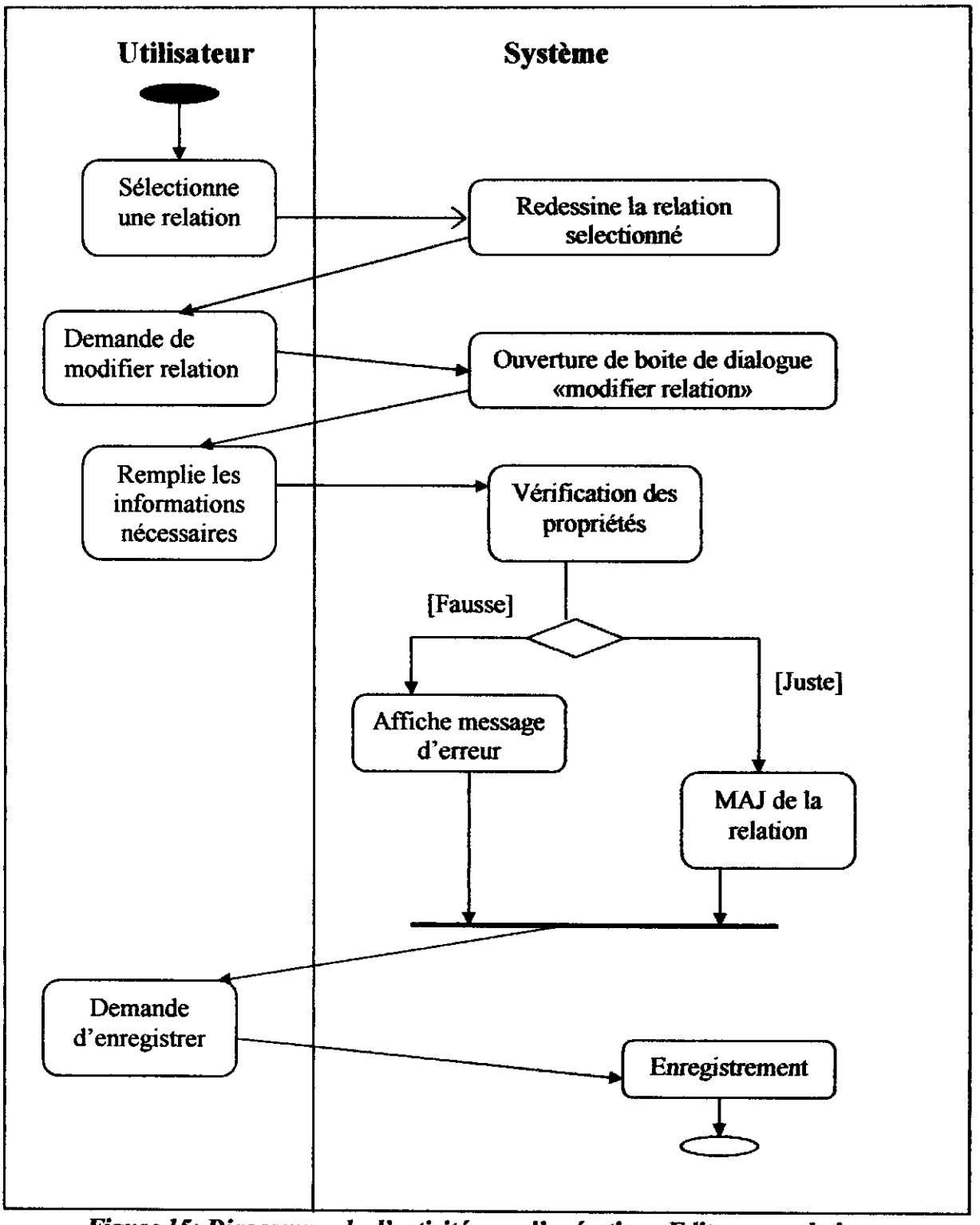


Figure 15: Diagramme de d'activité pour l'opération «Editer une relation».

- **Importer un composant :**

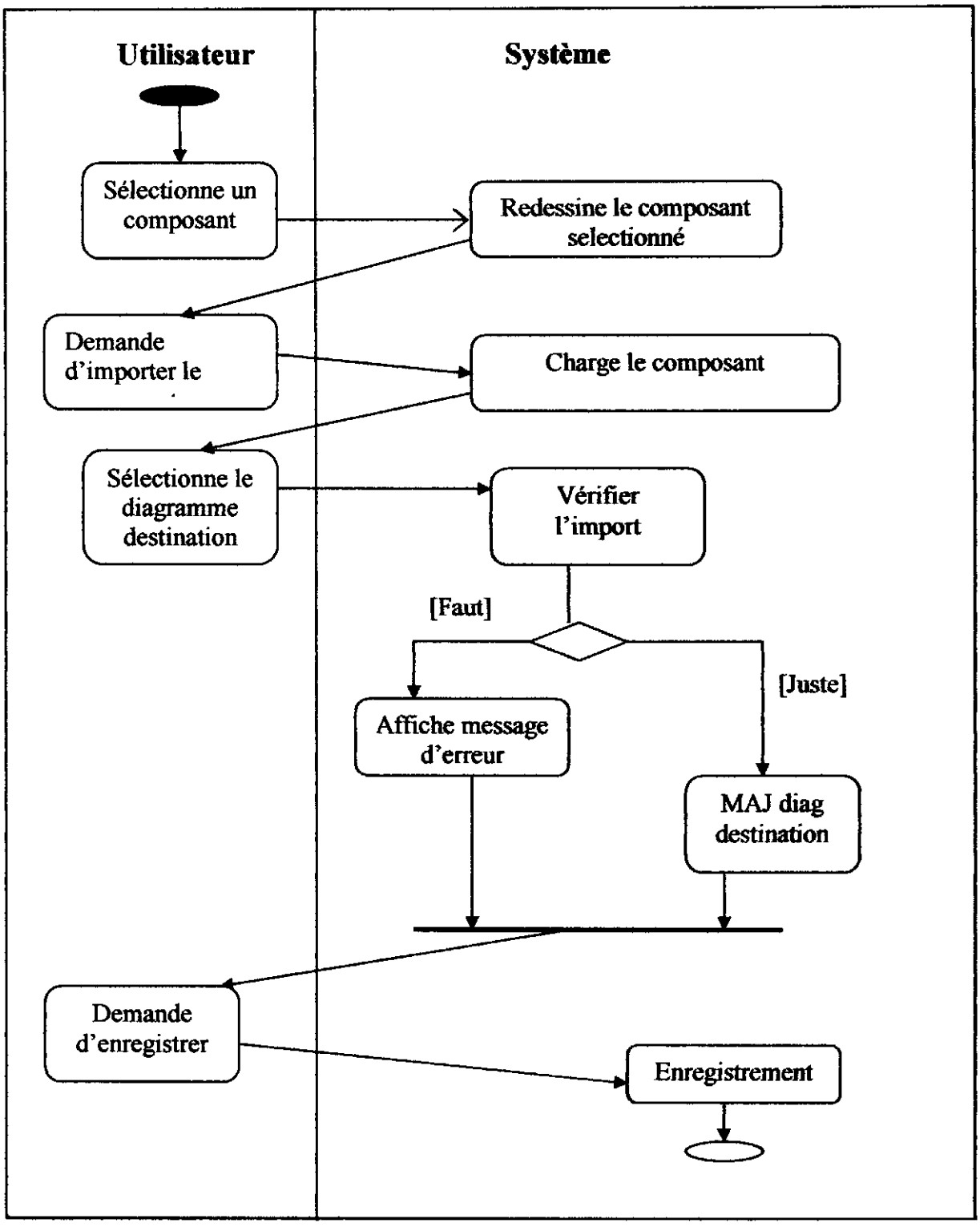


Figure 16: Diagramme de d'activité pour l'opération «Importer un Composant».

CHAPITRE VI

L'implémentation

1. Introduction :

Cette phase correspond à la programmation proprement dite des fonctions sur la base des informations venant de la phase de conception [8].

Cette partie sera organisée comme suit : nous commençons par décrire l'environnement de programmation, ensuite nous allons décrire comment notre logiciel est implémenté en citant les principales classes et les principales méthodes fournies par chaque classe.

2. Le langage c++ Builder : [reisdorph 98][7]

2.1. Introduction :

Le langage C++ est une « amélioration » du langage C (le langage c a été mis au point par **M.Ritchie** et **B .W.Kernighan** au début des années 70). Bjarne Stroustrup, un ingénieur considéré comme l'inventeur du c++, a en effet décidé d'ajouter au langage C les propriétés de l'approche orienté objet. Ainsi, vers la fin des années 80 un nouveau langage baptisé C with classes (traduisez « C avec des classes »), apparaît. Celui-ci a ensuite a été renommé en C++, clin d'œil au symbole d'incrémentation ++ du langage C, afin de signaler qu'il s'agit du langage C amélioré (langage C+1).

2.2. Les améliorations de C++ :

Le C++ reprend la quasi intégralité des concepts présents dans le langage C, si bien que les programmes écrits en langage C fonctionnent avec un compilateur C++.

En réalité le langage C++ est un sur-ensemble du C, il y ajoute, entre autres, les fonctionnalités suivantes :

- L'héritage (simple et multiple).
- Le polymorphisme.

Ainsi qu'un ensemble de nouvelles fonctionnalités, parmi lesquelles :

- Le contrôle de type.
- Les arguments par défaut
- La surcharge de fonctions.

2.3. Le fichier source :

Le fichier source d'un programme écrit en langage C++ est un simple fichier texte dont l'extension est par convention .CPP.

Lorsque le programme est prêt a être «essayé», il s'agit de le compiler (le traduire en langage machine) a l'aide d'un compilateur.

Il existe des environnements fournissant un éditeur de texte, un éditeur de liens et un compilateur. Ces environnement sont appelé EDI (Environnement de Développement Intégré).les principaux EDI permettant le développement d'application en langage C++ sont :

- Inprise Borland C++.
- Inprise Borland C++ Builder.
- Microsoft Visual C++.

2.4. Aspect d'un programme en C++ :

Un programme écrit en langage C++, rassemble beaucoup a un programme écrit en C, a la différence prés qu'il contient essentiellement des classes. Il comporte ainsi un fonction principale appelée Main( ) renfermant les instructions qui doivent être exécutées. Celles-ci sont comprises entre des accolades qui suivent le nom de la fonction. Cela vous semble tombé du ciel si vous n'avez jamais programmé en C++, mais il faut admettre pour l'instant la manière d'écrire un programme en C++.

2.5. Pourquoi C++ Builder ?

2.5.1. Introduction :

En choisissant le plus innovant des outils de programmation, il est évidant que les fonctions du C++ Builder sont mises à notre disposition. Ce nouveau produit de développement rapide d'applications (RAD) de Borland pour la réalisation d'applications en

C++. Avec C++ Builder vous développez des programmes en C++ pour Windows avec plus de facilité et de rapidité qu'au paravent.

C++ Builder regroupe toute la puissance C++ dans un environnement RAD, cela signifie que vous pouvez créer instantanément l'interface utilisateur d'un programme (c'est-à-dire les menus, les boites de dialogues, les fenêtres principales,... etc.) a l'aide de la technique de glisser-déposer. En effet C++ Builder réalise pour vous une quantité de tâches de base qui constituent le cœur d'un programme Windows, mais il ne peut pas écrire le programme a votre place. Donc l'avantage de C++ Builder est de rendre la tâche plus facile et même agréable.

2.5.2. Aperçu rapide d e l'EDI C++ Builder :

Vous êtes censé suffisamment compétant en la matière pour lancer C++ Builder. Lorsque vous démarrez le programme, vous obtenez a la fois une fiche vierge et l'EDI comme le montre **la figure1**.

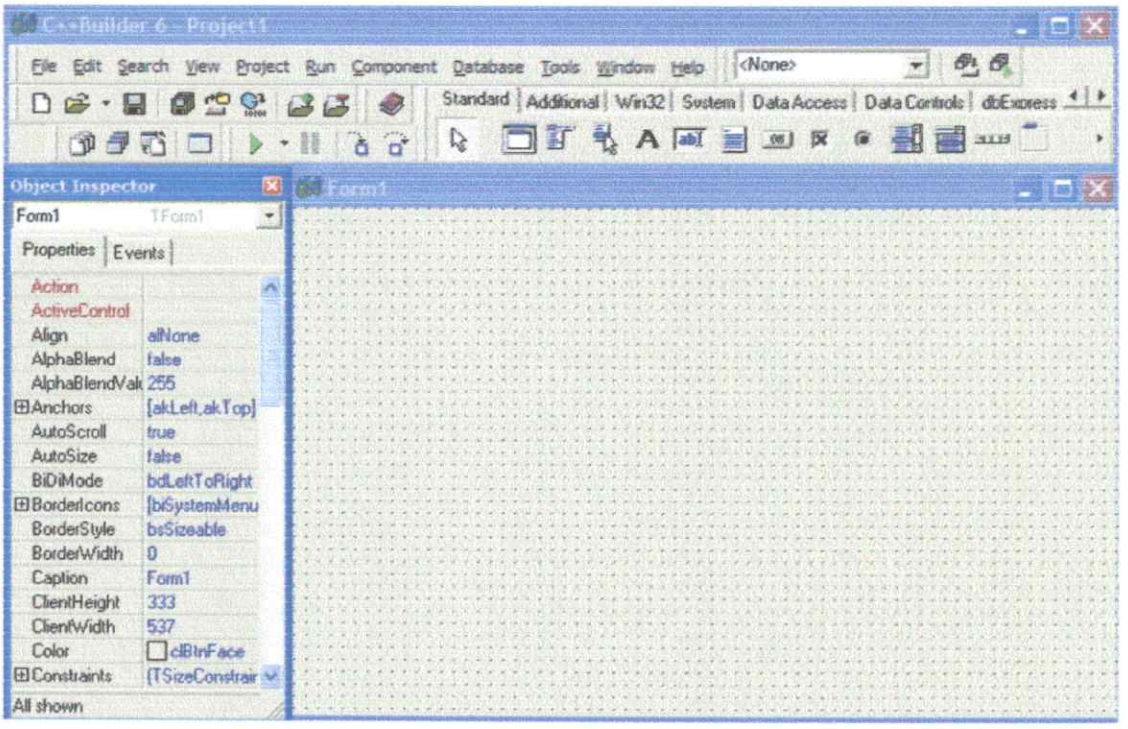


Figure1 : L'interface EDI C++ Builder et la fiche vierge apparaissant au démarrage.

L'EDI (Environnement de Développement Intégré) C++ Builder se compose de trois éléments :

1. La fenêtre supérieure est considérée comme la fenêtre principale, elle contient la barre d'outils a gauche et la palette des composants a droite.

La barre d'outils vous permet d'accéder par simple clic à différentes tâches comme l'ouverture, l'enregistrement et la compilation de projets.

La palette des composants contient une grande quantité de composants que vous pouvez déposer sur vos fiches (les composants sont les libellés (Label), zone de saisie (Edit), zones de listes (ListBox), les boutons (Button) et autres chose de ce style).

Par soucie de simplification, les composants sont répartis en groupes. Pour placer un composant sur la fiche il vous suffit de cliquer sur le bouton correspondant de la palette des composants, puis de cliquer dans votre fiche à l'endroit voulu.

2. Au dessous de la barre d'outils et de la palette des composant, dans l'angle supérieur gauche de l'écran, se trouve l'inspecteur d'objets, c'est à l'aide de cet inspecteur d'objets que vous modifier les évènements et les propriétés d'un composant. La propriété d'un composant contrôle la façon dont il opère. La liste des propriétés varie d'un composant a un autre, bien que la plupart d'entre eux aient généralement plusieurs éléments communs (exemple : Height et Width).

3. A droite de l'inspecteur d'objet se trouve l'espace de travail C++ Builder. Il comprend à l'origine le concepteur de fiches. Dans C++ Builder, une fiche représente une fenêtre dans le programme. La fiche peut être la fenêtre principale, une boite de dialogue ou tout autre type de fenêtres. Vous utilisez le concepteur de fiches pour insérer, déplacer et dimensionner les composants lors de la phase de conception de la fiche. Derrière le concepteur de fiches de cache l'éditeur de code. C'est dans l'éditeur de code que vous tapez le texte lorsque vous écrivez vos programmes.

L'inspecteur d'objets, le concepteur de fiches, l'éditeur de texte et la palette des composants fonctionnent interactivement lorsque vous construisez vos applications.

4. Implémentation de notre logiciel :

Nous avons vu précédemment que notre logiciel est constitue de deux modules qui sont l'éditeur graphique et l'interface utilisateur. Dans cette phase nous allons décrire comment chaque module est implémenté.

L'éditeur graphique :

Comme on l'a déjà cité, ce module offre a l'utilisateur du système la possibilité de spécifier ses projets et les diagrammes de ses projets de manière graphique en lui offrant un ensemble d'outils.

Pour implémenter ce module nous avons crées un ensemble de classes. Dans ce qui suit nous allons cités les principales classes et leurs méthodes :

- **Classe individu :**

C'est la principale classe pour les composant : tous les composants héritent de cette classe (par exemple classe, actor, use case, objet...).

Les principales méthodes fournies par cette classe sont les suivantes :

1) **AnsiString Get_name():** retourne le nom de l'individu.
2) **TPoint Get_Point() :** retourne le point du centre de l'individu.
3) **void Set_name(AnsiString n) :** permet de définir le nom de l'individu.
4) **void Set_XY(int x,int y):** permet de définir la position de l'individu.

- **Classe Relation :**

C'est la principale classe pour les relations : tous les relations héritent de cette classe (comme dépendance, agrégation, héritage, branche note,...).

- **Classe classdiagramm :**

C'est une classe interface qui est responsable de la création de tous les éléments graphiques du diagramme de classe (comme les classes, paquetages, agrégations,...).

Les principales méthodes fournies par cette classe sont :

1) Méthode Draw (individu I,Bool selected) : c'est elle qui dessine les composants. Les paramètres sont l'individu à dessiner et un booléen indiquant si l'individu est sélectionné.

Voici un extrait du code de la méthode Draw pour le composant classe :

```
// Pour dessiner la classe
{ drawzone->Canvas->Brush->Color = clYellow;  drawzone->Canvas->Pen->Width = 1;
drawzone->Canvas->Pen->Color = clBlack;  drawzone->Canvas->Rectangle(C.Get_Rect());
drawzone->Canvas->TextOut(C.Get_Rect().Left+ 5,C.Get_Rect().Top+2,C.Get_name());
drawzone->Canvas->MoveTo(C.Get_Rect().left,C.Get_Rect().Top+20);
drawzone->Canvas->LineTo(C.Get_Rect().Right,C.Get_Rect().Top+20);
int top = C.Get_Rect().top+22;  int left = C.Get_Rect().Left+5;
drawzone->Canvas->Font->Style <<fsBold<<fsUnderline;drawzone->Canvas->Font->Size= 8;
// Pour afficher les attributs
for(int i =0;i<C.Get_Attributs().Get_Cardinalite();i++) { AnsiString pre;switch(C.Get_Attributs()[i].visibilite)
{case Public : pre ="+";break;  case Private : pre ="-";break;  case Protected: pre ="#";break;  default: break; }
drawzone->Canvas->TextOut(left,top+i*15,pre+C.Get_Attributs()[i].nom);  }
int l;if(C.Get_Attributs().Get_Cardinalite()<2)l= 40; else l = 40+15*(C.Get_Attributs().Get_Cardinalite()-1);
drawzone->Canvas->MoveTo(C.Get_Rect().left,C.Get_Rect().Top+l);
drawzone->Canvas->LineTo(C.Get_Rect().Right,C.Get_Rect().Top+l); top = C.Get_Rect().Top+l+2;
// Pour afficher les méthodes
for(int i =0;i<C.Get_Methodes().Get_Cardinalite();i++) { AnsiString pre; switch(C.Get_Methodes()[i].visibilite)
{ case Public : pre ="+";break; case Private : pre ="-";break;case Protected: pre ="#";break; default: break; }
if(C.Get_Methodes()[i].taille)drawzone->Canvas->TextOut(left,top+i*15,pre+C.Get_Methodes()[i].nom+
"("+IntToStr(C.Get_Methodes()[i].taille)+")");else
drawzone->Canvas-> TextOut(left,top+i*15,pre+C.Get_Methodes()[i].nom+"()");}
drawzone->Canvas->MoveTo(C.Get_Rect().left,C.Get_Rect().bottom);
drawzone->Canvas->LineTo(C.Get_Rect().Right,C.Get_Rect().bottom);
drawzone->Canvas->MoveTo(x,y);drawzone->Canvas->LineTo(x-5,y);
drawzone->Canvas->MoveTo(x,y);drawzone->Canvas->LineTo(x,y+5)
;x = C.Get_Rect().left;y = C.Get_Rect().Bottom;drawzone->Canvas->MoveTo(x,y);
drawzone->Canvas->LineTo(x,y-5);}
```

Et voici un extrait pour branche :

```
{ drawzone->Canvas->Pen->Width = 1; drawzone->Canvas->Pen->Color = A.Get_color();
 AnsiString c1,c2;
if((A.Get_update())&&(operationcourante !=newbranchenote)&&(operationcourante !=newheritage))
 { if(A.Get_C()[0].maximale < 1000)c1 = IntToStr(A.Get_C()[0].minimale)+".."+IntToStr(A.Get_C()[0].maximale);
  else c1 = IntToStr(A.Get_C()[0].minimale)+"..*";
  if(A.Get_C()[1].maximale < 1000)c2 = IntToStr(A.Get_C()[1].minimale)+".."+IntToStr(A.Get_C()[1].maximale);
else c2 = IntToStr(A.Get_C()[1].minimale)+"..*"; }
                                        //si relation uniaire.
 if(A.Uniaire())
  { TPoint points[4];  points[0] = A[0].Get_Point();
points[1].x = A[0].Get_Rect().right+30;points[1].y =A[0].Get_top();  points[2] = points[1];
  points[2].y = A[0].Get_Rect().top-30;  points[3] = points[0];points[3].y = points[2].y;
  drawzone->Canvas->Polygon(points,3);
  tagPOINT r1,r2;  r1.x = points[1].x-(A.Get_Roles()[0].Length()*4);  r1.y = points[1].y-15;
  r2.x = A[0].Get_left()-(A.Get_Roles()[1].Length()*4);  r2.y = points[2].y-15;
  drawzone->Canvas->TextOutA(points[1].x-(A.Get_name().Length()*2),r2.y+30,A.Get_name());
  if(A.Get_update())
  {  drawzone->Canvas->TextOutA(r1.x,r1.y,A.Get_Roles()[0]);
   drawzone->Canvas->TextOutA(r2.x,r2.y,A.Get_Roles()[1]); drawzone->Canvas->TextOutA(r1.x,r1.y+20,c1);
   drawzone->Canvas->TextOutA(r2.x,r2.y+20,c2); } return ; }
 drawzone->Canvas->MoveTo(A[0].Get_left(),A[0].Get_top());
 drawzone->Canvas->LineTo(A[1].Get_left(),A[1].Get_top());
 TSegmentDroite Seg(A[0].Get_Point(),A.Get_I()[1].Get_Point());
 POINT m = Seg.Get_Middle(); drawzone->Canvas->TextOutA(m.x-(A.Get_name().Length()*4),m.y+5,A.Get_name());
 if(!Drawing) {
 POINT p1,p2;  Seg.intersection(A.Get_I()[0].Get_Rect(),p1);  Seg.intersection(A.Get_I()[1].Get_Rect(),p2);
 TSegmentDroite* seg = new TSegmentDroite(p1,p2);
 POINT m = seg->Get_Middle();  float l = seg->LongeurSegment()/2;  drawzone->Canvas->MoveTo(500,0);
 POINT r1 = seg->pointdistant(l-10,m,false);  POINT r2 = seg->pointdistant(l-10,m,true);
 if(r1.x > m.x)r1.x -=(A.Get_Roles()[0].Length()*((A.Get_Roles()[0].Length()>4)?7:5));
 if(r2.x > m.x)r2.x -=(A.Get_Roles()[1].Length()*((A.Get_Roles()[1].Length()>4)?7:5));
 if(A.Get_update())  { drawzone->Canvas->TextOutA(r1.x,r1.y-15,A.Get_Roles()[0]);
  drawzone->Canvas->TextOutA(r2.x,r2.y-15,A.Get_Roles()[1]);
  drawzone->Canvas->TextOutA(r1.x,r1.y+5,c1); drawzone->Canvas->TextOutA(r2.x,r2.y+5,c2); } delete seg;
 }}
```

2) Méthode UpdateIndividu (TIndividu I) : Cette permet de faire une mise a jour a chaque fois qu'un individu soit modifier (déplacer, éditer, supprimer,…).

Il y a aussi deux autres classes interfaces qui rassemble à cette classe pour les deux autres diagrammes de cas d'utilisation et séquence.

- **Classe projet :**

Cette classe définie un ensemble de méthodes qui permettent de gérer un projet comme ajouter/supprimer vue, ajouter/ supprimer diagramme, nouveau projet, charger, enregistrer projet….

La fenêtre principale du logiciel :

Quand vous lancez notre logiciel l'interface suivante va apparaître :

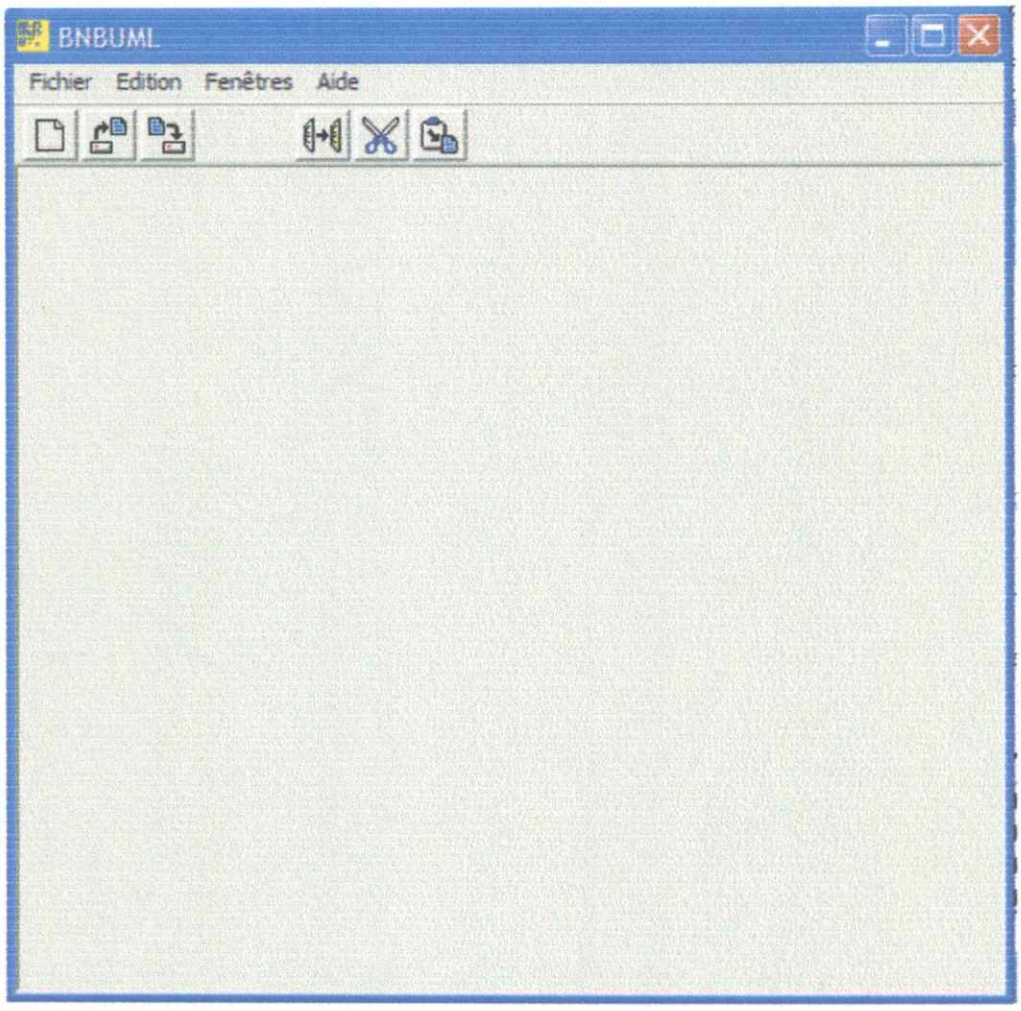


Figure 2 : fenêtre principale du logiciel.

Dés le premier regard, il est clair qu'elle se compose de deux parties :

- **Bar de Menu :**

Cette une barre de menus est conçue pour offrir à l'utilisateur du logiciel quelques fonctionnalités essentielles, elle contient les menus suivants :

- **Fichier :**

Permet de réaliser les traitements habituels sur les fichiers comme : créer (un projet, diagramme de classe, diagramme de cas d'utilisation ou diagramme de séquence), ouvrir, enregistrer, enregistrer sous et quitter.

- **Edition :**

Ce menu permet de manipuler facilement les composants dans les diagrammes en permettant de couper un composant, copier un composant et enfin le coller.

- **Fenêtres :**

Comme on peut créer plusieurs fenêtres, il est parfois utile de les arranger et c'est le rôle de ce menu. Donc ce menu permet d'arranger les fenêtres soit en cascade, soit en fenêtres, soit de les réduire tous.

- **Aide :**

Elle contient une présentation générale de notre logiciel.

- **Barre d'outils :**

Cette barre d'outils porte plusieurs boutons, ces boutons sont répartis selon leurs fonctions en deux groupes :

- ❖ Les trois premiers boutons permettent de réaliser des opérations sur les fichiers (nouveau, ouvrir, sauvegarder).

- ❖ Les trois autres boutons permettent de réaliser des opérations sur les composants (couper, copier, coller).

Cette fenêtre principale reste toujours visible et ne se ferme que si on quitte le logiciel.

La fenêtre projet :

Cette fenêtre apparaît après la création d'un projet ou l'ouverture d'un projet.

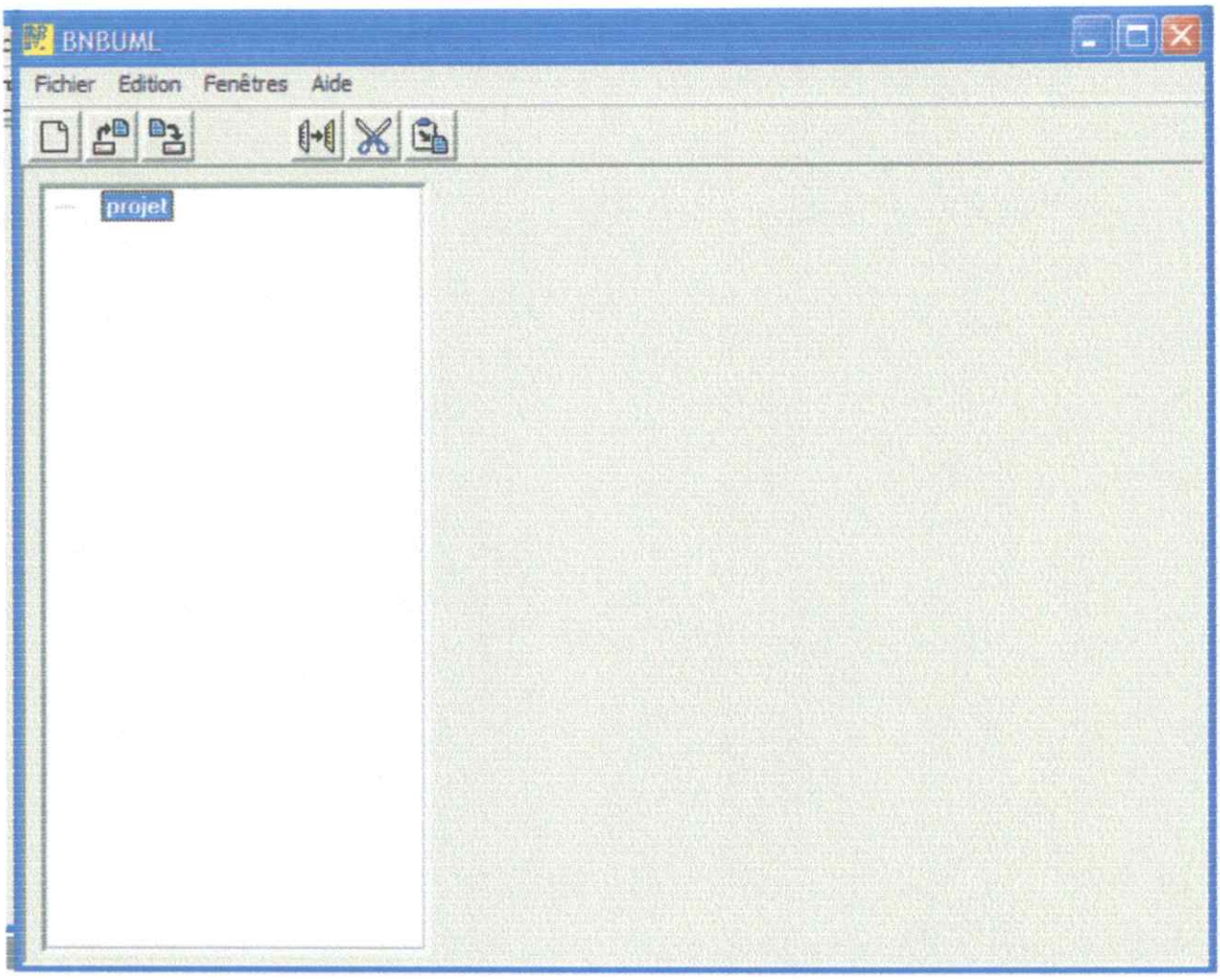


Figure 3 : fenêtre projet.

Comme on le voit, cette fenêtre contient en plus des composants de la fenêtre principale un arbre dynamique qui permet de visualiser les vues et les diagrammes du projet. Et voici la fenêtre projet après création de vues et de diagrammes.

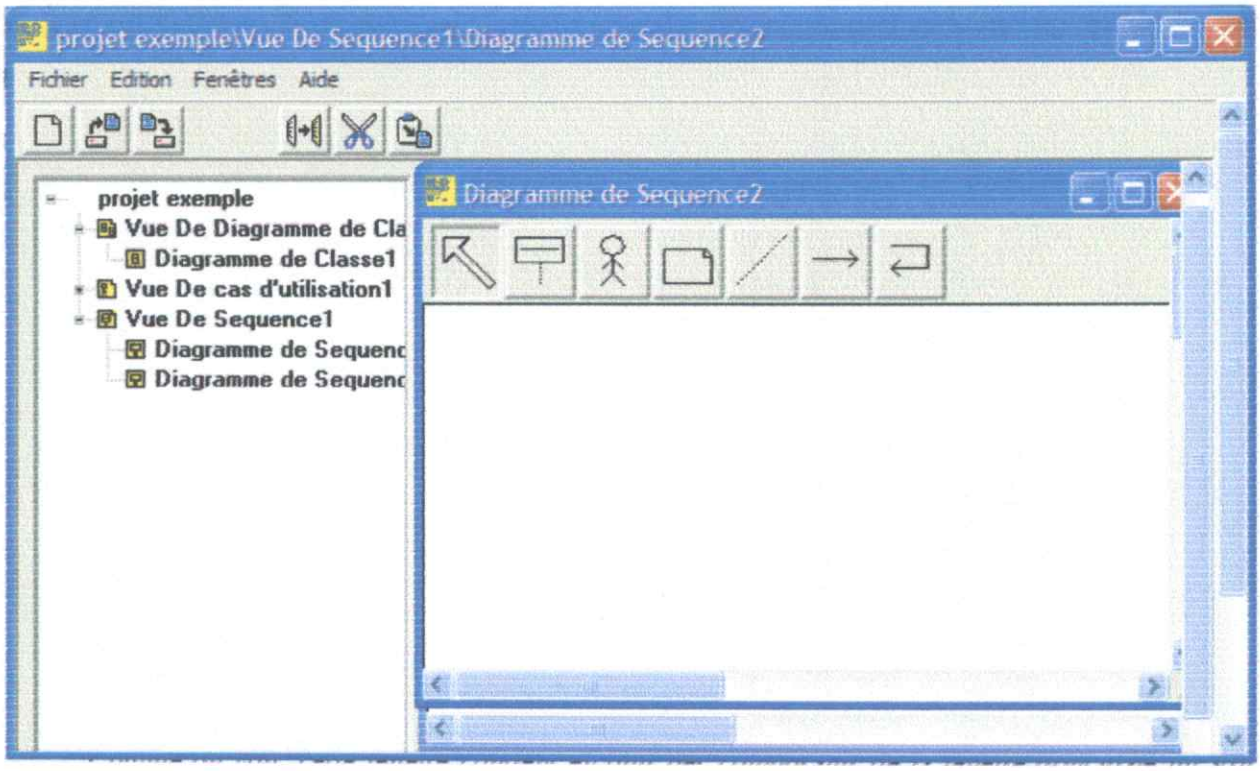


Figure 4 : fenêtre projet avec vues et diagrammes.

La fenêtre diagrammes :

Cette fenêtre est la même pour les trois type de diagrammes, le seul changement est dans les boutons des composants et relations à afficher qui ce changent selon le type de diagramme.

Cette fenêtre contient : une zone de dessin pour qu'on place les composants et les relations et le boutons permutant de dessiner les composants et relations entre composant. Voici les fenêtres qui s'affichent pour chaque diagramme.

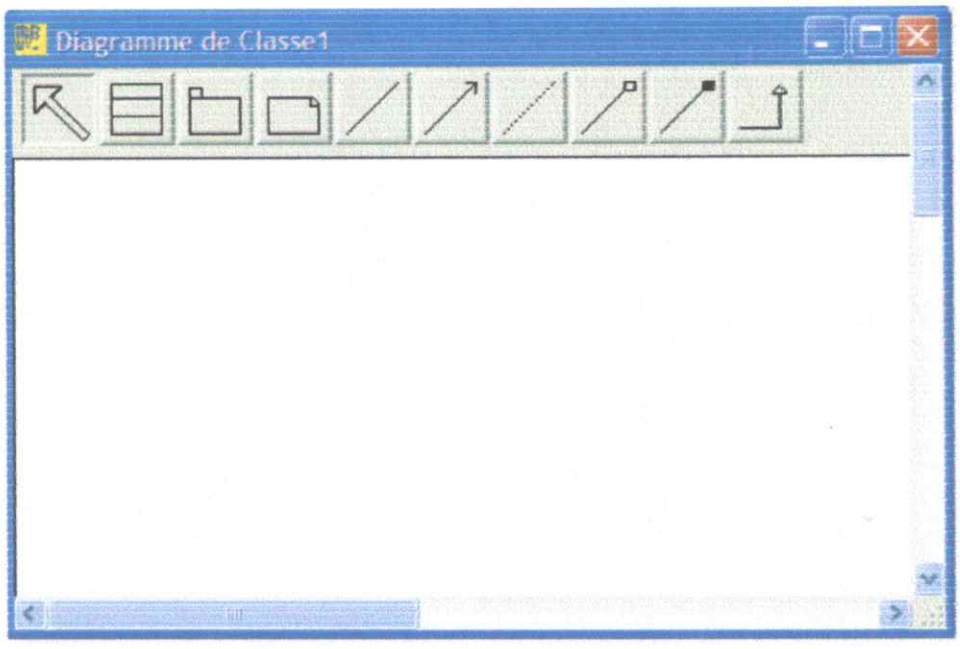


Figure 5 : fenêtre diagramme de classes.

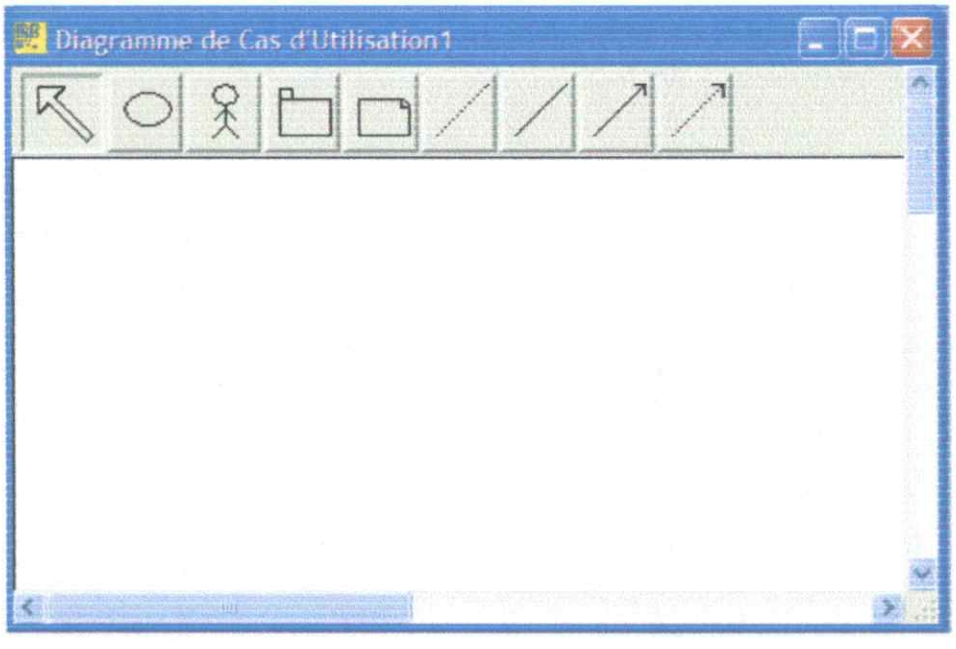


Figure 6 : fenêtre diagramme de cas d'utilisation.

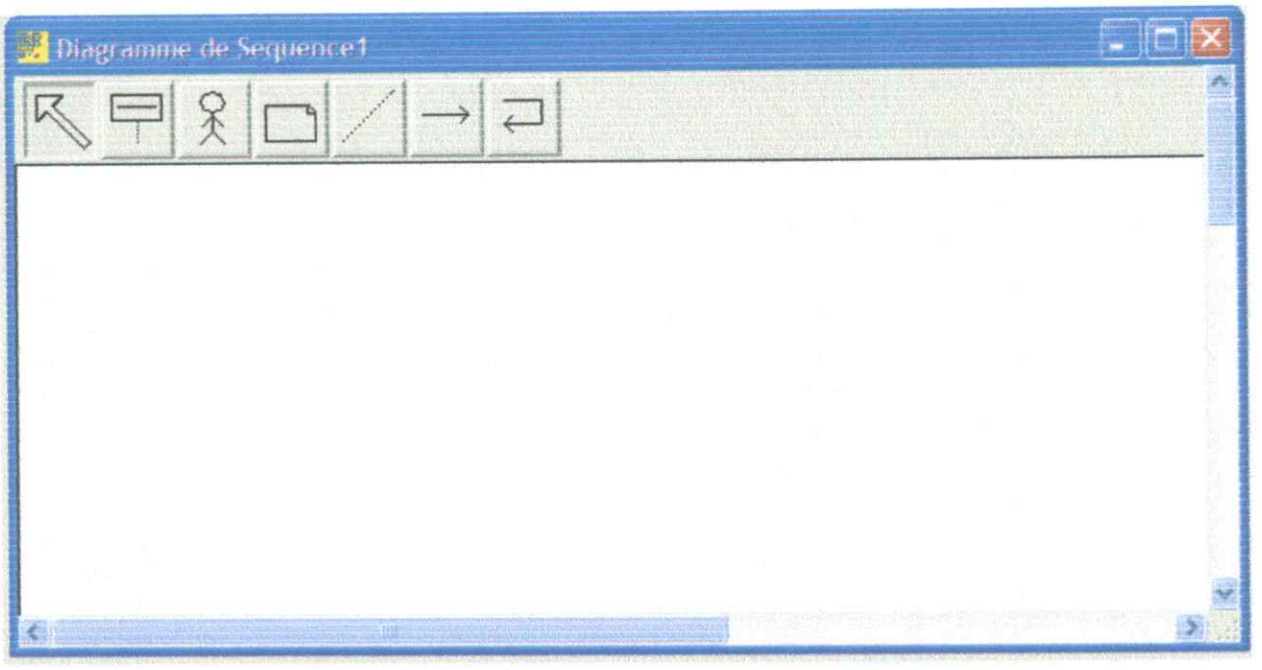


Figure 7 : fenêtre diagramme de séquence.

C'est dans chacune de ces fenêtres et en utilisant les boutons existants qu'on peut dessiner nos diagrammes.

Les boites de dialogues :

Ces boites sont utilisées pour permettre à l'utilisateur de spécifier les propriétés d'un composant. Par exemple pour la classe on a une boite pour les attributs et une autre pour les méthodes, même chose pour les relations : on a une boite de dialogue qui permet de spécifier les rôles et les cardinalité et parfois le type de la relation et pour une note (les notes sont utilisées dans les trois diagrammes avec même implémentation et même signification) on a aussi une boite de dialogue standard.

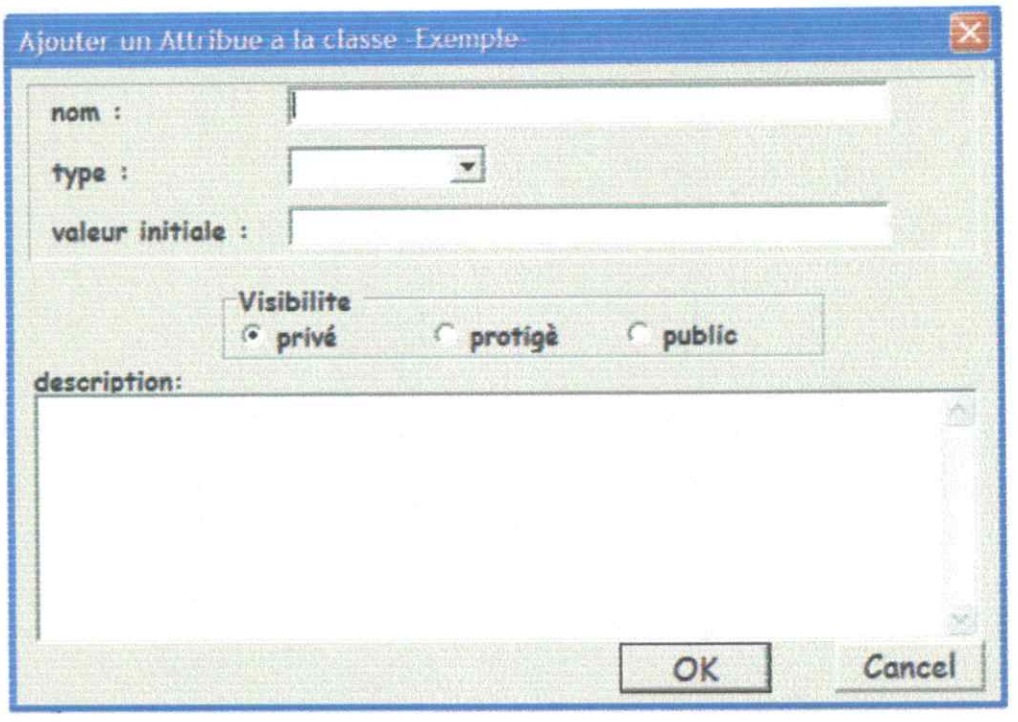


Figure 8 : Boite de dialogue ajouter attributs pour la classe.

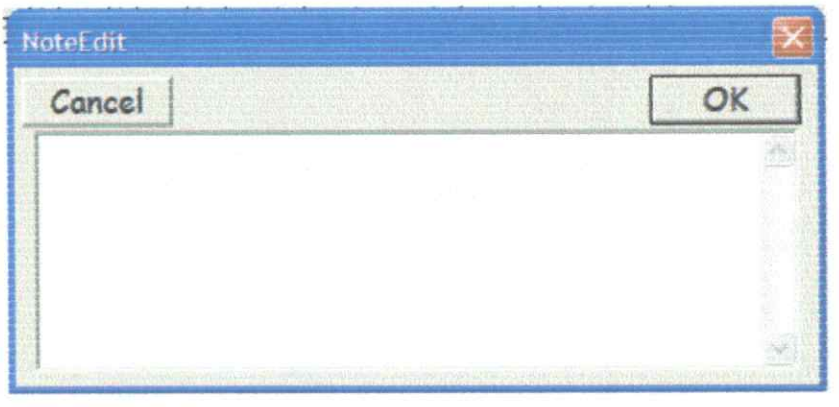


Figure 9 : Boite de dialogue éditer une note.

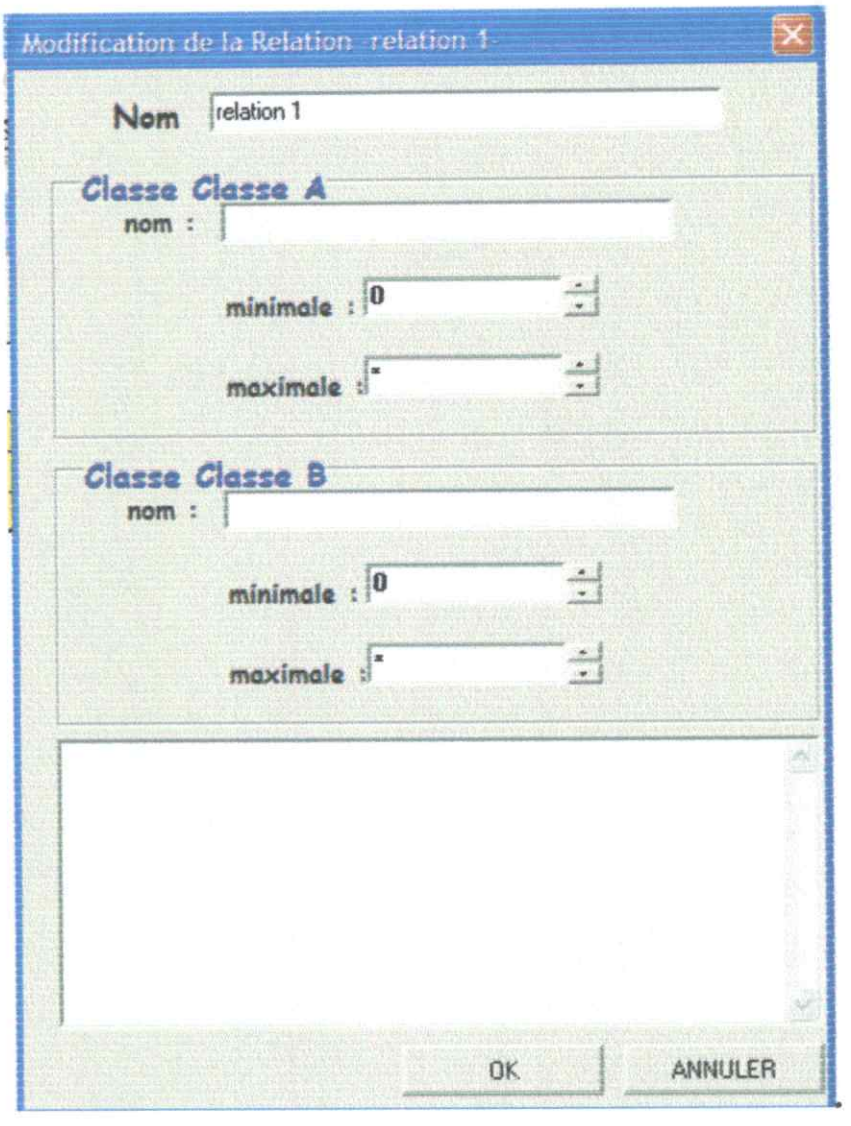


Figure 10 : Boite de dialogue éditer une relation.

4. Problèmes rencontrés lors de la réalisation :

Problème de rafraîchissement :

L'un des problèmes majeurs que nous avons rencontrés durant le développement est le problème de rafraîchissement de dessin après que la fenêtre soit redimensionner, masquer ou iconifier ou qu'une fenêtre de dialogue s'ouvre. Dans ces cas cités le dessin disparaît de la zone de dessin.

Solution adaptée :

Pour résoudre le problèmes de rafraîchissement du dessin on à procédé comme suit :

Définir une méthode qui définie comment chaque composant doit se dessiner. Cette méthode est : Void Paint (AnsiString nom) mais avant d'appelé Paint il faut sauvegarder les informations de tous les composants que contient le dessin, pour cela on a définit pour chaque type de composant un ensemble qui contient les composants de ce type dans le dessin.

Problème de non stabilité de dessin lors du déplacement d'un composant :

Ce problème apparu lorsqu'on déplace un composant sur la zone de dessin ce qui à impliquer le rafraîchissement fréquent du dessin en redessinant le composant déplacer dans ses nouvelles positions ce qui a causé une instabilité du dessin.

Solution adaptée :

Ne pas utilisé le rafraîchissement inutiles c'est-à-dire ne rafraîchir que dans les cas ou nous avons besoin.

Problème de superposition de relations dans le cas de plusieurs relations entre les mêmes deux composants :

Ce problème apparu lorsqu'on crée deux relations ou plus entre deux composants car la méthode utilisée pour le dessin des relations utilise le centre de chaque composant comme point de départ.

Ce problème a deux solutions : soit permettre a l'utilisateur de mettre des points d'encrage (il peut casser la ligne) ou lui permettre de visualiser l'une des relation en cliquant sur elle par le bouton droit et demander de la visualiser. C'est cette deuxième solution que nous avons adaptés.

CHAPITRE VII

Test et Validation

1. Introduction :

Dans ce chapitre nous s'assurons, par l'exécution du logiciel, que le système conçu satisfait a ces objectifs fonctionnels spécifiés en phase d'expression des besoins. Normalement nous devons testés quelques cas d'utilisation déjà vue dans la phase d'expression des besoins, mais comme nous l'avons cités dans le chapitre « processus de développement », le langage utilisé dans la modélisation de notre projet est UML et tous les diagrammes présentés jusqu'à la ont été réalisés par notre logiciel, ça sera donc une validation de notre logiciel et une preuve qu'il permet bien de modéliser les trois diagrammes qu'il support.

La partie qui reste de la validation et du test est de vérifier que notre logiciel détecte les erreurs et c'est ce que nous allons voir avec les figures qui viennent.

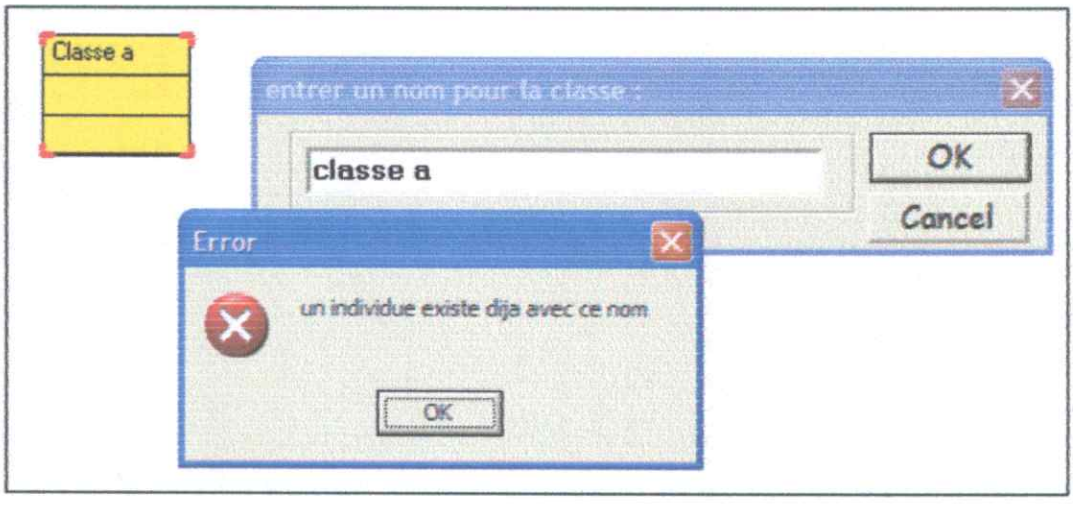


Figure 1 : Erreur de création de classe avec même nom.

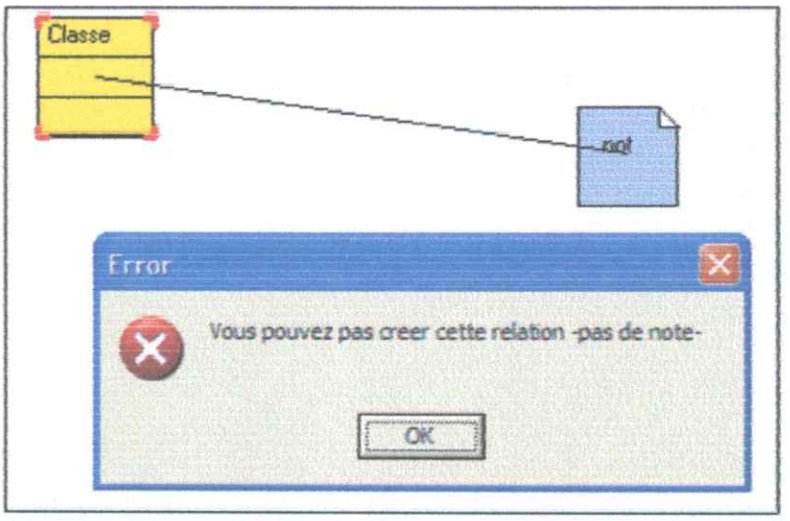


Figure 2 : Erreur de création de relation non définie pour le type de composant.

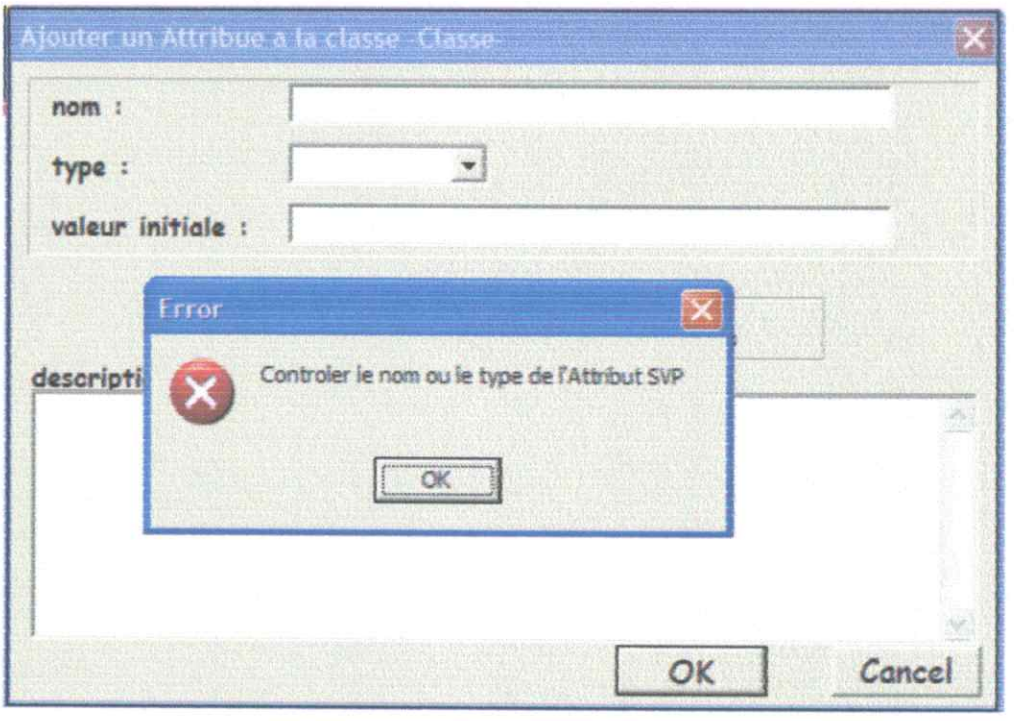


Figure 3 : Erreur de remplissage de propriétés.

2. Conclusion :

Après avoir testé notre logiciel tout au long du processus de développement et après avoir testé quelques exceptions on est arriver a valider notre logiciel après n'avoir détecté aucune anomalie.

CHAPITRE VIII

Conclusion et perspectives

« UML est le langage de modélisation orienté objet le plus connu et le plus utilisé au monde »[1].

Issu "du terrain" et fruit d'un travail d'experts reconnus, UML est le résultat d'un large consensus. De très nombreux acteurs industriels de renom ont adopté UML et participent à son développement.

En l'espace d'une poignée d'années seulement, UML est devenu un standard incontournable.

UML à permet de voir la conception d'un point de vue différent et montrer la voie à une approche nouvelle de développement et de la conception a base d'objet.

Le travail présenté dans ce mémoire consiste à la conception et réalisation d'un éditeur de diagrammes pour l'UML en utilisant le C++ Builder.

L'utilisation de C++ Builder pour l'implémentation nous à permet de découvrir les avantages qu'offre C++ et d'utiliser ces performances pour réaliser les meilleurs logiciel de modélisation.

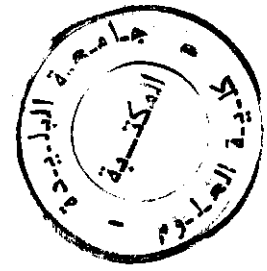
Le logiciel que nous avons développé offre plusieurs facilités à ces utilisateurs que nous avions mentionnés précédemment.

[1] Xavier Blanc, XB-UML

Comme nous l'avons cité à l'introduction générale notre travail s'annonce comme un point de départ d'un grand projet qui s'articule sur le langage **UML** pour arriver a réaliser un éditeur de diagrammes **UML** dédier aux systèmes distribués. De ce fait on peut cité comme perspectives :

- Ajouter les autres diagrammes aux trois diagrammes que supporte notre logiciel pour qu'il permet de réaliser tous les diagrammes du langage UML
- Ajouter la génération de codes source à partir des diagrammes modélisés pour les langages les plus connus comme Java et C++.
- Ajouter la reverce-engeniring : a partir du code source générer les diagrammes.
- Sauvegarder les fichiers sous le format standard XML pour permettre une compatibilité avec les autres logiciels d'édition de diagrammes UML.
- Ajouter l'intégration avec l'outil Office de microsoft.

Nous espérons, a la fin que ce modeste travail soit a la hauteur , et reflète bien les efforts déployés tout au long d'une année de travail afin d'aboutir a un travail honorable.



La bibliographie :

[1] [Bouzeghoub 97] : M.Bouzeghoub, G.Gardarin, P.Valduriez, Les objet, 2° tirage, Eyrolles, 1997.

[2] [Lai 97] : M. Lai, la notation unifiée de modélisation objet, application en JAVA, Masson 1997.

[3] [Muller 97] : P.A.Muller, Modélisation objet avec UML, Eyrolles, 1997.

[4] [Fannader 99] : R. Fannader , UML : principes de modélisation, Dunod 1999.

[5] : Ludovic Courtes, Zoé Drey, Sébastien Pierre, UML Modeling Environment Spécifications.

[6] : G.Booch, J.Rumbaugh, I.Jacobson, Le guide de l'utilisateur UML, Groupe Eyrolles, Paris 2003.

[7] [Reisdorph 98]: R.Reisdorph , Le programmeur, apprendre C++ Builder 3.0 en 21 jours, Simon & Schuster Macmillan, 1998.

[8] : J.Printz, le ginie logiciel, Univercité de France, Mai 2000.

Liens intéressants :

http://www.gentleware.com: Le site de l'éditeur de **POSEIDON.**

http://www.rational.com/ UML: Les pages UML de Rational.

http://www.uml.free.fr/ : Site francophone dédie à UML.