UNIVERSITE SAAD DAHLAB

Faculté des Sciences

Département d'Informatique

MÉMOIRE DE FIN D'ÉTUDES

Pour l'obtention du diplôme d'ingénieur d'état en Informatique

Option : Intelligence Artificielle

APPROCHE ORIENTÉE OBJETS ET PATTERNS : APPLICATION À LA MODÉLISATION D'UN BRAS ARTICULÉ

M^elle. HASSAÏNE Salima

Et

M^r. MAZARI-BOUFARES Farid

Devant le jury composé de :

Dr. BENBLIDIA. N	Présidente de jury
Dr. AIDE. A	Examinateur
Dr. BLIDIA.	Examinateur
Dr. KAZEDE.	Examinateur
Dr. OUKID-KHOUAS. S	Examinatrice

Blida, Algérie

Septembre 2004

UNIVERSITE SAAD DAHLAB

Faculté des Sciences

Département d'Informatique

MÉMOIRE DE FIN D'ÉTUDES

Pour l'obtention du diplôme d'ingénieur d'état en Informatique
Option : Intelligence Artificielle

APPROCHE ORIENTÉE OBJETS ET PATTERNS : APPLICATION À LA MODÉLISATION D'UN BRAS ARTICULÉ

M^elle. HASSAÏNE Salima
Et
M^r. MAZARI-BOUFARES Farid

Devant le jury composé de :

Dr. BENBLIDIA. N	Présidente de jury
Dr. AIDE. A	Examinateur
Dr. BLIDIA.	Examinateur
Dr. KAZEDE.	Examinateur
Dr. OUKID-KHOUAS. S	Examinatrice

Blida, Algérie

Septembre 2004

REMERCIEMENTS

Nous remercions vivement les membres du jury de nous avoir fait l'honneur d'être rapporteurs et examinateurs tout en apportant leurs remarques et leurs contributions à l'élaboration de ce mémoire.

Tous nos remerciements aux personnes qui, par leurs conseils et leurs encouragements ont contribué à l'aboutissement de ce travail :

À Mr Mahieddine Mohamed notre promoteur qui nous a fait bénéficier beaucoup de son expérience et qui a tout fait pour faire aboutir ce travail.

À Mme Oukid-Khouas la directrice du laboratoire LRDSI (Laboratoire de Recherche en Développement des Systèmes Informatiques) de l'Université de Blida, pour nous y avoir accueilli et avoir facilité le bon déroulement de nos travaux.

Nous adressons nos sincères remerciements au chef de département Mme Benstiti Souâd qui nous a introduit au domaine de l'informatique, qui nous a mis sur une bonne voie au début de notre carrière scientifique et qui continue toujours à nous aider.

Nous ne manquerons d'exprimer notre gratitude envers nos professeurs pour les conseils judicieux qu'ils ont su nous communiquer, ainsi que pour leurs expériences et leurs professionnalismes dont ils nous ont fait profiter :

Melle Benblidia, Mr Benhouhou, Mr Bennouar, Mme Benstiti, Mme Boughrera, Melle Boustia, Mr Chelali, Mr Derbala, Mme Guerti, Mr Hadj Yahia, Mr Hannane, , Mr kebbir,Mr Koudil, Mr Mennacer, Melle Mokhtari Mme Ouahrani, Mme Oukid-Khouas, Melle Reguieg, , ... Cette liste n'est bien sûr pas exhaustive, et que les personnes non mentionnées veuillent bien nous en excuser.

Nous exprimons notre profonde reconnaissance à tous ceux qui nous ont aidé, de prés ou de loin, matériellement ou moralement, partager nos efforts pour voir naître ce modeste ouvrage.

Et enfin, Nous tenons à remercier tous nos amis Brahim Mezaourou, Tahar Benstiti, Souâd Boukrif, Abderahmen Bounouni, Nadjia Merabet, Mohamed Kochrane, Mohamed Nadjib Hamdi, Yacine Lakhdhari pour les bons moments qu'on a passé ensemble et Abdel Ouahab pour son amitié et l'aide qu'il nous a apportée pour l'édition de ce mémoire.

Dédicace

À mon père	*Tes conseils, ton support et ta volonté m'ont toujours inspirés et sont pour moi la clé de ma réussite*
À ma mère	*Ton amour inconditionnel et ta grande charité forment l'essence de mon existence. Tu es donc avec moi dans tout ce que je fais ;*
À ma sœur	*Nassima ;*
À mes deux frères	*Amine et Aymène ;*
À toute ma famille ;	
À tous mes Amis :	*Rehabe Bouchareb, Brahim Mezaourou, Abdel Ouahab , Souâd Boukrif, Tahar Benstiti, Merabet Nadjia, Mohammed Kochrane, Adb Errahman Bounouni , Mohamed Nadjib Hamdi, Yacine Lakhdhari, Sofiane Attou,Rachid Houmani, Nadjet Zerf,... et tous ceux avec qui j'ai passé de bons moments.*

Je vous dédie ce travail

Salima Hassaïne.

Dédicace

Je dédie ce mémoire à mes parents qui m'ont soutenu tout au long de ma vie, qui m'ont guidé et m'ont orienté dans la bonne voie. Je les remercie pour m'avoir aidé à surmonter les obstacles et les difficultés.

A mes deux merveilleux frères Salim et Liçou,

A mes chers amis : Khireddine, Smisha, Mehdi, Alpha(l'escroc), Aouada Rostom, Tahar, Redouane, Brahime Mezaourou, Hamza, Adnane, Abdelkader, Boudar, Amine Boulahia, Hishem, Zaki Loulouche, Zaki Abidate, yassine, Abdelouahab djongar, Abdelouahb (cyber), Gilmoore, Rabie et Imade, Hadjouti, Imane, Nesma, Amel, Nadjia, Abdelrahmane, Rahmani, kocherane, Mohamed, Nadjib .....

A tout ceux que j'ai oublié de citer.

Que le rêve "Old Raven" continue.

MAZARI-BOUFARES Farid

« Ce n'est pas dans la science qu'est le bonheur,
mais dans l'acquisition de la science »

Edgar Allen Poe.
Tales of The Grothesque and Arabesque,
The Power of Words

SOMMAIRE

LISTE DES FIGURES

CHAPITRE I
RAPPEL DES NOTIONS GÉNÉRALES SUR LES BRAS MANIPULATEURS

CHAPITRE II
ÉTAT DE L'ART SUR LES SIMULATIONS DES BRAS MANIPULATEURS

CHAPITRE III
PROBLÉMATIQUE ET SPÉCIFICATION DES BESOINS

CHAPITRE IV
CONCEPTION ET RÉALISATION DU SYSTÈME

CHAPITRE V
LES TESTS

RÉSUMÉ

Les travaux récents s'orientent vers l'orienté objet et les patterns. Dans le domaine de la simulation des bras manipulateurs, on a remarqué l'absence de cette approche à travers l'étude de travaux réalisées dans diverses universités : l'université allemande Stuttgart (RoboSim) et l'université américaine Bridgeport (SARO).

L'objectif général de notre travail est d'appliquer les nouvelles approches de conception tel que l'orienté objet et les patterns, dans le domaine des bras manipulateurs, tout en suivant une méthode de développement agile.

ABSTRACT

Object Oriented design and Patterns represent modern techniques largely used nowadays. Through our investigations we noticed that these techniques have not been applied to the simulation of manipulator arms, "RoboSim" from Stuttgart, German and "SARO" from Bridgepor,USA are some of these examples.

The purpose of our work is to apply the above techniques to the manipulator arm using the XP (extreme programing) agile development methodology.

CHAPITRE I

RAPPEL DES NOTIONS GÉNÉRALES SUR LES BRAS MANIPULATEURS

INTRODUCTION

L'étude abordée dans ce projet se rapporte à l'utilisation des nouvelles techniques de développement (*Orienté Objet* et *Patrons de conception « Patterns »*) pour réaliser une simulation d'un bras manipulateur tout en suivant une méthode de développement agile (XP).

Cela vient du fait que les travaux qui ont été faits, dans le domaine de la simulation des bras manipulateurs, ont des problèmes de fragilité, la rigidité et l'immobilité (*voir Annexe A*) car ils utilisent des langages de programmation Orienté Objet, mais n'exploitent pas les principes fondamentaux de l'OOD (***Object Oriented Design-Conception Orientée Objets***) qui sont (*voir Annexe A*) : l'abstraction, l'OCP *(Open-Closed Principle)*, Polymorphisme dynamique, la substitution de Liskov *(LSP)*, l'inversion de dépendances *(DIP)*, L'ISP *(Interface Segregation Principle)*,...etc. Pour remédier à ces problèmes de dépendances et d'architecture **[1]**, **[2]**, **[3]**, nous avons opté pour une approche Orientée Objets et Patterns.

La conception Orientée Objet est un ensemble de techniques se basant sur l'abstraction, on crée le modèle d'un domaine, on définit les relations et les interactions entre les éléments de ce domaine. Cet ensemble de techniques nous permet de réaliser des systèmes moins rigides et moins fragiles, en éliminant les dépendances imprévues entre les composants du système.

Les patrons de conception « patterns » fournissent un ensemble de techniques facilitant l'analyse, la conception et la réalisation de systèmes informatiques. Ils réduisent l'immobilité d'un système en permettant la création de modules réutilisables. La rigidité et la fragilité d'un système sont aussi atténuées du fait de leur utilisation.

1. PRÉREQUIS

Pour réaliser une simulation d'un bras manipulateur en utilisant une approche orienté objet et patterns on a besoin d'avoir des connaissances dans les domaines suivants :

- Modélisation d'objets dans un espace tridimensionnel.
- Cinématique des bras manipulateurs (en particulier PUMA 560).
- Les patrons de conception,

- L'Orienté Objet,
- Les Méthodes agiles de développement.

2. ETAT DE L'ART

Quelques simulations de bras manipulateurs ont été déjà développées. Leur problème principal est la mauvaise utilisation des langages orientés objet (absence des concepts de l'OOD) et la difficulté de réutiliser des composants issus de ces systèmes.

Les 3 travaux sur les quels on s'est basé pour réaliser notre simulation sont **GRAS** développé par ***Chris Lattner*** en 1998, il Simule un manipulateur PUMA 560, **RoboSim** qui est une simulation d'un bras "Lynxmotion" développée par ***Johannes Schützner*** en 1996 à l'université de ***Stuttgart*** en ***Allemagne*** et **SARO** qui a été développé à l'université de ***Bridgport*** aux ***USA*** par ***Tarek Sobh***.

3. CONTENU DU MÉMOIRE

Notre travail consiste donc, à appliquer les nouvelles techniques de développement (Conception orientée objets et patrons de conception) à la réalisation d'un simulateur de bras manipulateur tout en suivant une méthode de développement XP. Pour cela, nous avons divisé notre travail en cinq chapitres :

- Le premier chapitre est un rappel des notions générales sur les bras manipulateurs.
- Le deuxième chapitre décrit quelques travaux antérieurs de simulations.
- Le troisième chapitre présente les problèmes des travaux exposés dans le chapitre II et la spécification des besoins.
- Le quatrième chapitre est consacré à la conception et réalisation de notre système, dans ce chapitre on va détailler notre solution pour résoudre les problèmes cités dans le chapitre précédent. On va utiliser le langage de notation UML pour présenter la structure des modules du système, ainsi que de son comportement.
- Le cinquième chapitre est consacré aux tests, on va montrer les tests des différentes étapes de réalisation de notre système.

I-1. INTRODUCTION

Ce chapitre présente une introduction de la terminologie **[4], [5], [6]** et des principes qui sont utilisés pour étudier et concevoir des robots. On va aborder plus précisément l'étude de l'architecture et de la cinématique des robots.

I-2. LES BRAS MANIPULATEURS

Un robot manipulateur peut être considéré comme une série de liaisons (*links*) connectées par des articulations (*joints*).

I-2.1. Liaison :

Une liaison est un maillon de la chaîne articulée, son rôle est de maintenir le rapport entre deux articulations (*joints*), elle est caractérisée par deux paramètres, a_n, α_n :

- a_n est la distance mesurée sur la normale commune, qui sépare deux axes d'articulation ***n*** et ***n+1*** (la longueur de la liaison).
- α_n est l'angle de torsion qui représente la rotation qui doit être effectuée pour faire coïncider l'axe de rotation ***n*** avec l'axe ***n+1***. **(Figure I.1).**

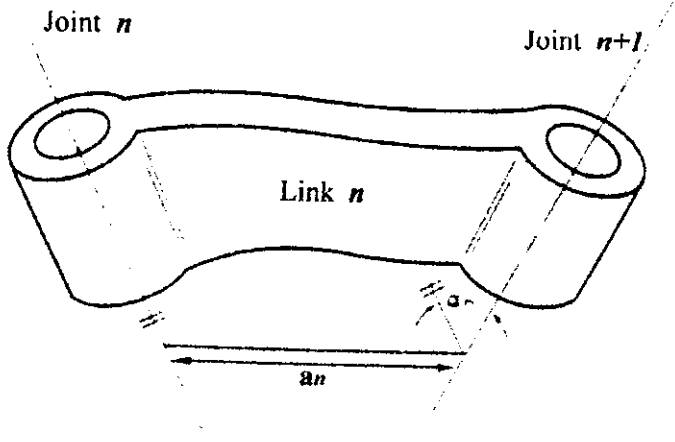


Figure I.1 [5]

Un axe doit avoir deux normales, une pour chaque liaison. La distance qui sépare deux normales adjacentes est désignée par $\mathbf{d_n}$ pour l'articulation ***n***, et θ_n représente la rotation

qu'il faut faire subir à la normale ***n-1*** pour qu'elle soit parallèle à ***n*** (angle entre deux liens ***n-1*** et ***n).***

I-2.2. Articulation (Joint) :

Généralement les articulations ont un degré de liberté, Elles peuvent être de deux types : articulation rotative (*Rotary Joint* or *Revolute Joint*) ou articulation prismatique (*Prismatic Joint* or *Linear Joint*). Dans le cas d'une articulation rotative c'est θ_n qui est variable et les autres paramètres restent constants **(Figure I.2)** par contre dans une articulation prismatique la seule variable est $\mathbf{d_n}$ **(Figure I.3)**.

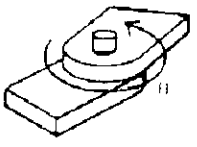


Figure I.2

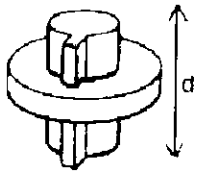


Figure I.3

I-2.3. Degrés de mobilité (DM):

Chaque paire liaison-articulation a un degré de mobilité (1 DM).

I-2.4. Degrés de liberté (DL) :

C'est le nombre de variables indépendantes qui permettent de définir la position de toutes las parties du mécanisme. Généralement à chaque articulation est associé un degré de liberté. Par conséquent le degré de liberté d'un bras manipulateur est égal au nombre d'articulations. **[5]**

I-2.5. Structure des robots (bras manipulateur) :

Une manière de décrire les robots consiste à préciser leur architecture (ou leur morphologie). Cette description donne des indications sur la façon dont les divers segments d un robot sont organisés et le déplacement des uns par rapport aux autres.

Une classification grossière de la morphologie des robots peut résumer ceux-ci à quatre types: Les robots de structure cartésienne **(Figure I.4-a)**, cylindrique **(Figure I.4-b)**, sphérique **(Figure I.4-c)** et les systèmes apparentés à un bras humain (bras articulé) **(Figure I.4-d)**. **[4], [6].**

a) **Robots à structure cartésienne (TTT) :**

Dans ce cas, le bras se déplace suivant les trois axes de translation x, y, z donc la structure possède 03 degrés de libertés de translation (TTT), voir **(Figure I.4-a)**. Le volume de travail de ce bras est un parallélépipède rectangle. **[4].**

b) **Robot à structure cylindrique (TTR) :**

Dans ce cas, le bras se déplace suivant les deux axes de translation et tourne autour de l'axe vertical donc la structure possède 02 degrés de libertés de translation et un degré de liberté de rotation(TTR), voir **(Figure I.4-b)**. Le volume de travail de ce bras est un élément de cylindre. **[4].**

c) **Robot à structure sphérique ou polaire (TRR) :**

Dans ce cas, le bras se déplace suivant un axes de translation et tourne autour de deux autres (TRR), voir **(Figure I.4-c)**.Le volume de travail de ce bras est une portion de sphéroïde (un élément de sphère). **[4].**

d) **Robot à structure articulée (RRR) :**

Dans ce cas, le bras est articulé à l'épaule, au coude et au poignet et ressemble donc plus au bras humain, les parties du bras peuvent tourner autour des 03 axes pour permettre un mouvement dans différents plans (RRR), voir **(Figure I.4-d)**.Le poignet possède un élément terminal qui ajoute trois degrés de liberté au bras. Le volume de travail de ce bras est un élément d'angle solide. **[4].**

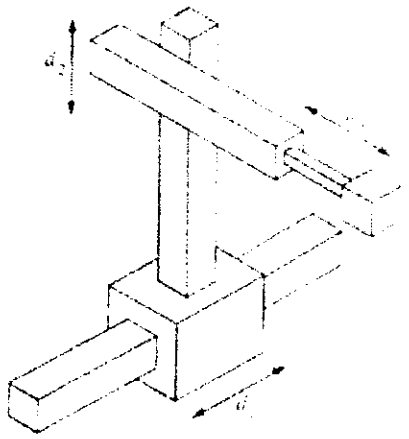


Figure I.4-a [6].

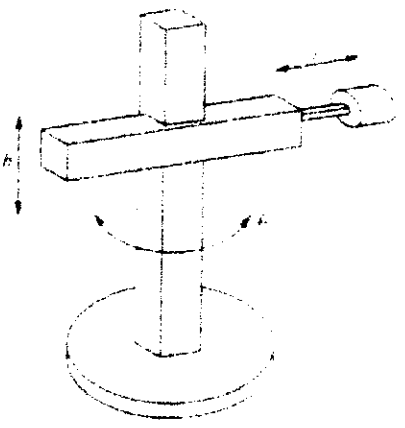


Figure I.4-b [6].

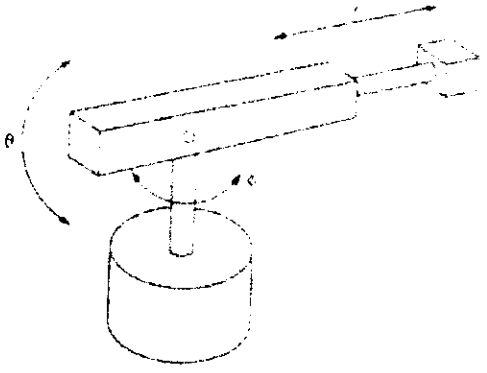


Figure I.4-c [6].

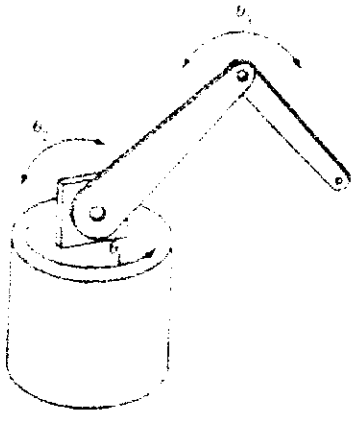


Figure I.4-d [6].

I-2.6. Notations de Denavit-Hartenberg :

Les notations de Denavit-Hartenberg se réfèrent à un système mécanique articulé doté de ***n*** degrés de liberté. L'élément de départ (la base) est considérée comme une liaison origine et porte le numéro 0.Chaque liaison de cette chaîne est caractérisée par son propre repère.

Les notations DH nous permettent de décrire la position et l'orientation de chaque repère par rapport à un autre en donnant la transformation nécessaire pour maintenir le rapport entre deux liaison successives, en d'autre terme pour passer d'une liaison à la prochaine.

On a vu dans les sections précédentes qu'une liaison ***n*** est caractérisée par $\mathbf{d_n}$ et θ_n. D'autre part, étant donné qu'une chaîne articulée, comme celle décrite dans **(Figure I.5)**, présente des axes de rotations dont l'orientation est quelconque, ces axes de rotation auront deux normales caractérisées par leurs longueurs a_n et a_{n-1}.On peut associer à chaque liaison de la chaîne un système d'axes de coordonnées. Par exemple, le référentiel associé avec le lien ***n*** aura pour origine o_n, lorsque les axes ***n*** et ***n+1*** présentent un point d'intersection, c'est ce point qui est choisi comme origine o_n.

Comme autre convention, l'axe $\mathbf{Z_n}$ du référentiel associé au lien ***n*** est aligné avec l'axe de rotation de l'articulation ***n+1***. Enfin, l'axe $\mathbf{X_n}$ du référentiel o_n est aligné sur la normale commune et dirigée du lien ***n*** vers le lien ***n+1***.L'axe $\mathbf{Y_n}$ est déduit de la position relative des axes $\mathbf{X_n}$ et $\mathbf{Z_n}$. Pour finir, l'angle θ_n sera nul lorsque les axes $\mathbf{X_n}$ et $\mathbf{X_{n-1}}$ seront parallèles et auront la même orientation.

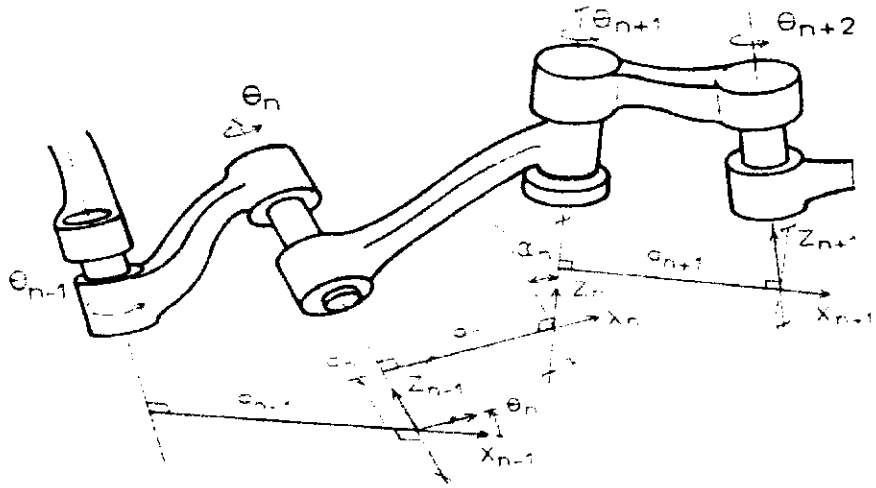


Figure I.5 [4], [5]

I-2.7. Matrices de transformations An et Tn :

Les matrices ***An*** décrivent la manière dont on passe d'un système de coordonnées au suivant le long d'une chaîne articulée. De ce fait, chaque matrice ***An*** caractérise la liaison à laquelle elle se rapporte et nous allons déterminer son expression.

Pour faire coïncider le référentiel ***n-1*** au référentiel ***n***, **(Figure I.5),** il faut translater le référentiel o_{n-1} le long de l'axe $\mathbf{Z}_{n-1}$ d'une longueur $\mathbf{d}_n$, puis de tourner le référentiel o_{n-1} autour de l'axe $\mathbf{Z}_{n-1}$ d'une quantité θ_n, ensuite, le référentiel doit être déplacé de a_n selon l'axe $\mathbf{X}_{n-1}$, l'axe est maintenant orienté dans la direction de la normale commune. Ce qui fait que les référentiels o_{n-1} et o_n ont leurs origines qui coïncident. Pour superposer tous les axes des deux référentiels, il est nécessaire d'effectuer une rotation du référentiel o_{n-1} au tour de l'axe $\mathbf{X}_{n-1} = \mathbf{X}_n$ avec une amplitude α_n. Ces transformations s'expriment par des translations et des rotations:

$$A \quad \text{Trans}(0,0,\mathbf{d})\ \text{Rot}(z,\theta)\ \text{Trans}(a\ 0,0)\ \text{Rot}(x,\alpha).$$

$$A = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & d \\ 0 & 0 & 0 & 1 \end{bmatrix} * \begin{bmatrix} \cos\theta & -\sin\theta & 0 & 0 \\ \sin\theta & \cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} * \begin{bmatrix} 1 & 0 & 0 & a \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} * \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\alpha & -\sin\alpha & 0 \\ 0 & \sin\alpha & \cos\alpha & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Ce qui donne :

$$A = \begin{pmatrix} \cos\theta & -\sin\theta.\cos\alpha & \sin\theta.\sin\alpha & a.\cos\theta \\ \sin\theta & \cos\theta.\cos\alpha & -\cos\theta.\sin\alpha & a.\sin\theta \\ 0 & \sin\alpha & \cos\alpha & d \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Les matrices ***Tn*** décrivent l'organisation de l'architecture des mécanismes articulés, et l'indice ***n*** décrit le nombre de degrés de liberté du système envisagé. C'est en ce sens que les matrices décrivent la transposition des coordonnées. Pour un système à six degrés de liberté, la transformation qui fait passer du référentiel de base lié à l'élément terminal d'un système est décrite par la matrice T_6 :

$$T_6 = A_1 A_2 A_3 A_4 A_5 A_6 \qquad (2\text{-}1)$$

I-2.8. Position de l'élément terminal d'un robot:

La position et l'orientation de l'élément terminal d'un robot se fait habituellement en précisant les coordonnées ***p*** du point de travail **(Figure I.6)** relativement au référentiel absolu. Dans cette description on associe un repère orthonormé à la pince **(Figure I.6),**

repère dans lequel le vecteur d'approche ***a*** décrit la direction de l'approche que réalise la pince par rapport à un objet, le vecteur d'orientation ***o*** est un vecteur spécifiant l'orientation des mâchoires de la pince et le vecteur normal ***n*** désigne une normale telle que :

$$\boldsymbol{n} = \boldsymbol{o} \times \boldsymbol{a}. \quad (2\text{-}2)$$

Enfin, les coordonnées du point de travail sont décrites par un vecteur ***p*** dont les composantes sont relatives à un référentiel absolu. Avec ces conventions la matrice ***T*** est reliée aux coordonnées du robot par:

$$T=\begin{pmatrix} nx\ ox\ ax.px \\ ny\ oy\ ay\ py \\ nz\ oz\ az\ pz \\ 0\ 0\ 0\ 1 \end{pmatrix}$$

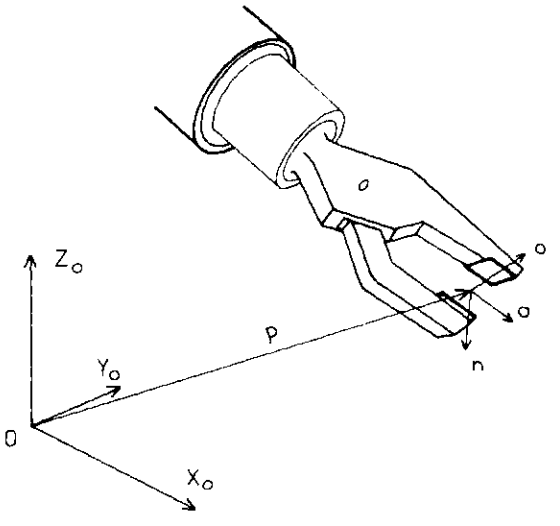


Figure I.6 [4].

I-2.9. Cinématique directe :

La façon la plus simple d'obtenir des mouvements consistants du bras manipulateur est d'utiliser la cinématique directe.

Le problème cinématique est toujours soluble. On peut en effet déduire une position géométrique à partir des valeurs des différents angles d'articulations θ_n, en utilisant les matrices de transformation T_n pour chaque liaison ***n***.

I-2.10. Cinématique inverse :

Cette deuxième façon de procéder permet de manipuler le bras d'une façon beaucoup plus effective. Elle consiste à exprimer les angles d'articulations à partir de la donnée des coordonnées et de l'orientation de l'élément terminal d'un robot.

La transposition inverse des coordonnées est un problème difficile **[4]** car il n'y a pas de méthode générale pour déduire les angles d'articulations θ. Pour un robot à 6 degrés de liberté, par exemple, la formulation de ce problème consiste à trouver la matrice avec les vecteurs n,o,a et soit:

$$T_6 = A_1.A_2.A_3.A_4.A_5.A_6 = \begin{bmatrix} n & o & a & p \\ 0 & 0 & 0 & 1 \end{bmatrix} \qquad (2\text{-}3)$$

La position $\boldsymbol{p}$ de l'élément terminal et son orientation $\boldsymbol{o}$ formant une donnée. Il faut rechercher les angles θ qui satisfont la formule ci-dessus. De façon à procéder p et pes, l'angle θ_1 est d'abord isolé en formant la relation suivante :

$$A_1^{-1}T_6 = A_1^{-1}.\begin{bmatrix} n & o & a & p \\ 0 & 0 & 0 & 1 \end{bmatrix} = A_2.A_3.A_4.A_5.A_6 \qquad (2\text{-}4)$$

Par identification terme à terme des deux membres de droite de (2-4), une relation est trouvée en exprimant θ_1 à partir des données n, o, a, p.

La procédure e t répétée pour θ_2 pour θ_3 ...etc. Donnant:

$$A_2^{-1}A_1^{-1}T_6 = A_2^{-1}A_1^{-1}.\begin{bmatrix} n & o & a & p \\ 0 & 0 & 0 & 1 \end{bmatrix} = A_3.A_4.A_5.A_6 \qquad (2\text{-}5)$$

$$A_3^{-1}A_2^{-1}A_1^{-1}T_6 = A_3^{-1}A_2^{-1}A_1^{-1}.\begin{bmatrix} n & o & a & p \\ 0 & 0 & 0 & 1 \end{bmatrix} = A_4 A_5 A_6 \qquad (2\text{-}6)$$

$$A_4^{-1}A_3^{-1}A_2^{-1}A_1^{-1}T_6 = A_4^{-1}A_3^{-1}A_2^{-1}A_1^{-1}.\begin{bmatrix} n & o & a & p \\ 0 & 0 & 0 & 1 \end{bmatrix} = A_5 A_6 \qquad (2\text{-}7)$$

$$A_5^{-1}A_4^{-1}A_3^{-1}A_2^{-1}A_1^{-1}T_6 = A_5^{-1}A_4^{-1}A_3^{-1}A_2^{-1}A_1^{-1}.\begin{bmatrix} n & o & a & p \\ 0 & 0 & 0 & 1 \end{bmatrix} = A_6 \qquad (2\text{-}8)$$

La résolution des systèmes d'équations obtenus en évaluant les relations ci-dessus nous permet de trouver les angles $\theta_1, \theta_2, \theta_3, \theta_4, \theta_5, \theta_6$. La position géométrique de l'élément terminal du robot est alors trouvée en appliquant la cinématique directe.

Le problème cinématique inverse n'est pas toujours soluble ce qui signifie que l'on ne peut pas atteindre n'importe quelle position (**Figure I.7**). De plus on se trouve souvent face à des équations ou un système d'équations dont la résolution est généralement complexe et donne des solutions multiples (**Figure I.8**). Dans ce cas on fait appel à des méthodes numériques utilisées pour la résolution des systèmes d'équations linéaires.

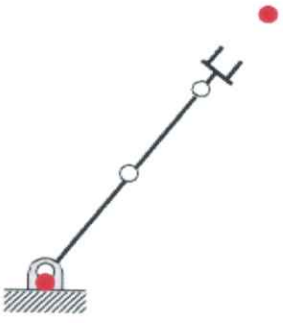


Figure I.7

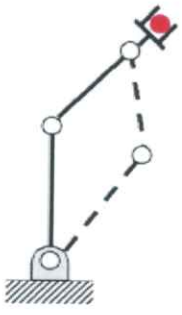


Figure I.8

I-3. PRÉSENTATION GÉNÉRALE DU ROBOT PUMA 560

Le bras manipulateur que nous avons utilisé est celui du robot PUMA 560, c'est un robot à six axes de rotation (six degrés de liberté). Chaque partie du bras manipulateur (*link*) est connectée aux autres par une articulation (*joint*), voir **Figure I.9.**

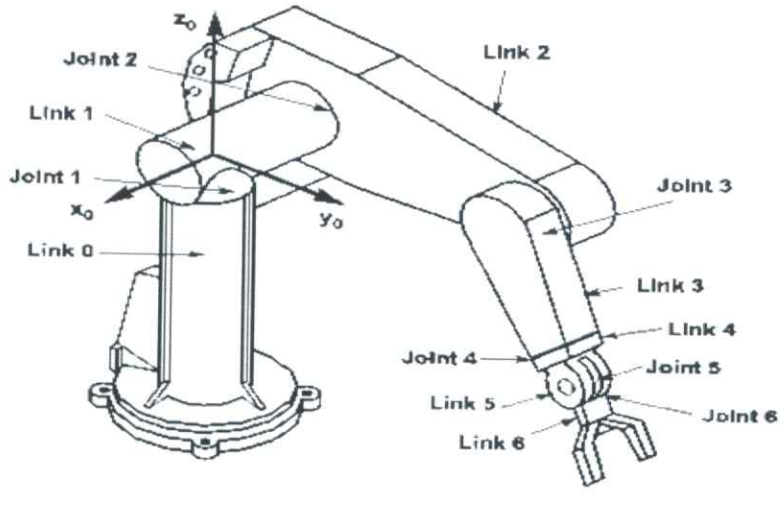


Figure I.9

La **Figure I.9** présente les repères des différentes articulations du robot en utilisant les notations de Denavit-Hartenberg.

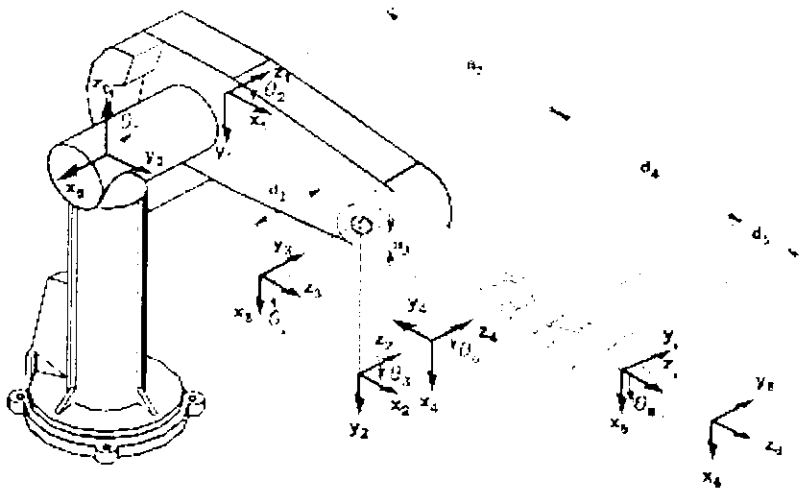


Figure I.10

La connaissance de la géométrie du modèle articulé (robot PUMA 560), c'est à dire la longueur de chaque segment ainsi que l'orientation de chaque liaison, permet de déduire mathématiquement la position de tout point attaché au système en fonction de la valeur des paramètres de liaison. La description des paramètres DH du robot PUMA 560 est présentée dans le tableau suivant :

Articulation	α_n	θ_n	$\mathbf{d_n}$	a_n
1	- 90°	θ_1	0	0
2	0	θ_2	d_2	a_2
3	90°	θ_3	0	a_3
4	- 90°	θ_4	d_4	0
5	90°	θ_5	0	0
6	0	θ_6	d_6	0

Tableau I.1

I-4. CONCLUSION

Ce chapitre nous a permis de présenter les bras manipulateurs , leurs structures et les démarches qui sont utilisées pour décrire les relations cinématiques d'un système articulé se déplaçant dans un espace à trois dimensions (3D).

CHAPITRE II

ÉTAT DE L'ART SUR LES SIMULATIONS DE BRAS MANIPULATEURS

II-1. INTRODUCTION

Dans ce chapitre nous allons présenter trois simulations développées en java. La première simulation ***(GRAS)*** et la seconde ***(SARO)*** simulent la cinématique directe d'un manipulateur ***PUMA 560***. La troisième ***(RoboSim)*** modélise la cinématique directe et inverse du bras « ***Lynxmotion***».

II-2. GRAS (GRAPHICAL ROBOT ANIMATOR AND SIMULATOR)

GRAS est un système développé par ***Chris Lattner*** en ***1998***. C'est un simulateur pour robots mécaniques, il est écrit en java.

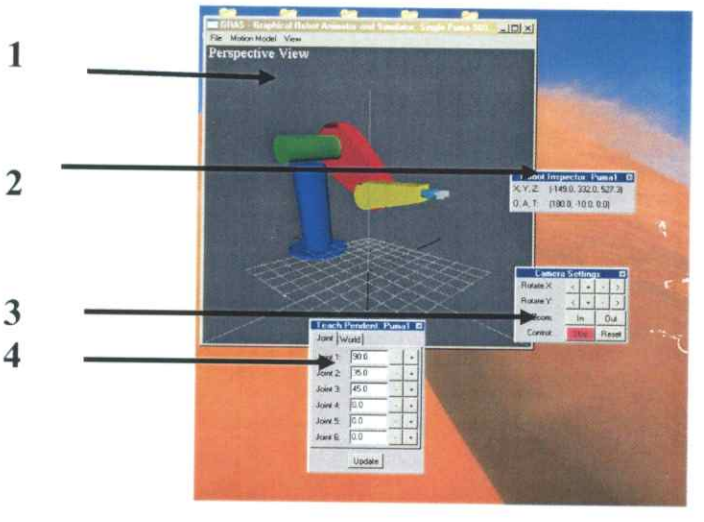


Figure I1.1

1 : La scène, elle contient le robot.

2 : visualiser la position de l'élément terminal du robot.

3 : Changement des paramètres de caméra.

4 : Pour changer la valeur des angles d'articulations du robot.

La figure ci-dessus montre l'exécution du programme GRAS dans sa forme la plus simple. La simulation présente une interface graphique qui permet de manipuler chaque lien du robot (cinématique directe).

Les figures **(Figure II.2), (Figure II.3), (Figure II.4) et (Figure II.5)** représentent les différentes options de *GRAS* :

- Les différentes vues de la caméra.
- L'incorporation des différents objets dans une même scène (cubes et robots).

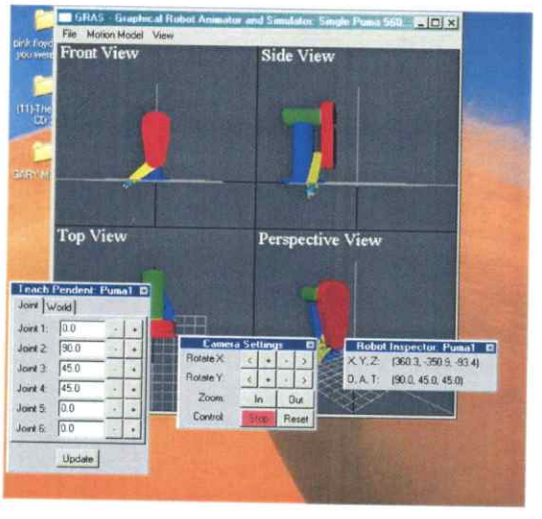


Figure I1.2

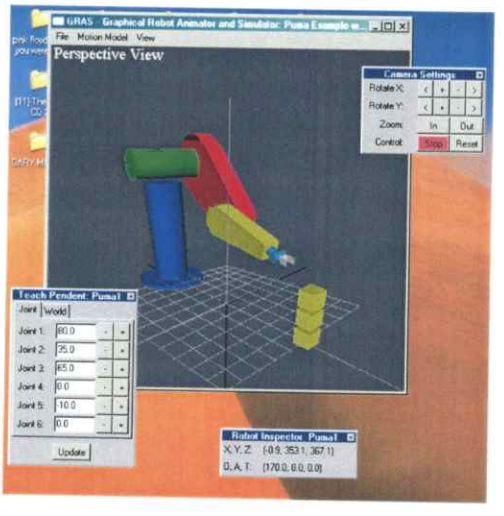


Figure 1I.3

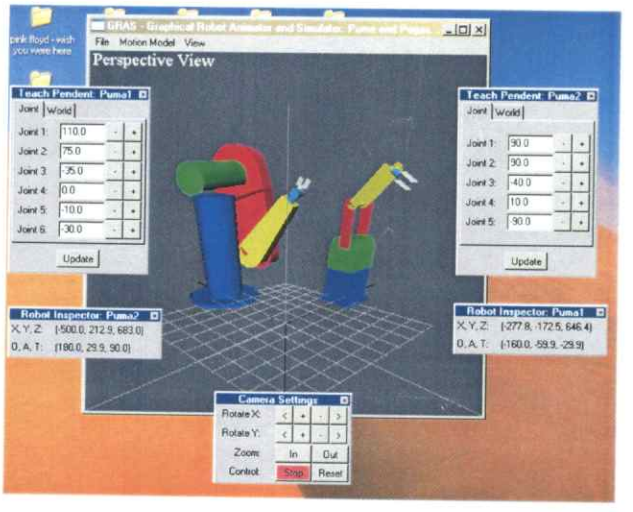


Figure 1I.4

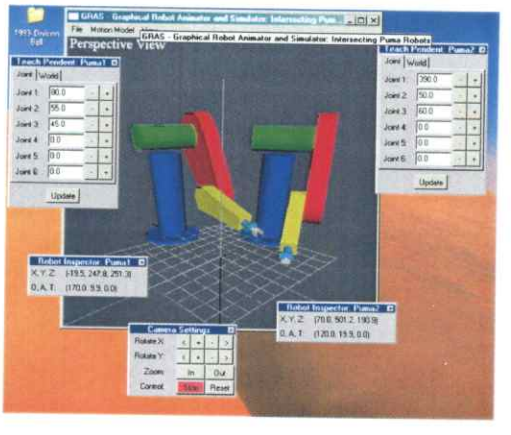


Figure 1I.5

II-3. SARO

Le but principal de ce logiciel est la simulation des robots industriels employés sur le marché . Il a été développé par *docteur* ***Tarek Sobh***, à l'université de ***Bridgeport*** institut de *Computer Science and Engineering Department. Au USA.*

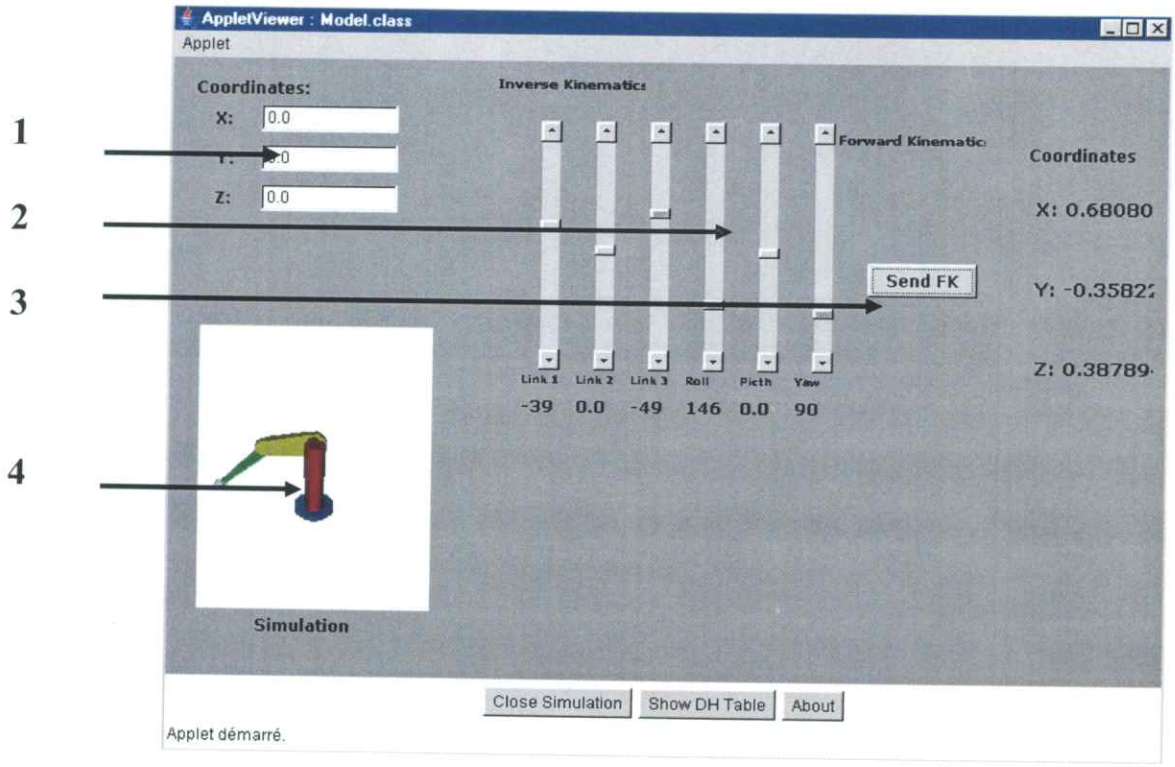


Figure I1.6

1 : Pour visualiser la position de l'élément terminal du robot.

2 : Pour changer la valeur des angles d'articulations du robot.

3 : Un bouton pour valider le choix.

4 : La scène qui contient le robot.

II-4. ROBOSIM

RoboSim est un programme qui simule un robot "***Lynxmotion***" disponible dans Joker Robotics.

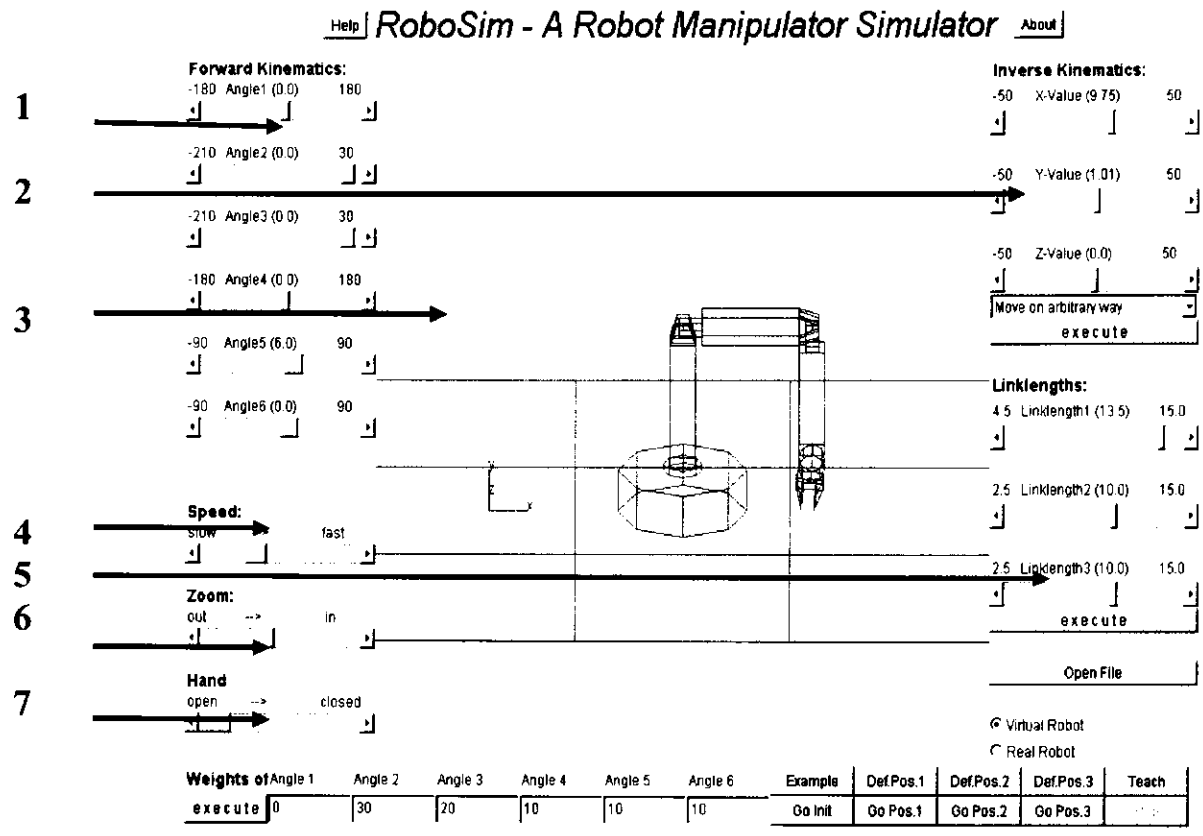


Figure 1.7

1 : Pour changer la valeur des angles articulaires du robot visualiser la position de l'élément terminal du robot.

2 : Pour introduire la position de l'élément terminal du robot, qui est utilisée dans le calcul de la cinématique inverse.

3 : La scène, elle contient le robot.

4 : Changement de la vitesse du robot.

5 : Changement des longueurs des liens (Links).

6 : Le changement du Zoom (manipulation de la caméra).

7 : La manipulation de la pince du robot.

Ce programme a été produit pour un cours pratique par Johannes Schützner, sous la direction du docteur ***Thomas Bräunl***, Université de Stuttgart, En ***1996***. il est basé sur le projet 'Studienarbeit' par Rainer Pollak, programmé en C en 1992.

CHAPITRE III

PROBLEMATIQUE ET SPECIFICATION DES BESOINS

III-1. INTRODUCTION

Dans le chapitre précédent, on a présenté quelques simulations de bras articulés utilisant un langage orienté objet (Java), mais on a remarqué, en examinant le code source, l'absence d'une conception orientée objets.

Ce chapitre montre quelques exemples de codes sources qui présentent des problèmes, ensuite nous allons introduire notre solution en donnant une description sommaire de ce que doit faire notre système.

III-2. PROBLÉMATIQUE ET SOLUTION PROPOSÉE

Exemple de *GRAS* :

```
public class Matrix4x4 extends Object {
        //...
        public double Values[][] = new double[4][4];
        // ...
}
```

Cette portion de code tirée du programme *GRAS*, elle représente la matrice de transformation et montre une violation d'un concept fondamental de l'orienté objets; principe d'encapsulation non respecté. (le tableau est public)

Un exemple de *Saro* : est présenté dans la figure ci-dessous, elle représente un lien géométrique du robot. On remarque qu'il est composé d'un tableau de faces non privé (*private).* De plus les faces sont initialisées dans la classe, pour les modifier il faut toucher au code source.

```
public class Link_3b extends Polyhedron {
    static int[][] faces= new int[12][3];

    static {
      faces[0][2] = 2; faces[0][1] = 3; faces[0][0] = 0;
      faces[1][2] = 1; faces[1][1] = 2; faces[1][0] = 0;
      faces[2][2] = 5; faces[2][1] = 1; faces[2][0] = 0;
      faces[3][2] = 4 ; faces[3][1] = 5; faces[3][0] = 0;
      faces[4][2] = 6; faces[4][1] = 2; faces[4][0] = 1;
      faces[5][2] = 5; faces[5][1] = 6; faces[5][0] = 1;
      faces[6][2] = 7; faces[6][1] = 3; faces[6][0] = 2;
      faces[7][2] = 6; faces[7][1] = 7; faces[7][0] = 2;
      faces[8][2] = 4; faces[8][1] = 0; faces[8][0] = 3;
      faces[9][2] = 7; faces[9][1] = 4; faces[9][0] = 3;
      faces[10][2] = 4; faces[10][1] = 7; faces[10][0] = 6;
      faces[11][2] = 4; faces[11][1] = 6; faces[11][0] = 5;
    }

    public Link_3b(float w) {
        super(8);
        float w2 = w/2;

       points[0].v[0] = 87.5f; points[0].v[1] = -25f; points[0].v[2] = 27.5f;
       points[1].v[0] = 97.5f; points[1].v[1] = -25f; points[1].v[2] = 27.5f;
       points[2].v[0] = 97.5f; points[2].v[1] = -25f; points[2].v[2] = 37.5f;
       points[3].v[0] = 87.5f; points[3].v[1] = -25f; points[3].v[2] = 37.5f;
       points[4].v[0] = 87.5f; points[4].v[1] = -15f; points[4].v[2] = 27.5f;
       points[5].v[0] = 97.5f; points[5].v[1] = -15f; points[5].v[2] = 27.5f;
       points[6].v[0] = 97.5f; points[6].v[1] = -15f; points[6].v[2] = 37.5f;
       points[7].v[0] = 87.5f; points[7].v[1] = -15f; points[7].v[2] = 37.5f;
    }

    public Link_3b() {
        this(1.f);
    }

    public int getPolygonCount() {
        return 12;
    }

    public Polygon3D transformPolygonToEye(int f, ViewTransformer
viewTransformer) {
        Polygon3D poly = new Polygon3D(color, 4);

        poly.addPoint(viewTransformer.transform(points[faces[f][0]]));
        poly.addPoint(viewTransformer.transform(points[faces[f][1]]));
        poly.addPoint(viewTransformer.transform(points[faces[f][2]]));

        return poly;
    }
}
```

Un autre exemple toujours de *Saro* :

```
// ...
public class Model extends Applet implements Runnable {

   public Model () {
   }

   public void init() {
      // ...

         VisualWorld vw = new VisualWorld();
         vw.setBgColor(bgColor);


         base = new Base(50.f);
         base.setColor(new Color(0, 0, 255));
         vw.add(base);

         link_1a = new Link_1a(50.f);
         link_1a.setColor(new Color(255, 0, 0));
         vw.add(link_1a);

         link_1b = new Link_1b(50.f);
         link_1b.setColor(new Color(255, 0, 0));
         vw.add(link_1b);

         link_2a = new Link_2a(50.f);
         link_2a.setColor(new Color(255, 255, 0));
         vw.add(link_2a);

         link_2b = new Link_2b(50.f);
         link_2b.setColor(new Color(255, 255, 0));
         vw.add(link_2b);

         link_3a = new Link_3a(50.f);
         link_3a.setColor(new Color(0, 255, 0));
         vw.add(link_3a);

         link_3b = new Link_3b(50.f);
         link_3b.setColor(new Color(0, 255, 0));
         vw.add(link_3b);

         wrist = new Wrist(50.f);
         wrist.setColor(new Color(220, 220, 220));
         vw.add(wrist);

         cube2a = new Cube(5);
         cube2a.setColor(new Color(0,0,0));
         vw.add(cube2a);

      // ...
   }
}
```

Dans le code ci-dessus on remarque que les différents liens géométriques du robot n'appartiennent pas à la même classe, donc pour changer le robot il faut changer ces classes , de même pour rajouter un nouveau lien il faut écrire une nouvelle classe pour le représenter .
On remarque une violation d'un autre concept fondamental de l'orienté objets; principe de la réutilisation des objets.

Un exemple de robosim :

```
// ...
public class RoboProto extends Frame
{

      // Declaration of the graphics constants:
      public final static int numberOfPoints = 154;
      public final static int numberOfEdges = 211;
      public final static int numberOfAreas = 100;

      //Declaration of the variables specifying the robot:
      public AreasPointsList[] apl = new AreasPointsList[numberOfAreas];
      public Points light = null;
      public long[][] colorTable = new long[7][31];
      public double[] linkLengths = new double[4];
      public int[] angleWeights = new int[6];
      public int[] power = new int[6];
      public AngleDates[] angles = new AngleDates[6];
      public Points[] line = new Points[2];
      public Edges gkl = null;
      public int[] rpl = new int[14];
      public Points[] coorpl = new Points[4];
      public Edges coorc[] = new Edges[3];
      public Points[] pl = new Points[numberOfPoints];
      public Points[] PL = new Points[numberOfPoints];//vertices of the
robot
      public Edges[] el = new Edges[numberOfEdges];   //edges of the robot
      public Areas[] al = new Areas[numberOfAreas];

      // ...
}
```

Dans les deux exemples précédents (saro et *robosim*), on constate qu'il n'y a pas de séparation entre le robot et le *UI* (*User Intertface*).tel que :

- La classe de Saro est une applet (`public class Model extends Applet`).

- La classe de Robosim est une fenêtre (`public class RoboProto extends Frame`).

De là, on déduit que le classe du robot (Model dans saro , et RoboProto dans robosim) ne peut pas être réutilisable dans un autre système (il y a une dépendance entre les variables du robot et celles de l'interface utilisateur *UI*).

Les codes sources ne sont que quelques exemples sur les problèmes que nous avons détecté. En effet, pour plus de détails voir le chapitre10 (Conception et Réalisation du système).

Le but de notre travail est d'essayer de résoudre ces problèmes, en utilisant les nouvelles approches de développement telles que les méthodes Agiles, qui favorisent l'utilisation des patterns et de l'orienté objet. (donc adopter une solution patterns et orienté objet)

III-3. DEFINITION DES BESOINS

Dans cette première étape, nous utilisons le diagramme des cas d'utilisation de UML[1]. Le résultat de cette étape est une description sommaire de ce que doit faire notre système.

Dans le cas de notre étude, le diagramme des cas d'utilisations s'organise en deux modules fonctionnels, ceux introduits ci-dessous. Il fait intervenir deux types d'utilisateurs : administrateurs du système ou pas.

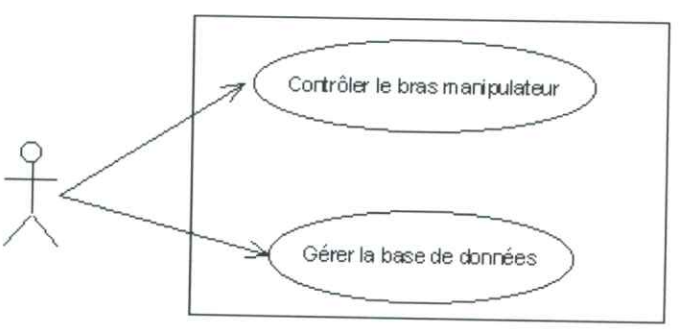


Figure III.1

[1] Voir Annexe D

Dans ce qui suit, on présentera les cas d'utilisations correspondants à chaque module.

- ***Le contrôle du bras manipulateur:***

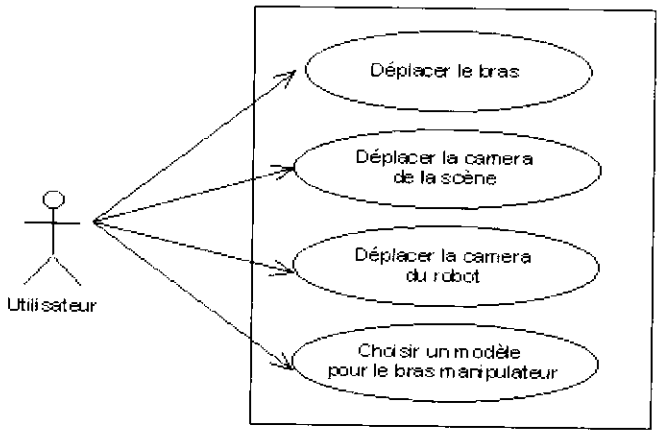


Figure III.2

L'utilisateur en question peut être soit un simple utilisateur ou un administrateur. Le système permet à l'utilisateur de :

- ✓ déplacer le bras manipulateur;
- ✓ déplacer la caméra de la scène;
- ✓ déplacer la caméra du robot;
- ✓ choisir un modèle de robot.

- **Le déplacement du bras manipulateur :**

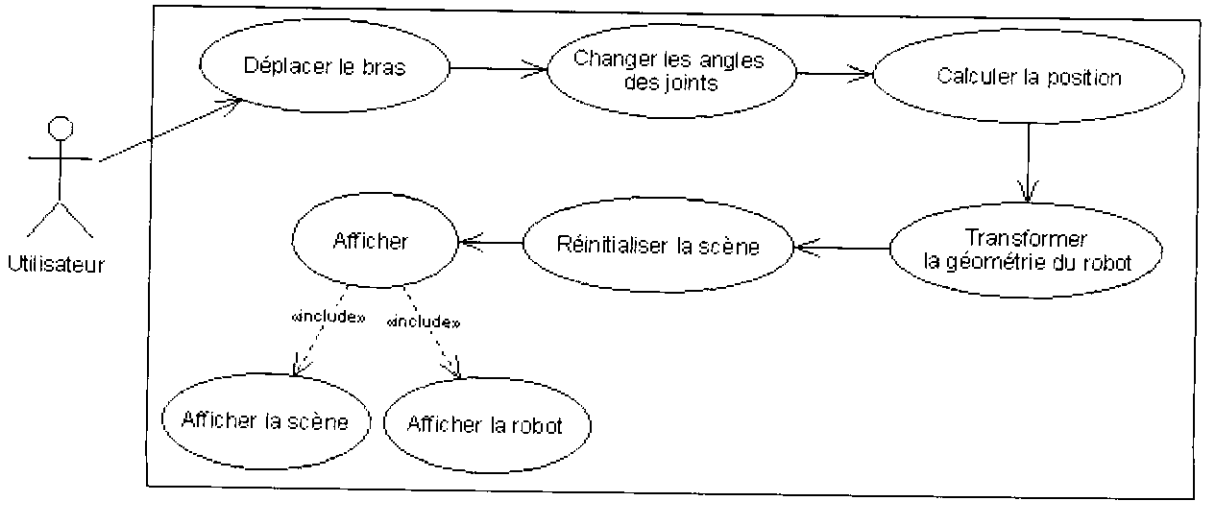


Figure III.3

Pour déplacer le bras manipulateur, il faut :

- Changer les angles au niveau des articulations,
- Calculer la nouvelle position du bras[2].
- Transformer la géométrie du robot.
- Réinitialiser la scène
- Afficher dans les deux vues.

Il faut que les différentes modifications qui peuvent être effectuées sur le bras ne puissent toucher qu'à sa présentation et non pas sa structure.

- **Le déplacement de la caméra de la scène :**

Après le déplacement de la caméra[3], il faut :

- Transformer la géométrie du robot et la géométrie de la caméra.
- Réinitialiser la scène puis afficher les deux vues[4].

[2] Résoudre la cinématique directe.
[3] Le déplacement de la caméra induit le changement de la vue de cette dernière.
[4] La vue de la caméra de la scène et la vue de la caméra du robot.

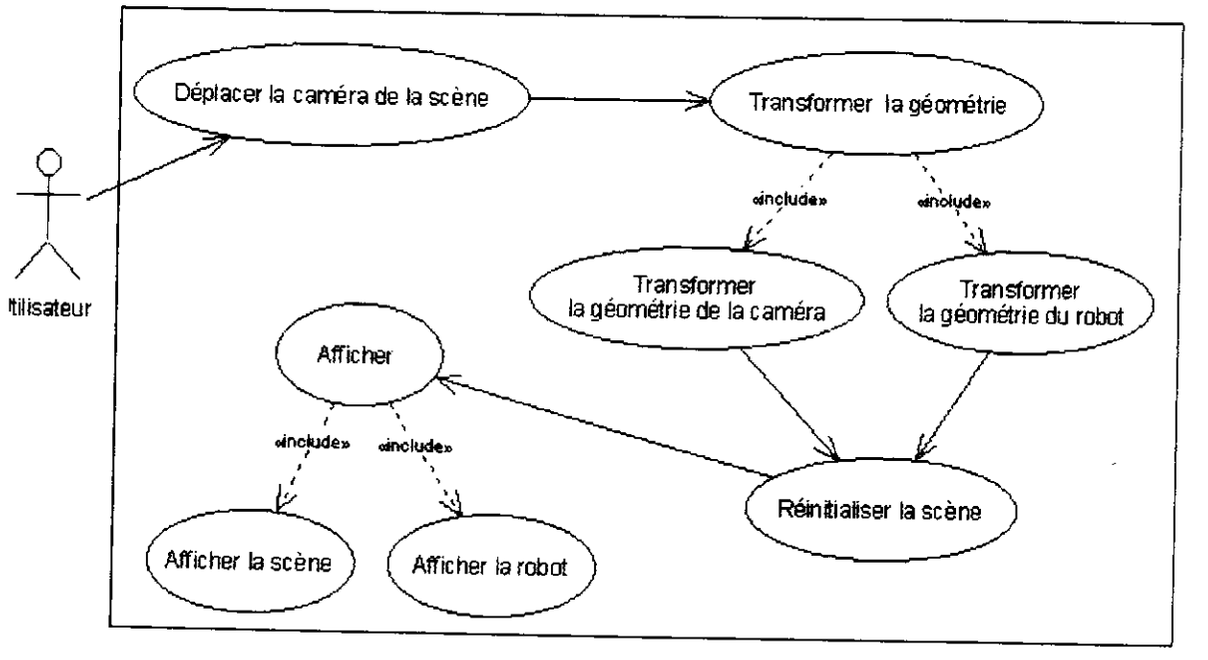


Figure III.4

- **Le déplacement de la caméra du robot :**

Après le déplacement de la caméra, il faut :

- Transformer la géométrie du robot[5] et la géométrie de la caméra[6].
- Réinitialiser la scène puis afficher les deux vues.

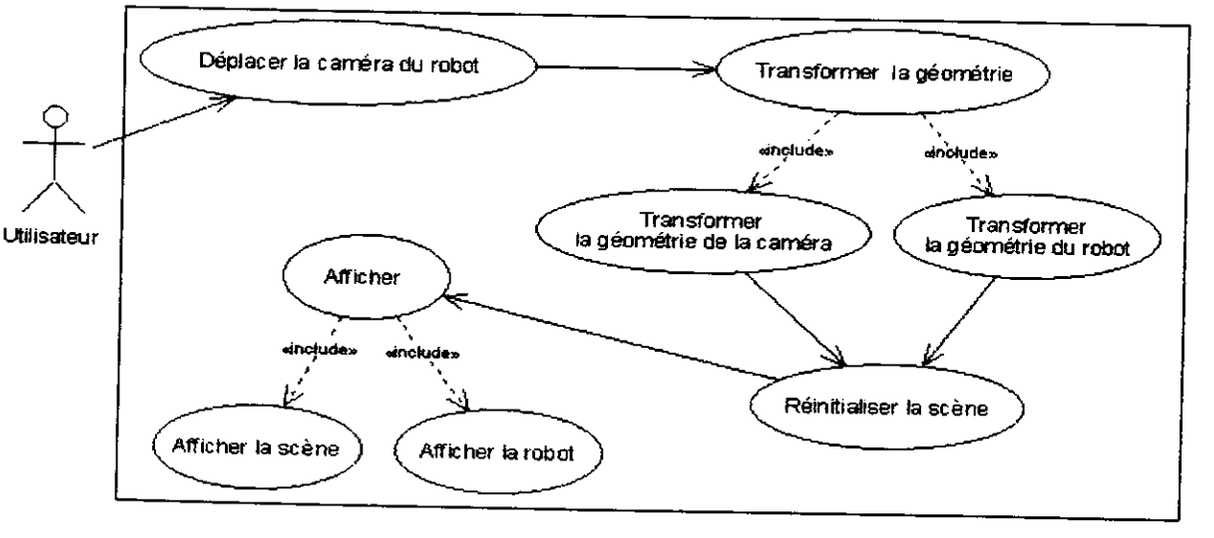


Figure III.5

[5] Appliquer la vue de la caméra du robot sur la géométrie du robot pour le réafficher dans la fenêtre du robot.
[6] Appliquer la vue de la caméra de la scène sur la géométrie de la caméra pour la réafficher dans la fenêtre de la scène.

- **Le choix du modèle :**
  - Après le choix du modèle[7], il faut :
  - Calculer la position initiale du bras8.
  - Transformer la géométrie du robot.
  - Réinitialiser la scène puis afficher dans les deux vues.

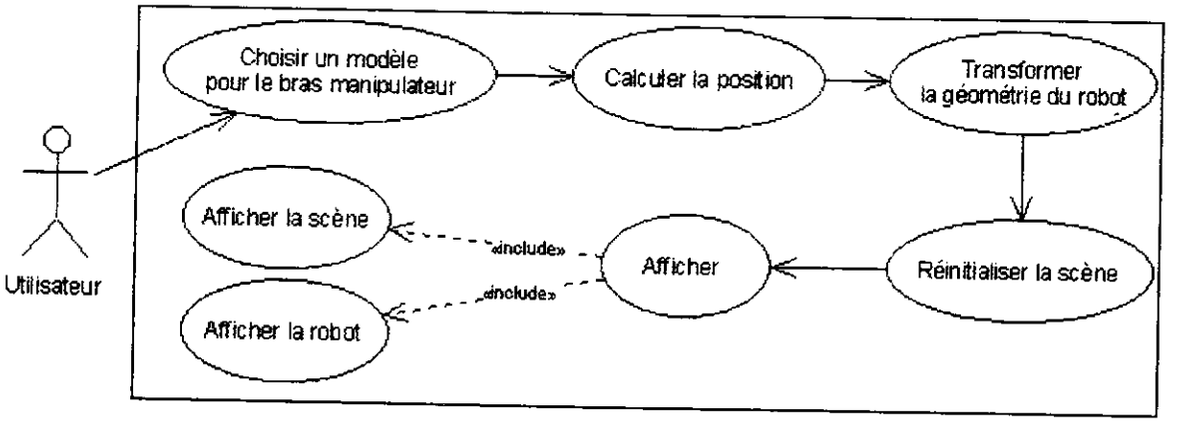


Figure III.6

Après l'intégration des différents cas d'utilisations ci-dessus, on obtient :

[7] La structure du robot est initialisée par une nouvelle, à partir d'une base de données.
[8] Résoudre la cinématique directe.

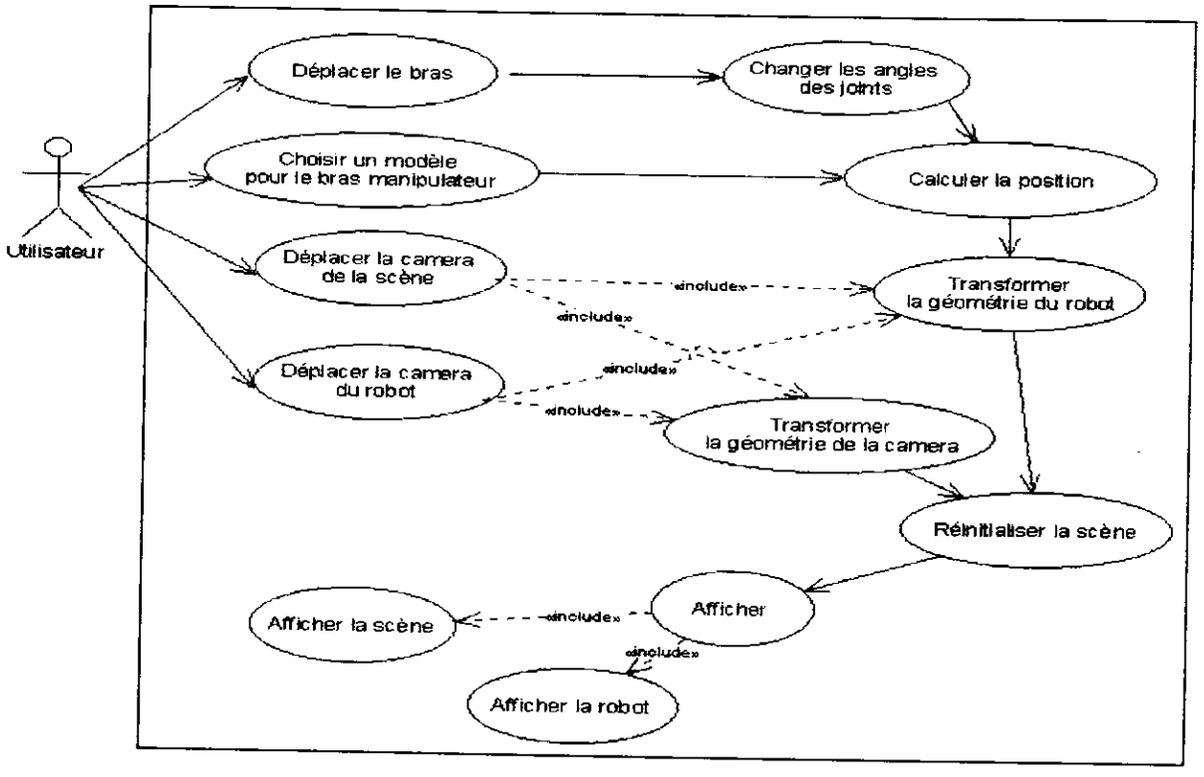


Figure III.7

- ***La gestion de la base de données :***

Seul l'administrateur[9] qui peut effectuer ces tâches : Rajout d'un nouveau modèle dans la base de données, la suppression ou la modification d'un modèle existant dans la base de données.

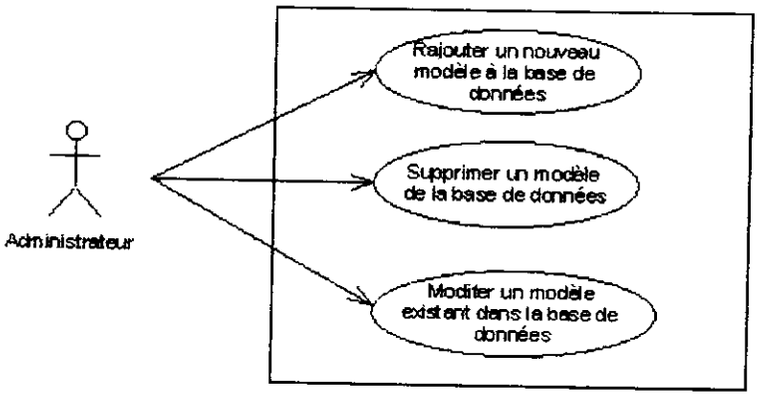


Figure III.8

[9] Pour garantir la cohérence des informations de la base de données.

CHAPITRE IV

CONCEPTION ET RÉALISATION DU SYSTÈME

IV-1. INTRODUCTION

Dans cette partie nous allons aborder notre travail. Dans la pratique nous avons suivi la méthode de modélisation XP[1], méthodes itératives et incrémentales. Pour présenter notre système on a choisi le langage UML [7]; Il s'articule autour de neuf diagrammes différents, représentant le système selon deux modes : L'un concernant sa structure et l'autre sa dynamique de fonctionnement.

Pour modéliser et concevoir notre système on a utilisé cinq des neufs diagrammes:

- Les cas d'utilisation (use case), pour définir les besoins de notre système.
- Le diagramme des classes, pour représenter son architecture (vue statique).
- Le diagramme d'objets, pour représenter son architecture (vue dynamique).
- Le diagramme de séquence, pour représenter les interactions entre les objets (vue dynamique).
- Le diagramme de collaboration, pour détailler les interactions présentées dans les diagrammes de séquence.

Pour l'implémentation de notre système on a utilisé le langage de programmation Java.

IV-2. L'ARCHITECTURE DU SYSTEME

Le résultat de cette étape est une description générale de l'architecture du système permettant d'identifier les différents composants et leurs liens.

[1] Voir Annexe C

IV-2.1. La vue statique du système

Dans cette deuxième étape, nous utilisons le diagramme des classes pour identifier, à partir du cahier des charges, les principales composantes du système à construire. Le diagramme de classe nous permet de modéliser, d'une manière statique, les relations qui existent entre l'ensemble de classes. Il développe d'une part la structure des entités du système et d'autre part celle d'un code orienté objet.

Afin de permettre au lecteur de bien comprendre notre système, nous présentons dans ce qui suit une vue d'ensemble sur l'architecture du système. **(Figure IV.1).** Ensuite nous allons oeuvrer à décomposer et à détailler chaque composant à part.

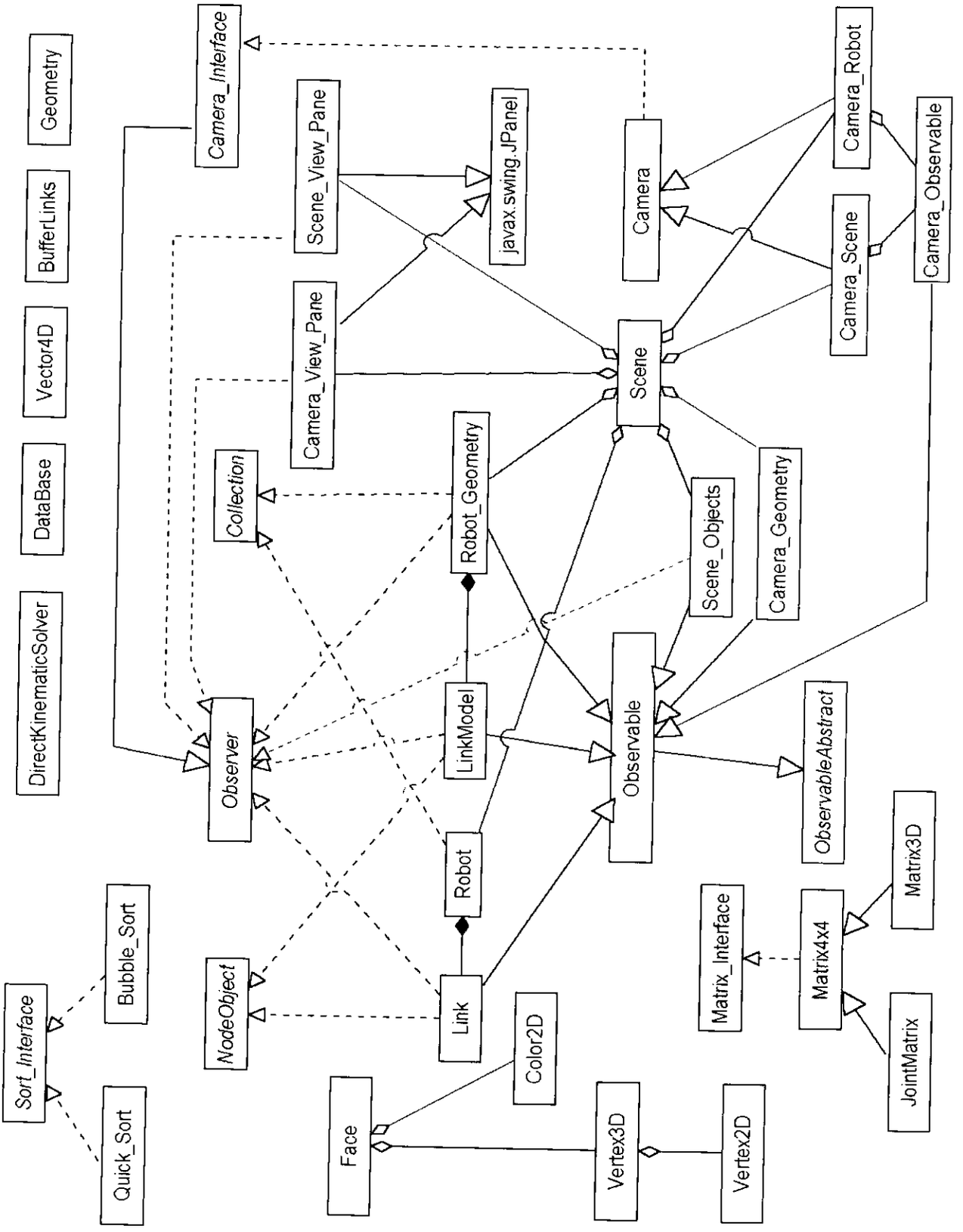

Geometry
BufferLinks
Vector4D
DataBase
DirectKinematicSolver
Camera_Interface
Scene_View_Pane
javax.swing.JPanel
Camera
Camera_Robot
Camera_Observable
Camera_Scene
Camera_View_Pane
Scene
Collection
Robot_Geometry
Scene_Objects
Camera_Geometry
Observer
LinkModel
Observable
ObservableAbstract
Robot
Matrix3D
Matrix_Interface
Matrix4x4
JointMatrix
Bubble_Sort
Sort_Interface
Quick_Sort
NodeObject
Link
Color2D
Face
Vertex3D
Vertex2D

Figure IV.1

IV-2.1.1. La classe Observable :

L'utilisation du patron de conception *Observer* de java nous a permis de comprendre son fonctionnement et son architecture. Dans cette section on va détailler l'architecture de notre propre patron de conception *Observer* en se basant sur les principes de base de ce dernier[1].

La classe Observable doit avoir une référence sur une liste d'observateurs. Lors d'une notification cette dernière est parcourue pour envoyer à chaque observateur la fonction update().

Dans notre cas cette liste est définie par l'interface *Collection*, elle peut être initialisée par toute liste qui implémente *Collection.*

La classe *(Observable)* hérite de la classe abstraite *(ObservableAbstract)* **(Figure IV.2).** L'utilisation d'une classe générale *(ObservableAbstract)* permet aux utilisateurs d'implanter l'Observé à leur manière, ils peuvent par exemple utiliser un tableau ou une autre structure de données au lieu de notre liste. On peut remarquer qu'on peut utiliser une interface au lieu d'une classe abstraite, mais on ne l'a pas utilisé car toutes ses méthodes sont *public*[2] et la méthode « *notifyObservers(Observable)* » doit être protégée (*protected*[3]).

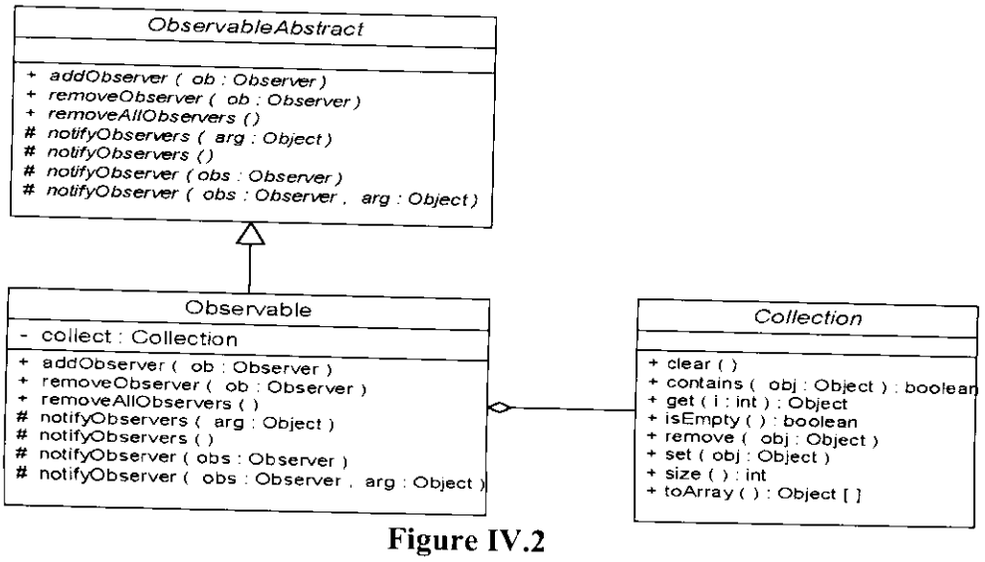


Figure IV.2

[1] Voir Annexe B.
[2] Voir Annexe A.
[3] Voir Annexe A.

IV-2.1.2. Le module du robot :

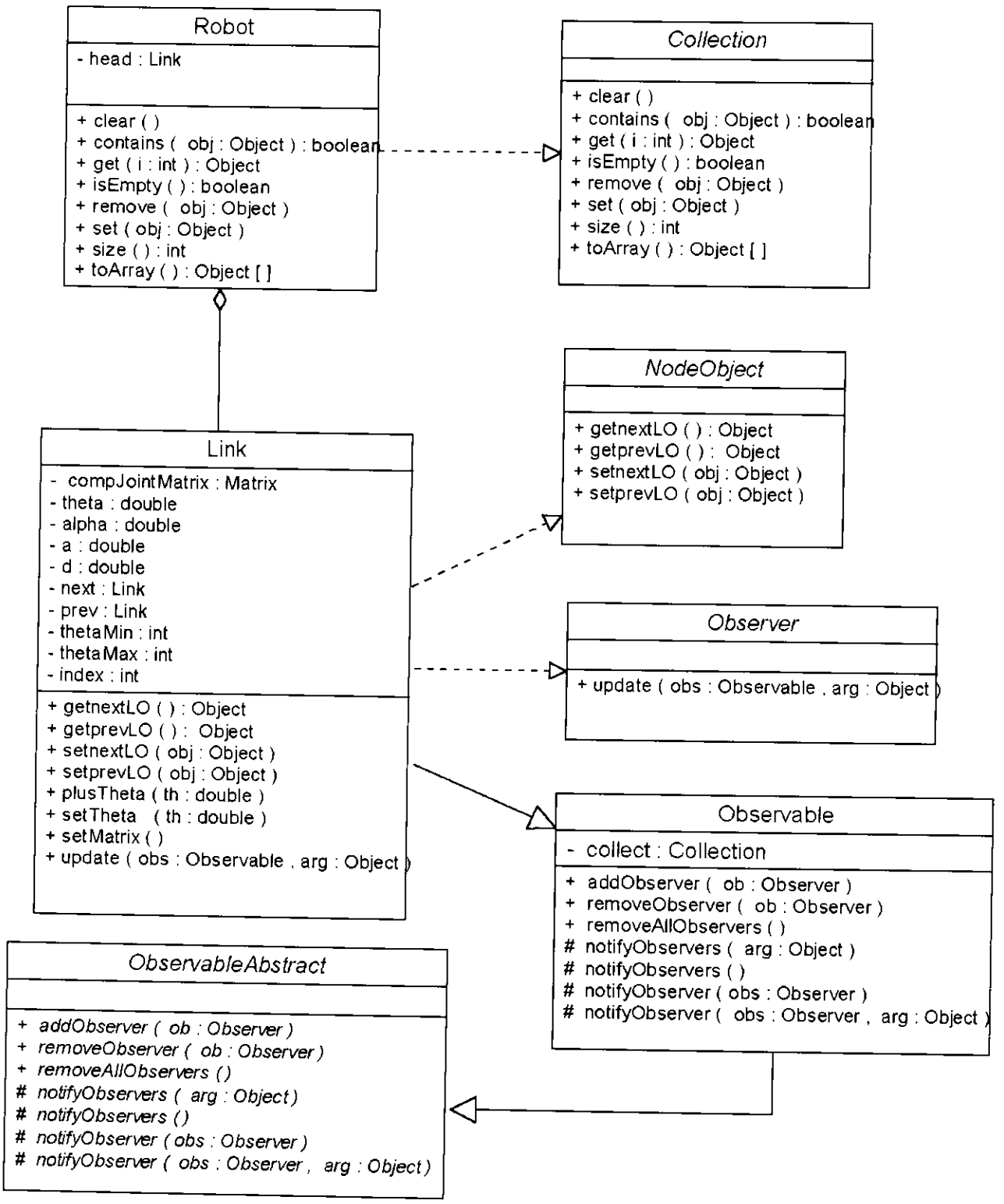


Figure IV.3

Le diagramme **(Figure IV.3)** représente le robot comme une liste de *liens (Link)*, il implémente l'interface *(Collection)*.

Théoriquement lorsqu'on change l'angle d'un lien on recalcule sa matrice de transformation[1], et ce changement se répercute sur tous les successeurs, les prédécesseurs restent inchangés. Exemple : Si on change le lien 4 alors il faut recalculer les matrices de lien 5, et lien 6, les autres : lien 0, lien 1, lien 2, lien 3 restent inchangés.

Le problème qui peut se poser est comment trouver les liens qui doivent changer sans parcourir toute la liste des liens.

Exemple : les logiciels *robosim, Gras* et *saro*[2] utilisent un tableau de lien, pour calculer la matrice de transformation du lien *i* ils doivent toujours parcourir la totalité du tableau[3] pour refaire le calcul de la cinématique à chaque fois à partir du lien 0 car ils ne maintiennent pas la matrice de transformation (C'est la matrice *T* de transformation qui résulte de la multiplication des matrices *DH* de tous les prédécesseurs) de chaque lien. De plus, ils n'ont aucune information sur lien qui a changé avant de parcourir le tableau et vérifier un à un leurs états.

Notre solution est basée sur l'utilisation du patron de conception « Observer » entre les liens tel que chaque lien est observé (Observable) par son successeur et observateur (Observer) de son prédécesseur. De plus on a choisi de maintenir la matrice composite de chaque lien.

Si on change l'angle d'un lien son successeur sera automatiquement informé, il récupère la matrice composite du lien qui a changé pour mettre à jours sa matrice de transformation. Puis il informe à son tour son successeur, et cela jusqu'au dernier lien du robot.

Donc un lien possède les informations concernant la cinématique :

- *alpha, angle de torsion,*
- *thêta, l'angle d'articulation* θ,
- *a, la distance sur la normale commune, qui sépare deux axes d'articulations,*
- *d, la distance qui sépare deux normales adjacentes,*
- *thetaMin*, la valeur minimale que peut prendre theta,
- *thetaMax*, la valeur maximale que peut prendre theta,

[1] Voir les matrices An et Tn dans le chapitre I.
[2] Voir chapitre II
[3] C'est un tableau de lien.

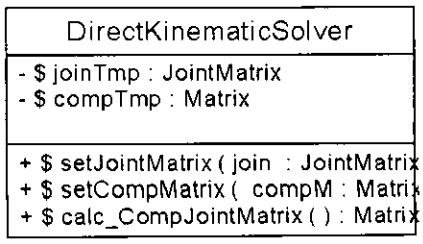
- une matrice *compJointMatrix* qui sert à sauvegarder sa matrice composite et des références sur son successeur et son prédécesseur : *next* et *prev,*

Un lien est un observateur, il doit implémenter l'interface *Observer.* Il est aussi observé, il doit hériter de *Observable.* Enfin, le lien est un élément du robot (la liste) il implémente l'interface *NodeObject.*

IV-2.1.3. La classe DirectKinematicSolver :

Dans cette section on représente la classe *DirectKinematicSolver* qui est utilisée pour résoudre la cinématique directe. Exemple : dans les logiciels comme *robosim et Gras* cette classe est définie comme suit : Elle contient un tableau initialisé par les liens du robot. Les méthodes de *DirectKinematicSolver* utilisent ce dernier pour résoudre la cinématique directe. Le logiciel *saro* n'a même pas de classe pour *DirectKinematicSolver,* il a quelques méthodes dans la classe principale (main) qui font les calculs, on remarque l'absence de la séparation entre l'UI (User Interface) et les traitements.

Notre solution est représentée dans le diagramme ci-dessous. Grâce à l'utilisation du patron de conception *Observer* entre les liens notre *DirectKinematicSolver* se réduit à un simple multiplicateur. C'est un outil indépendant des autres classes, il est utilisé par les liens pour calculer leurs matrices composite, il possède deux matrices de travail une sera initialisée par la matrice *DH* du lien appelant et l'autre sera initialisée par la de transformation de son prédécesseur. Le résultat est obtenu par une simple multiplication de ces deux matrices.



IV-2.1.4. Le module : géométrie du robot :

La question qui peut se poser est comment représenter la géométrie du robot. Exemple : le logiciel *robosim* utilise un tableau contenant toutes les coordonnées de la géométrie (sommet, arêtes,...) du robot. Lors du changement d'angle d'un *Link* on recalcule sa matrice de transformation, cette dernière est utilisée pour transformer[1] la géométrie associée à ce lien. Le problème qui se pose est : comment peut on trouver les coordonnées de cette géométrie, il n'y a pas une séparation entre les coordonnées des géométries des différentes parties du robot. *Le logiciel saro* utilise une classe pour chaque partie géométrique, par exemple : classe Base, class Coude, class Epaule,...etc. Puis, il utilisé dans chaque classe un tableau contenant les coordonnées géométriques. Cette méthode résout le problème de séparation des modèles géométriques mais son inconvénient est la nécessité de modifier ces classes à chaque changement du robot, de plus il faut avoir autant de classes des liens géométriques que de nombre des liens dans le robot.

La solution qu'on a adopté consiste à représenter la géométrie du robot par une liste de modèles géométriques.

Un question se pose : comment associer une géométrie de lien (*LinkModel)* à un lien ?

Pour remédier à ce problème on utilise le patron de conception Observer entre les liens et *leurs géométries* tel que chaque lien est Observable par la géométrie qui lui est associé. Donc classe *LinkModel* est Observer de la classe *Link.*

Si on change l'angle d'un lien sa géométrie sera automatiquement informé, il récupère la matrice de transformation du lien qui a changé pour mettre à jours sa géométrie.

Le modèle géométrique du robot (La classe *Robot_Géométrie*) doit conserver une liste de toutes les faces, car dans l'étape du rendu on doit trier toutes les faces du robot en même temps afin d'éliminer les faces cachées du robot. Un changement dans l'un des liens géométriques du modèle géométrique du robot doit réinitialiser cette liste, c'est pourquoi la classe *LinkModel* est Observable par la classe *Robot_Geometry.*

[1] Il s'agit d'un ensemble de rotations et translations nécessaires pour atteindre la position souhaitée.

Là on pourrait signaler que l'orienté objet "stricte" est un peu délaissé parce qu'un objet tout seul ne saurai aucunement réaliser un affichage sans prendre en compte d'autres objets (c'est-à-dire d'autres lien) de la scène.

Le diagramme **(Figure IV.4)**, représente la géométrie du robot (classe *Robot_Geometry),* composée de modèles géométriques des différents liens (*LinkModels)*, et qui implémente l'interface *Collection*

Un modèle géométrique d'un lien implémente l'interface *Observer.* Il doit aussi hériter de la classe *Observable.* Enfin, il est un élément de la liste qui représente la géométrie du robot, il implémente donc l'interface *NodeObject.*

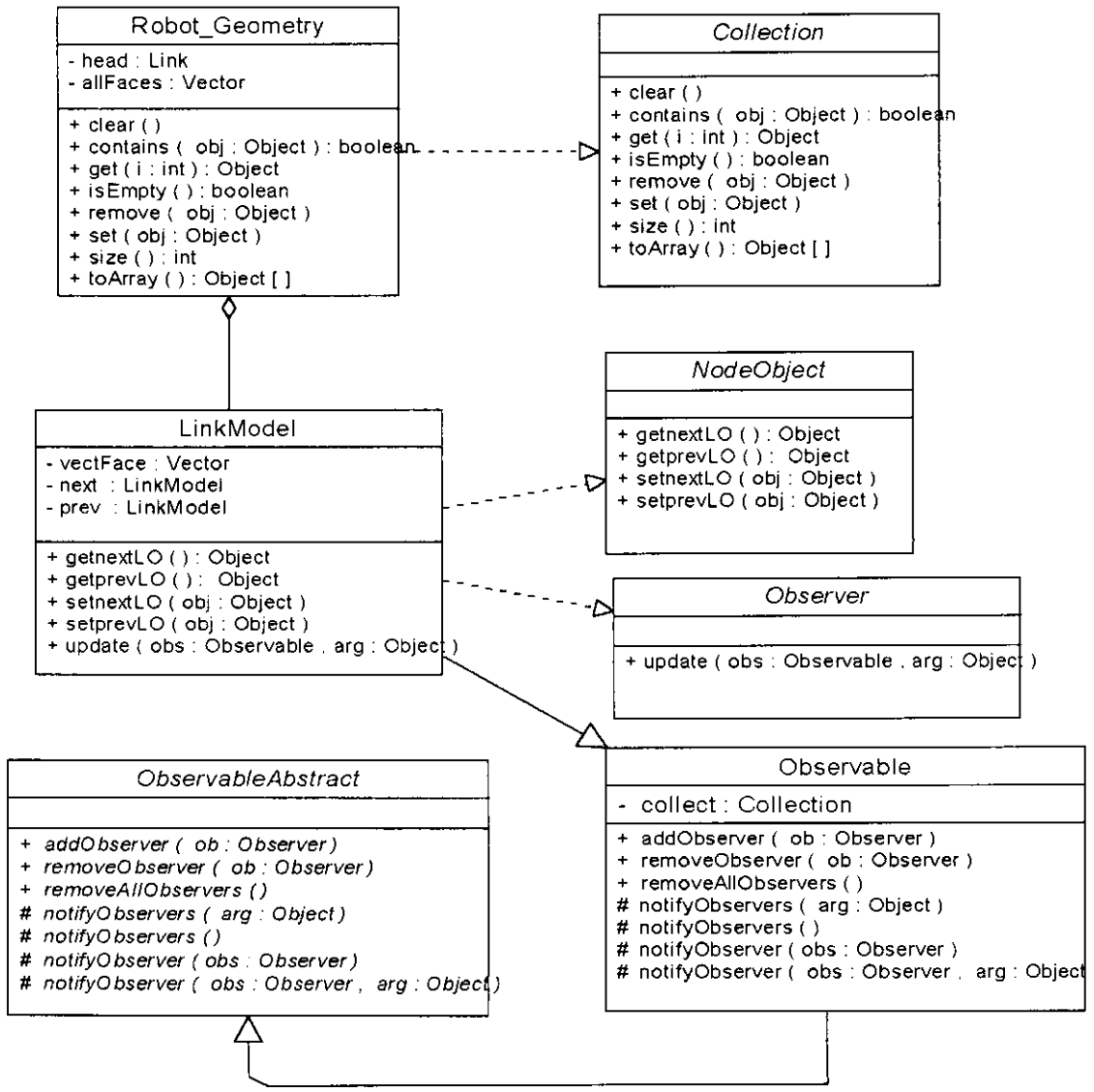


Figure IV.4

IV-2.1.5. Le module des caméras :

Dans notre réalisation nous avons voulu montrer que même en compliquant le système un peu plus, comme l'incorporation de deux caméras, on peut arriver à résoudre les problèmes qui se posent grâce aux patrons de conception.

Dans cette section on présente les deux caméras[1]. Exemple : les logiciels *robosim, Gras* et *saro* n'ont pas de classe *Caméra,* ils utilisent dans la classe principale du programme[2] des variables qui permettent de calculer la vue avant d'afficher le modèle géométrique. L'inconvénient remarqué est qu'il n'y a pas une séparation entre l'*UI (User Interface)* et les paramètres de la caméra, donc si on voudrait modifier la caméra d'affichage on doit refaire toute la classe de l'*UI.*

Pour résoudre ce problème, on propose l'utilisation d'une classe *Camera* indépendante, séparée de l'interface utilisateur, on doit utiliser le *pattern Observer* entre la *Camera* et le panneau d'affichage associé (couplage faible voir chapitre patterns); lorsque la camera change de position le panneau d'affichage réaffiche la géométrie selon la nouvelle vue.

De plus on propose l'utilisation du *pattern Strategy* qui nous permet de remplacer facilement la caméra utilisée par une autre[3].

Pour cela on utilise une interface *Strategy* qui contient tous les algorithmes d'une caméra *(Camera_Interface)*, puis notre classe Camera doit implémenter cette dernière. Lorsqu'on désire changer la classe *Camera*, on change uniquement au niveau de l'initialisation par une autre classe qui implémente *Camera_Interface* et le reste du programme reste inchangé (voir chapitre pattern).

[1] Caméra du robot et caméra de la scène
[2] C'est la classe de l'interface utilisateur UI.
[3] Car on a remarqué qu'il existe plusieurs manières d'implémenter une classe Camera.

Le diagramme ci-dessous **(Figure IV.5)**, représente la classe générale *Camera* qui implémente l'interface *Camera_Interface*, ainsi que deux autres classes : *Camera_Robot* et *Camera*_Scene qui héritent de la classe *Camera.*

Nos deux caméras doivent être observables par les panneaux d'affichages donc elles doivent hériter de Observable.

Un problème se pose à ce niveau, l'absence de l'héritage multiple en Java.

Pour remédier à ce problème, on utilise notre propre *pattern d'implantation (idiome)*[1] , la classe observée peut ne pas hériter de la classe *Observable* mais elle doit avoir une référence sur une classe qui hérite de *Observable,* cette dernière est représentée par la classe *Camera_Observable.* C'est elle qui s'occupe des opérations de notification des observers. On doit aussi rajouter une méthode *addObserver(Observer obs)*[2] dans les classes Camera_Scene et Camera_Robot qui attache les observers à *Camera_Observable.*

De plus nos deux caméras doivent avoir une liste des faces à transformer. La liste de la classe Camera_Scene contient les faces des objets de la scène (la géométrie d la caméra et le robot) et la liste de la classe Camera_Robot contient les faces du robot, si ce dernier change elle doit être mis à jours. Donc notre classe doit implémenter *Observer.* C 'est pourquoi l'interface *Camera_Interface* hérite de *Observer.*

[1] Voir Annexe B
[2] Lorsqu'on l'applique camera_Scene par exemple elle appelle addObserver(Observer) de Camera_Observable.

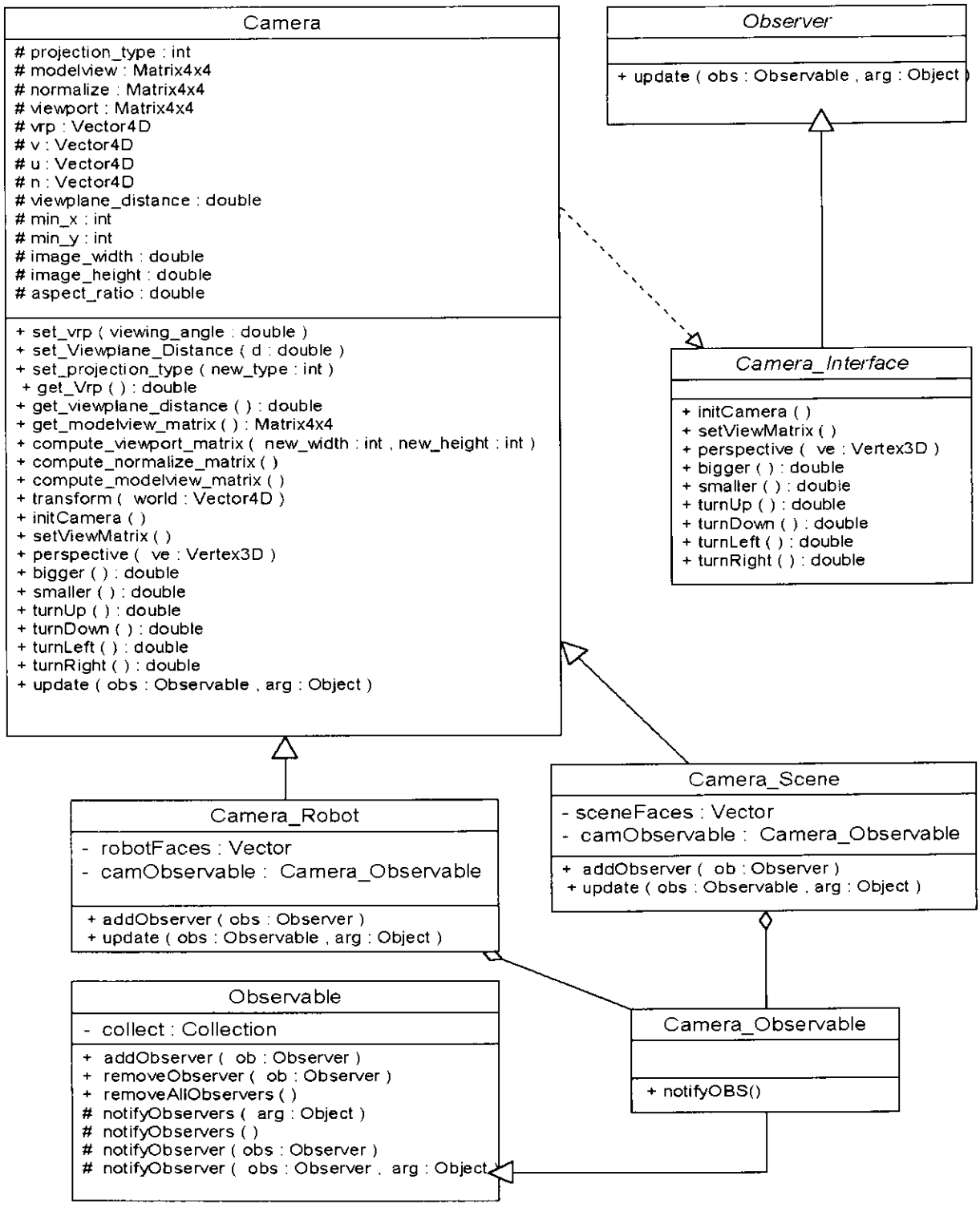

Camera
projection_type : int
modelview : Matrix4x4
normalize : Matrix4x4
viewport : Matrix4x4
vrp : Vector4D
v : Vector4D
u : Vector4D
n : Vector4D
viewplane_distance : double
min_x : int
min_y : int
image_width : double
image_height : double
aspect_ratio : double
+ set_vrp (viewing_angle : double)
+ set_Viewplane_Distance (d : double)
+ set_projection_type (new_type : int)
+ get_Vrp ( ) : double
+ get_viewplane_distance ( ) : double
+ get_modelview_matrix ( ) : Matrix4x4
+ compute_viewport_matrix (new_width : int , new_height : int)
+ compute_normalize_matrix ( )
+ compute_modelview_matrix ( )
+ transform (world : Vector4D)
+ initCamera ( )
+ setViewMatrix ( )
+ perspective (ve : Vertex3D)
+ bigger ( ) : double
+ smaller ( ) : double
+ turnUp ( ) : double
+ turnDown ( ) : double
+ turnLeft ( ) : double
+ turnRight ( ) : double
+ update (obs : Observable , arg : Object)
Observer
+ update (obs : Observable , arg : Object)
Camera_Interface
+ initCamera ( )
+ setViewMatrix ( )
+ perspective (ve : Vertex3D)
+ bigger ( ) : double
+ smaller ( ) : double
+ turnUp ( ) : double
+ turnDown ( ) : double
+ turnLeft ( ) : double
+ turnRight ( ) : double
Camera_Robot
- robotFaces : Vector
- camObservable : Camera_Observable
+ addObserver (obs : Observer)
+ update (obs : Observable , arg : Object)
Camera_Scene
- sceneFaces : Vector
- camObservable : Camera_Observable
+ addObserver (ob : Observer)
+ update (obs : Observable , arg : Object)
Observable
- collect : Collection
+ addObserver (ob : Observer)
+ removeObserver (ob : Observer)
+ removeAllObservers ( )
# notifyObservers (arg : Object)
# notifyObservers ( )
# notifyObserver (obs : Observer)
# notifyObserver (obs : Observer , arg : Object)
Camera_Observable
+ notifyOBS()

Figure IV.5

IV-2.1.6. La géométrie de la caméra :

La géométrie de la caméra (*Camera_Geometry*) d'après le modèle géométrique choisi, s'articule principalement autour de la liste de ses faces **(Figure IV.6)**. Elle est observée par Camera_Robot et Scene_Objects; elle hérite de *Observable*. Lorsqu'elle change de position elle notifie Camera_Robot pour qu'elle mette à jours ses paramètres de vue et elle notifie Scene_Objects pour qu'elle change sa liste des géométries.

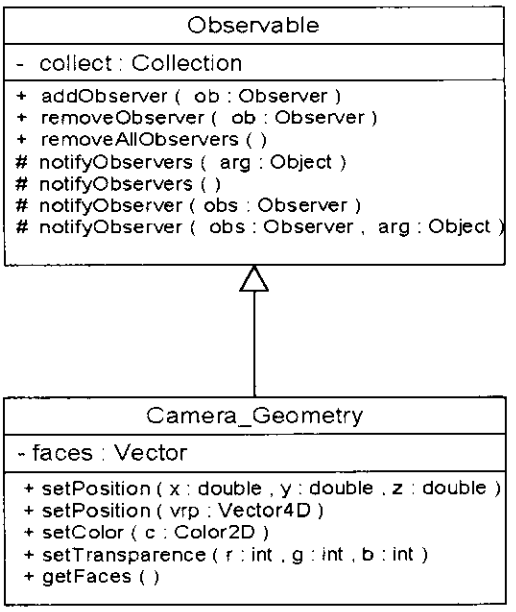
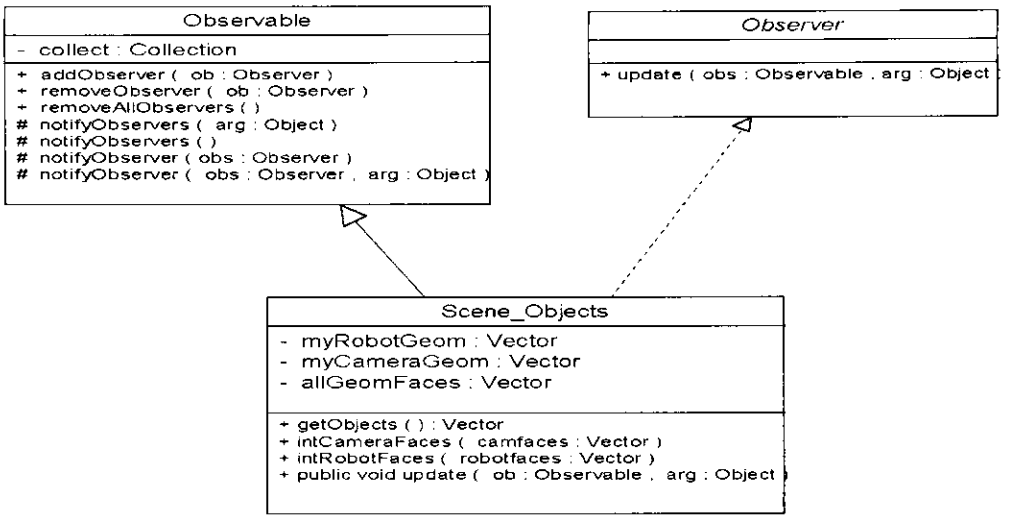
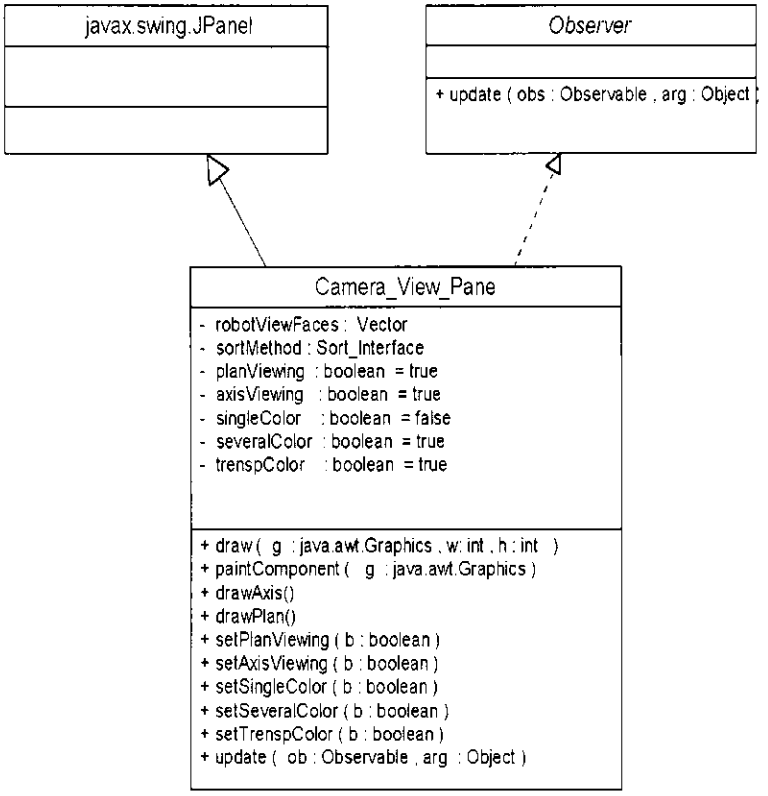


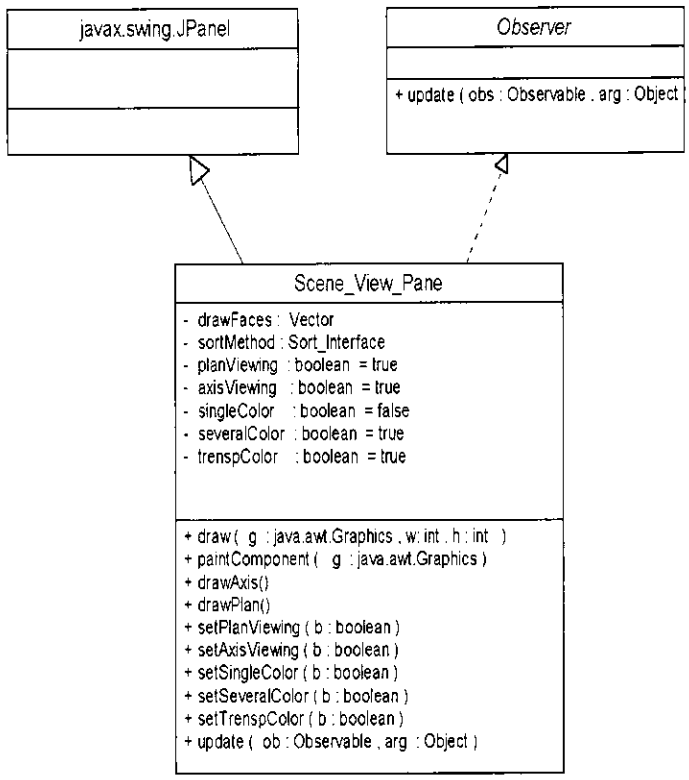
Figure IV.6

IV-2.1.7. Les objets de la scène :

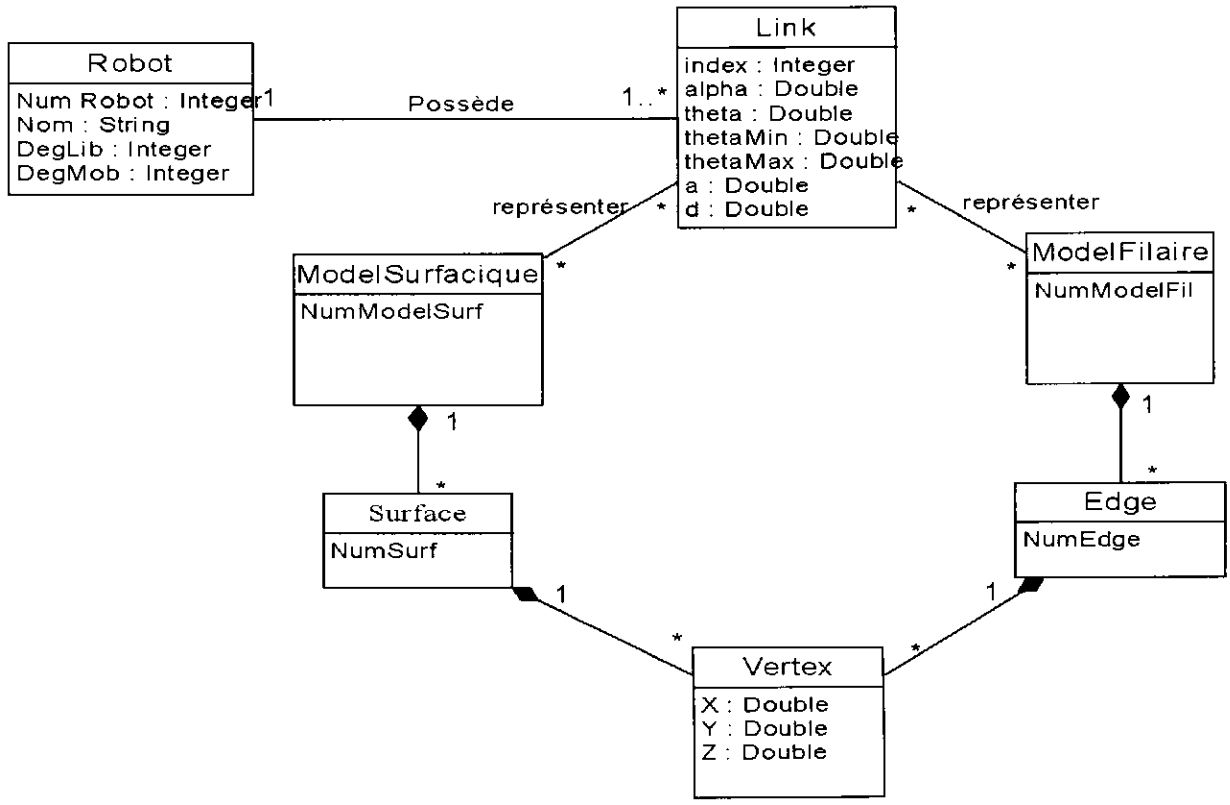


IV-2.1.8. Les panneaux d'affichage :





IV-2.1.9. La base données :



Nous présentons une vue d'ensemble sur les interactions entre les différents objets du système[1] dans le diagramme suivant :

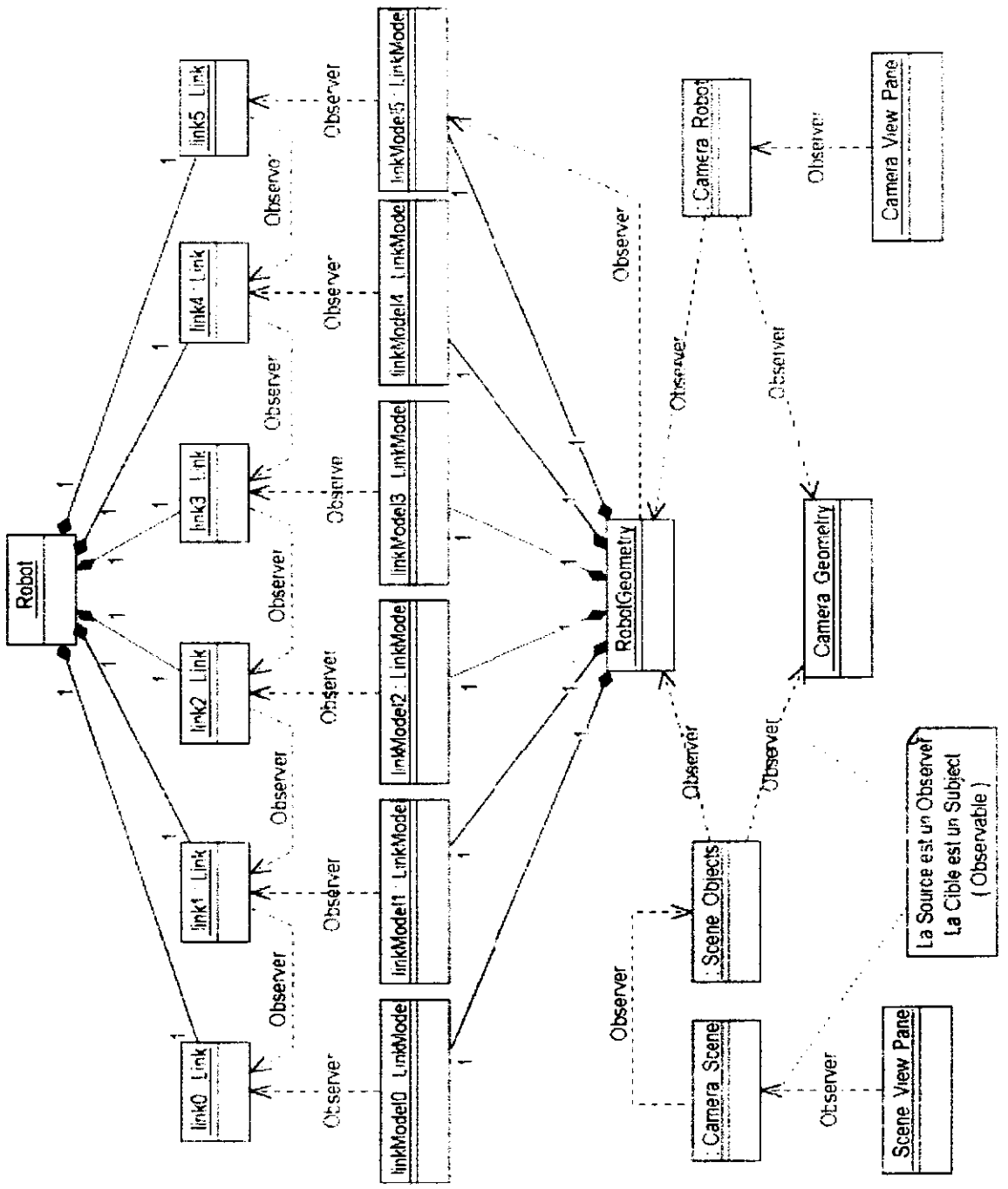


Figure IV.7

[1] Pas toutes les interactions mais seulement les plus importantes comme celles du pattern Observer.

IV-2.2. La vue dynamique du système :

Les diagrammes de séquence décrivent les interactions entre objets. Normalement, un diagramme de séquence sert à décrire le déroulement des événements d'un cas d'utilisation[1].

- ***Le contrôle du bras manipulateur :***

  - **Le déplacement du bras manipulateur :**

    Le diagramme suivant correspond au cas d'utilisation de changement des angles :

L'utilisateur change la valeur des angles au niveau des joints en glissant les *silders* associés dans l'interface utilisateur : *Joint_Slider_Frame*, ce changement va générer un évènement *stateChanged (ChangeEvent).* **(Figure IV.8)**

A la réponse de ce dernier, *Joint_Slider_Frame* envoie le message *setTheta(angle)* à la liaison *(Link)* concernée. Cette dernière met à jours son angle *Theta* puis elle exécute *notifyObservers()* qui notifie ses observateurs (le modèle géométrique associé (*LinkModel*) et lien suivant) du changement, en envoyant le message *update().***(Figure IV.8) et (Figure IV.9)**

Après la notification les observateurs (le modèle géométrique associé et lien suivant) récupèrent l'état du lien observé par l'exécution de *getMatrix()* pour changer leurs état interne[2].

[1] Dans notre cas on n'a pas cité toutes les interactions qui existent entre les objets, on s'est intéressé surtout aux messages du pattern Observer.
[2] Lien suivant change sa matrice DH, et sa géométrie associée change.

L'observer *Link* suivant notifie à son tour ses observers (son le modèle géométrique et son lien suivant), le même scénario se répète, l'envoie des évènement se répercute sur les liens et géométries associées jusqu'au dernier lien du *Robot.* **(Figure IV.9)**

Le dernier modèle géométrique notifié (*linkM5*) informe la géométrie du robot (*RobotGeometry*) pour mettre à jours sa géométrie.

Pour afficher, le *RobotGeometry* informe *Camera_Robot* et *Scene_Objects*, ces deux derniers récupèrent l'état du *RobotGeometry* en exécutant *getFaces()*.**(Figure IV.10)**

Ensuite la *Camera_Robot* transforme la liste des faces obtenue du *RobotGeometry*, puis elle l'envoie au panneau d'affichage du Robot : *Camera_View_Panel.*

De même la *Camera_Scene* récupère l'état de *Scene_Objects* en exécutant *getObjects*(), elle transforme la liste des faces des objets de la scène, puis elle l'envoie au panneau d'affichage de la scène : *Scene_View_Panel.* **(Figure IV.10)**

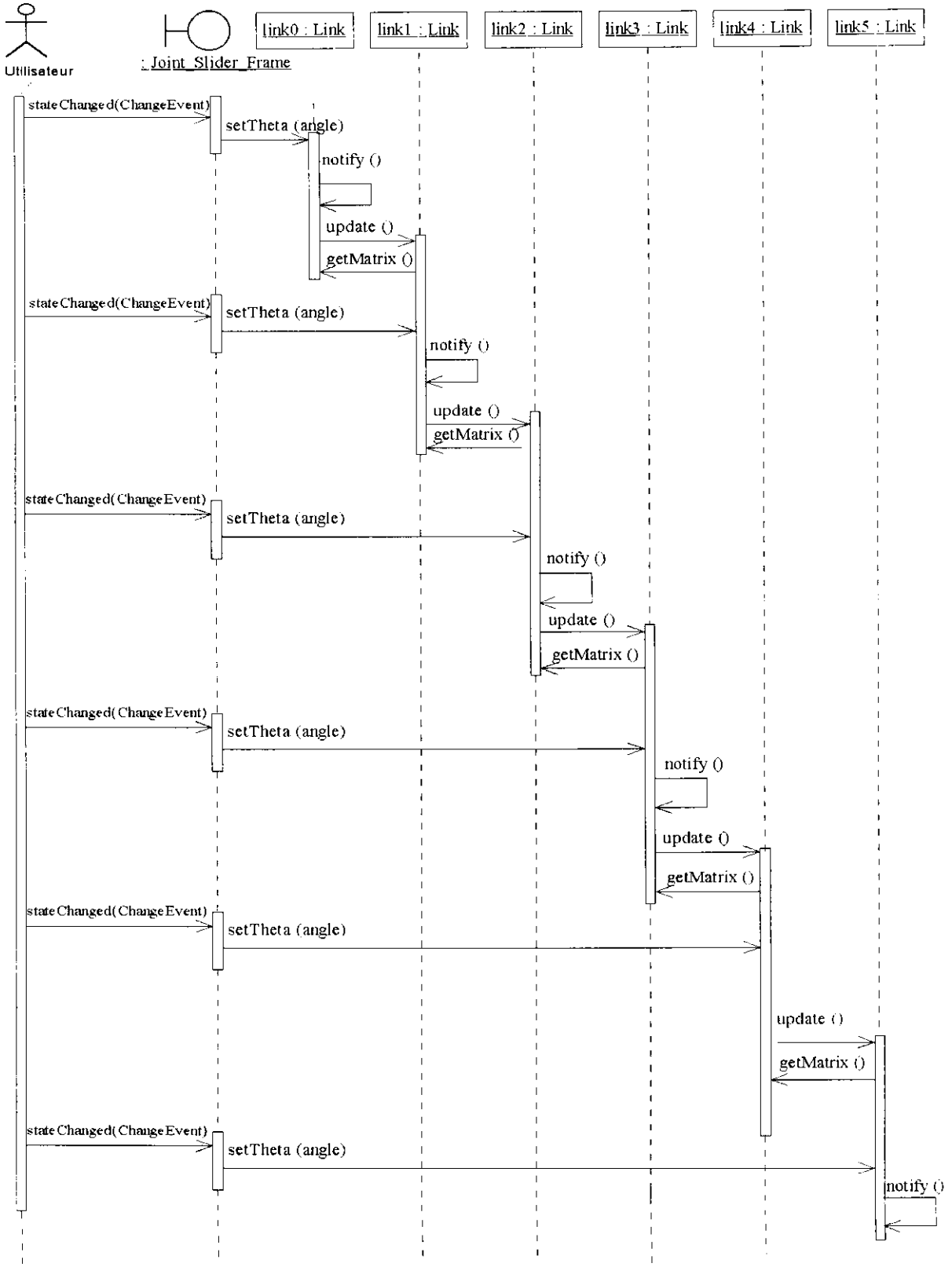

Utilisateur
: Joint_Slider_Frame
link0 : Link
link1 : Link
link2 : Link
link3 : Link
link4 : Link
link5 : Link
stateChanged(ChangeEvent)
setTheta (angle)
notify ()
update ()
getMatrix ()

Figure IV.8

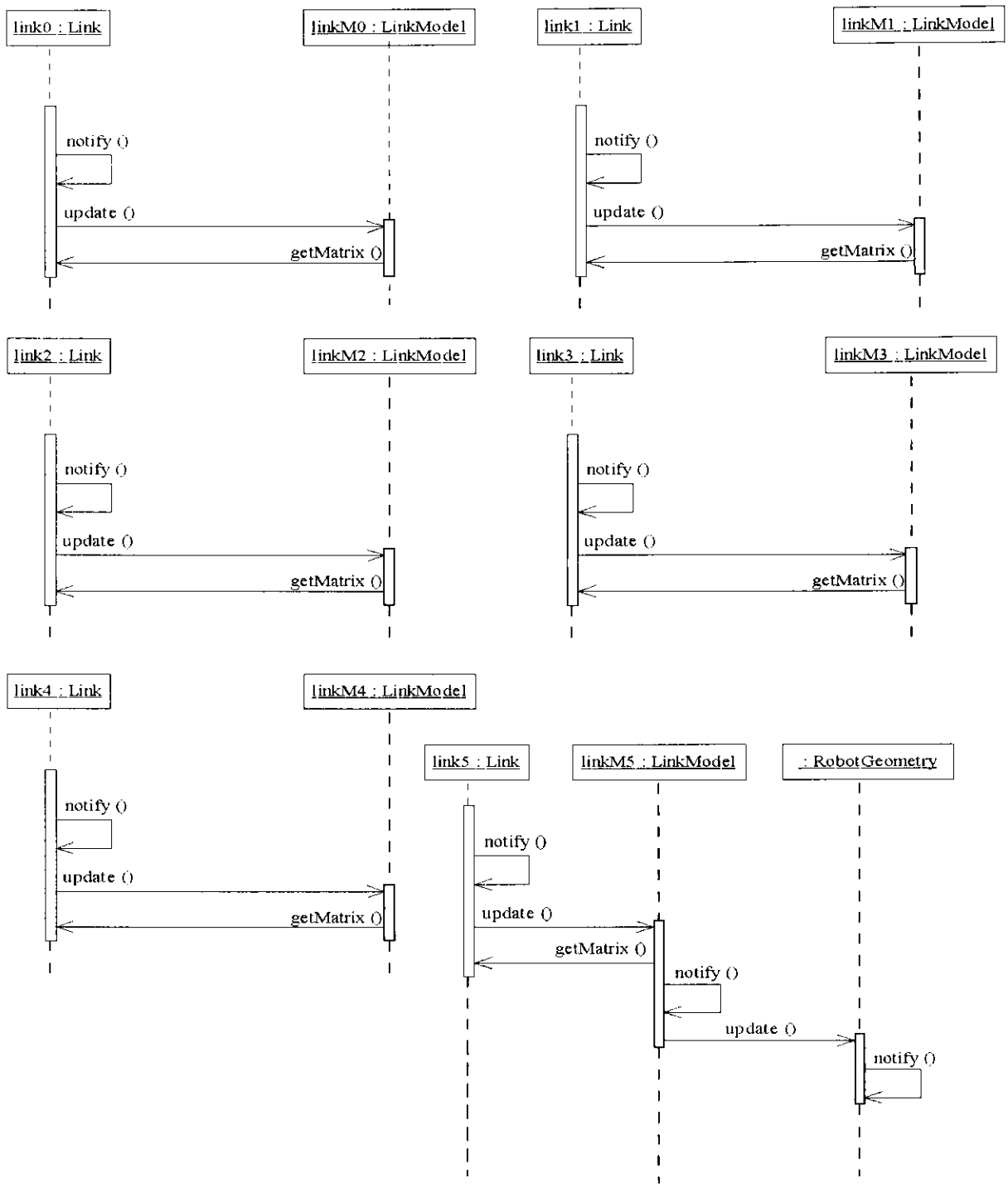

link0 : Link
linkM0 : LinkModel
notify ()
update ()
getMatrix ()
link1 : Link
linkM1 : LinkModel
notify ()
update ()
getMatrix ()
link2 : Link
linkM2 : LinkModel
notify ()
update ()
getMatrix ()
link3 : Link
linkM3 : LinkModel
notify ()
update ()
getMatrix ()
link4 : Link
linkM4 : LinkModel
notify ()
update ()
getMatrix ()
link5 : Link
linkM5 : LinkModel
: RobotGeometry
notify ()
update ()
getMatrix ()
notify ()
update ()
notify ()

Figure IV.9

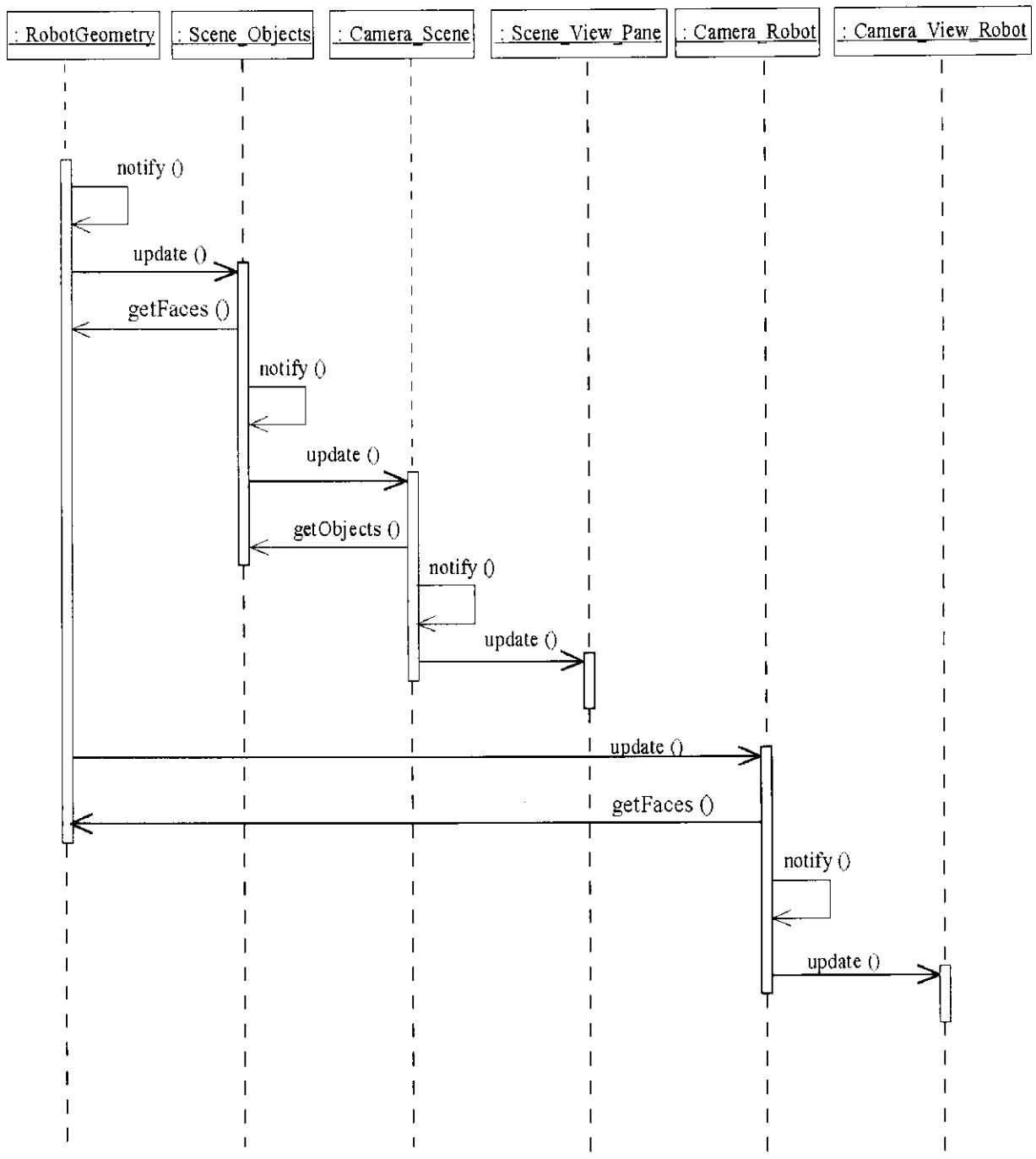


Figure IV.10

Dans les diagrammes de collaboration suivants nous allons détailler les interactions entre les différents liens (*Link*) du *Robot* et liens géométriques (*LinkModel*) du *Robot_Geometry* qui sont présentés dans les diagrammes de séquences précédents:

- Le premier diagramme (**Figure IV.11**) montre le flot d'informations échangées lors du changement de l'angle du premier lien (theta1).
- Le deuxième diagramme (**Figure IV.12**) montre le flot d'informations échangées lors du changement de l'angle du troisième lien (theta3).
- Le troisième diagramme (**Figure IV.13**) montre le flot d'informations échangées lors du changement de l'angle du sixième lien (theta6).

La remarque qu'on peut tirer de ces trois derniers, est que le flot d'information n'est pas le même dans le calcul de la cinématique des différents liens. Grâce au patterns *MVC* et *Observer* on optimise le calcul (on ne refait pas le calcul dès le début).

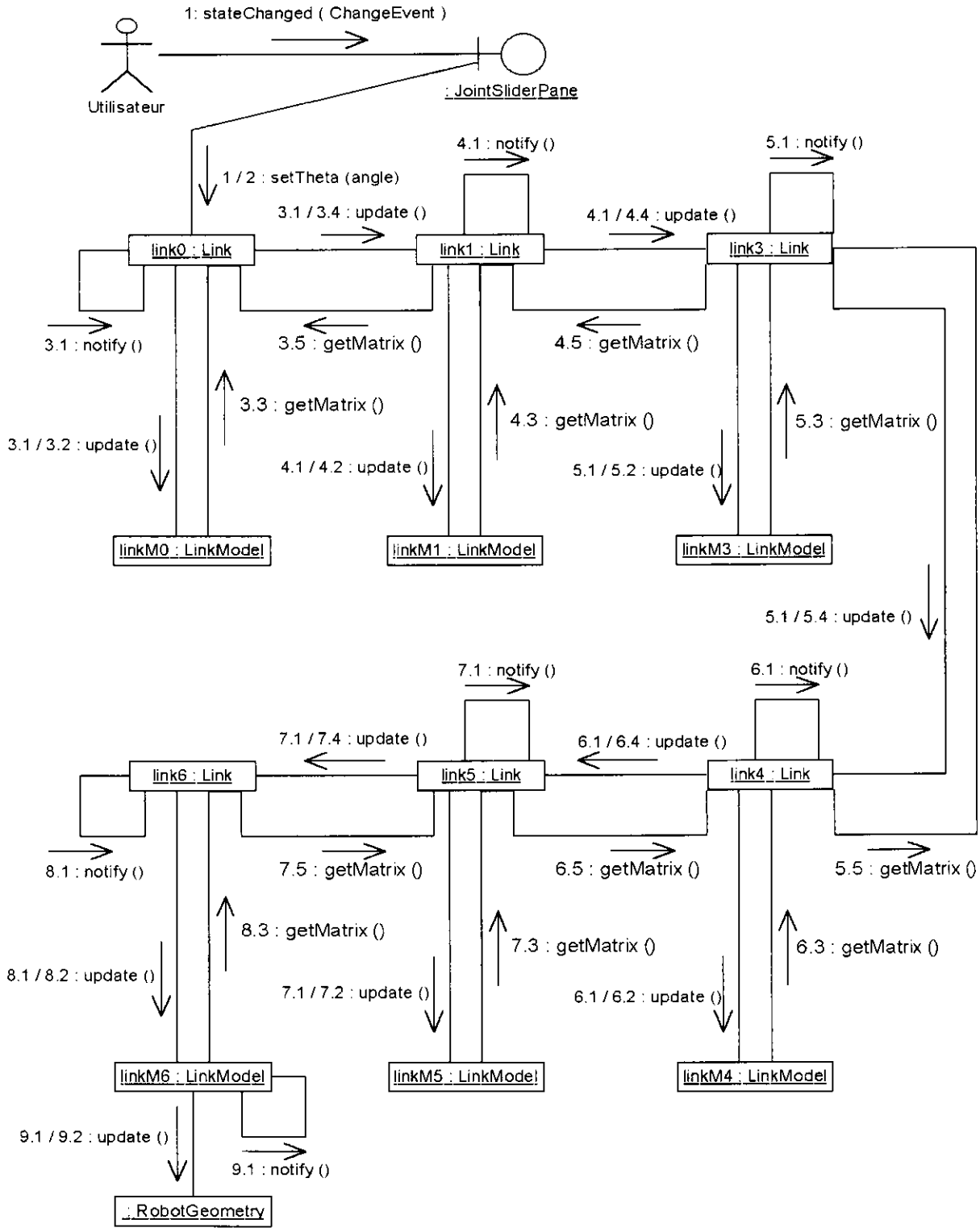

1: stateChanged (ChangeEvent)
Utilisateur
: JointSliderPane
1 / 2 : setTheta (angle)
4.1 : notify ()
5.1 : notify ()
3.1 / 3.4 : update ()
4.1 / 4.4 : update ()
link0 : Link
link1 : Link
link3 : Link
3.1 : notify ()
3.5 : getMatrix ()
4.5 : getMatrix ()
3.3 : getMatrix ()
4.3 : getMatrix ()
5.3 : getMatrix ()
3.1 / 3.2 : update ()
4.1 / 4.2 : update ()
5.1 / 5.2 : update ()
linkM0 : LinkModel
linkM1 : LinkModel
linkM3 : LinkModel
5.1 / 5.4 : update ()
7.1 : notify ()
6.1 : notify ()
7.1 / 7.4 : update ()
6.1 / 6.4 : update ()
link6 : Link
link5 : Link
link4 : Link
8.1 : notify ()
7.5 : getMatrix ()
6.5 : getMatrix ()
5.5 : getMatrix ()
8.3 : getMatrix ()
7.3 : getMatrix ()
6.3 : getMatrix ()
8.1 / 8.2 : update ()
7.1 / 7.2 : update ()
6.1 / 6.2 : update ()
linkM6 : LinkModel
linkM5 : LinkModel
linkM4 : LinkModel
9.1 / 9.2 : update ()
9.1 : notify ()
: RobotGeometry

Figure IV.11

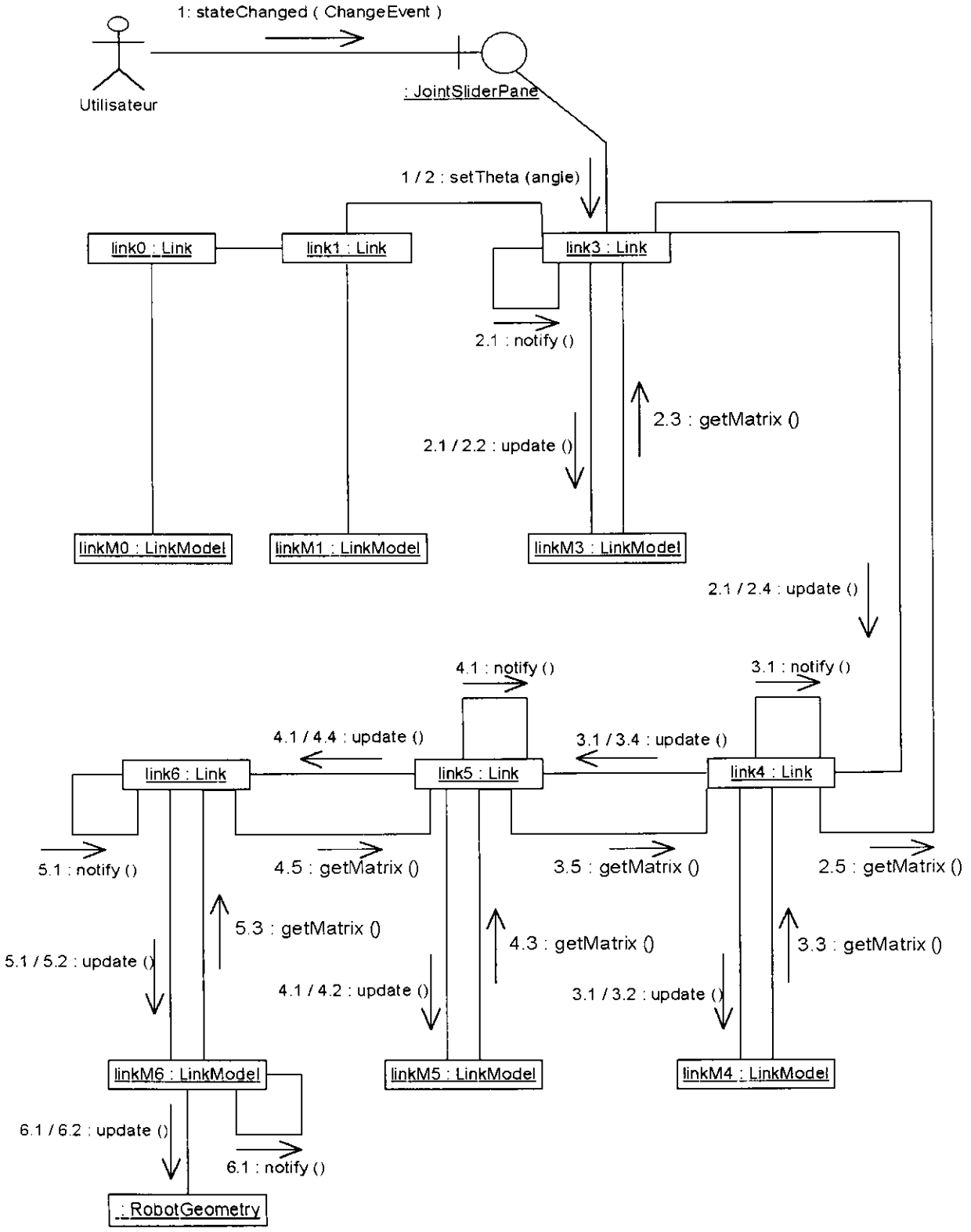

1: stateChanged (ChangeEvent)
Utilisateur
: JointSliderPane
1 / 2 : setTheta (angle)
link0 : Link
link1 : Link
link3 : Link
2.1 : notify ()
2.3 : getMatrix ()
2.1 / 2.2 : update ()
linkM0 : LinkModel
linkM1 : LinkModel
linkM3 : LinkModel
2.1 / 2.4 : update ()
4.1 : notify ()
3.1 : notify ()
4.1 / 4.4 : update ()
3.1 / 3.4 : update ()
link6 : Link
link5 : Link
link4 : Link
5.1 : notify ()
4.5 : getMatrix ()
3.5 : getMatrix ()
2.5 : getMatrix ()
5.3 : getMatrix ()
4.3 : getMatrix ()
3.3 : getMatrix ()
5.1 / 5.2 : update ()
4.1 / 4.2 : update ()
3.1 / 3.2 : update ()
linkM6 : LinkModel
linkM5 : LinkModel
linkM4 : LinkModel
6.1 / 6.2 : update ()
6.1 : notify ()
: RobotGeometry

Figure IV.12

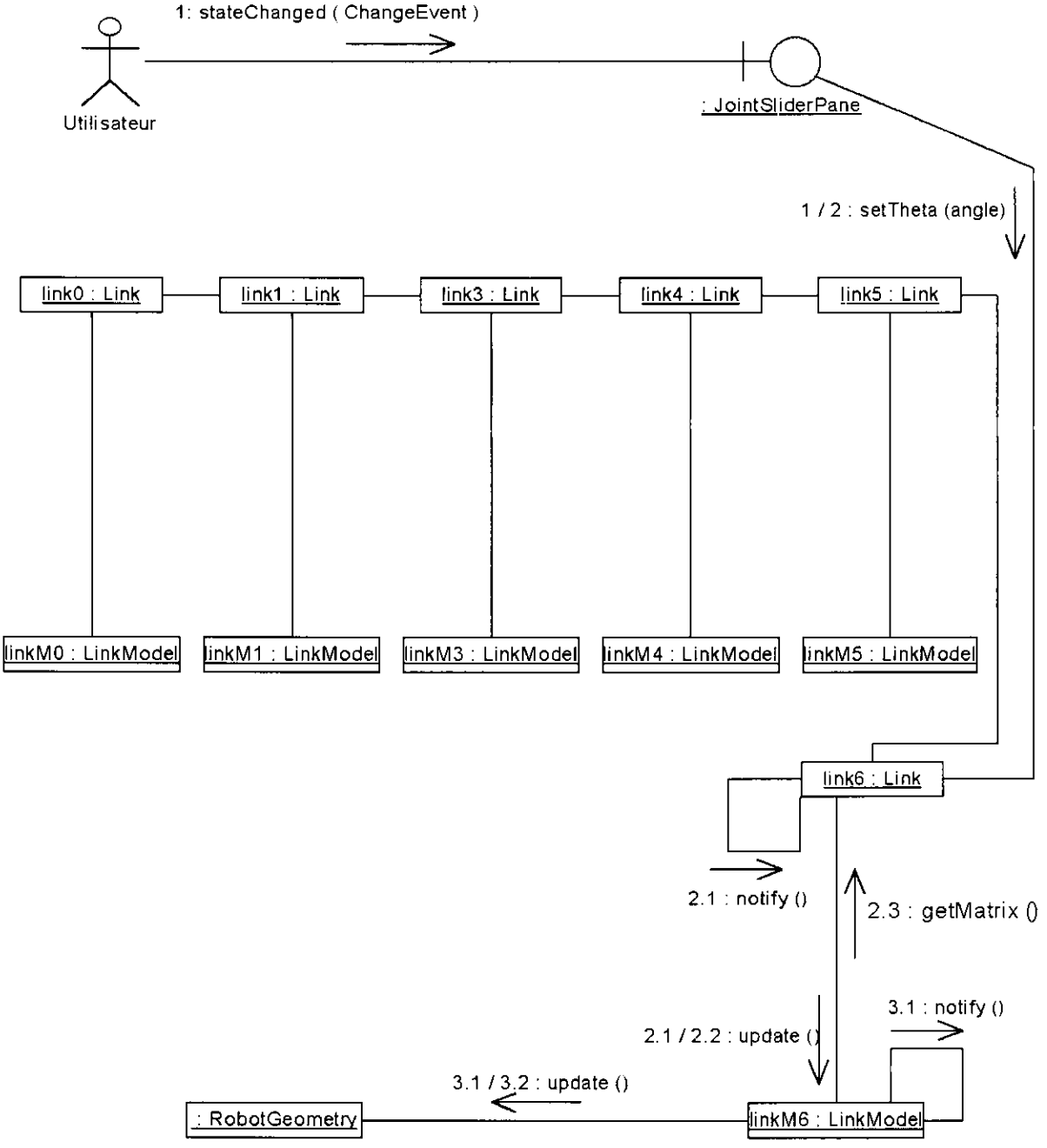

1: stateChanged (ChangeEvent)
Utilisateur
: JointSliderPane
1 / 2 : setTheta (angle)
link0 : Link
link1 : Link
link3 : Link
link4 : Link
link5 : Link
linkM0 : LinkModel
linkM1 : LinkModel
linkM3 : LinkModel
linkM4 : LinkModel
linkM5 : LinkModel
link6 : Link
2.1 : notify ()
2.3 : getMatrix ()
3.1 : notify ()
2.1 / 2.2 : update ()
3.1 / 3.2 : update ()
: RobotGeometry
linkM6 : LinkModel

Figure IV.13

- **Le déplacement de la caméra de la scène :**

L'utilisateur change la position de la caméra de la scène en glissant les *silders* associés dans l'interface utilisateur : *Camera_Scene_Pane*, ce changement va générer un évènement *stateChanged (ChangeEvent).* **(Figure IV.14)**

A la réponse de ce dernier, *Camera_Scene_Pane* envoie le message *setPosition(x,y,z)* à la caméra de la scène.

Cette dernière met à jours sa position et transforme la liste des faces de la géométrie du robot et la géométrie de la caméra puis elle exécute *notifyObservers()* pour notifier son observer (*Scene_View_Pane*) du changement, en envoyant le message *update().***(Figure IV.14)**

Enfin, *Scene_View_Pane* affiche les objets de la scène.

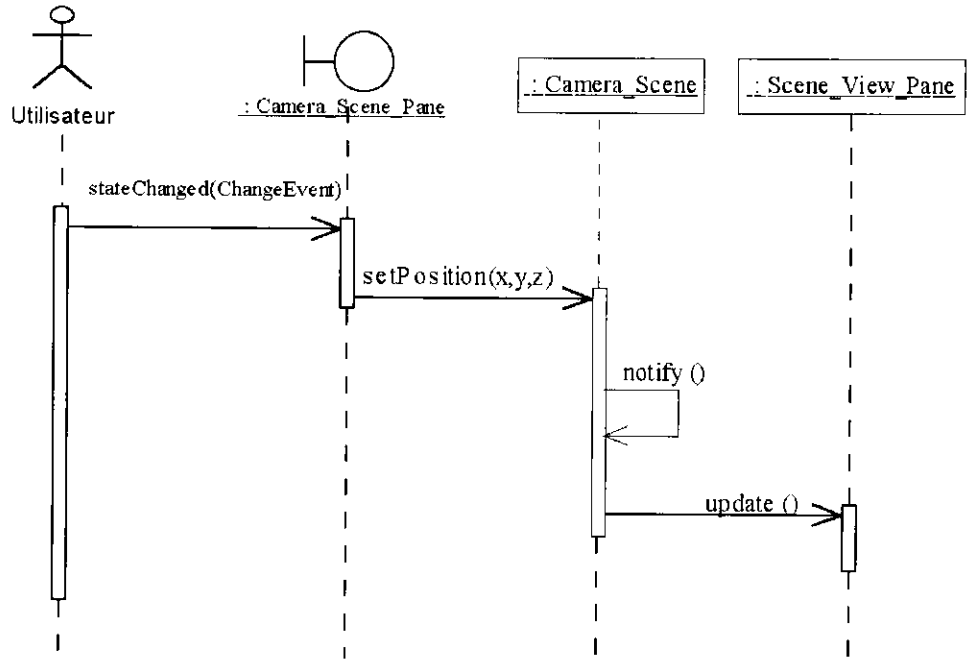


Figure IV.14

- **Le déplacement de la caméra du robot :**

L'utilisateur change la position de la caméra du robot en glissant les *silders* associés dans l'interface utilisateur : *Camera_Robot_Pane*, ce changement va générer un évènement *stateChanged (ChangeEvent).* **(Figure IV.15)** *et* **(Figure IV.16)**

A la réponse de ce dernier, *Camera_Robot_Pane* envoie le message *setPosition(x,y,z)* à la géométrie de la caméra.

Cette dernière met à jours sa position et transforme la liste de ses faces puis elle exécute *notifyObservers()* pour notifier ses observer (*Camera_Robot* et *Scene_Objects)* du changement, en envoyant le message *update().***(Figure IV.15)** et **(Figure IV.16)**

La *Scene_Objects* met à jours la liste de ses objets puis elle informe la *Camera_Scene* du changement.

Cette dernière récupère l'état de *Scene_Objects* en exécutant *getObjects*(), elle transforme la liste des faces des objets de la scène, puis elle l'envoie au panneau d'affichage de la scène : *Scene_View_Panel.* **(Figure IV.15)** et **(Figure IV.16)**

De même la *Camera_Robot* récupère l'état de la *Camera_Geometry* en exécutant *getPosition()*, elle transforme la liste des faces du *RobotGeometry*, puis elle l'envoie au panneau d'affichage du Robot : *Camera_View_Panel.* **(Figure IV.15)** et **(Figure IV.16)**

Enfin, *Camera_View_Pane* affiche le Robot.

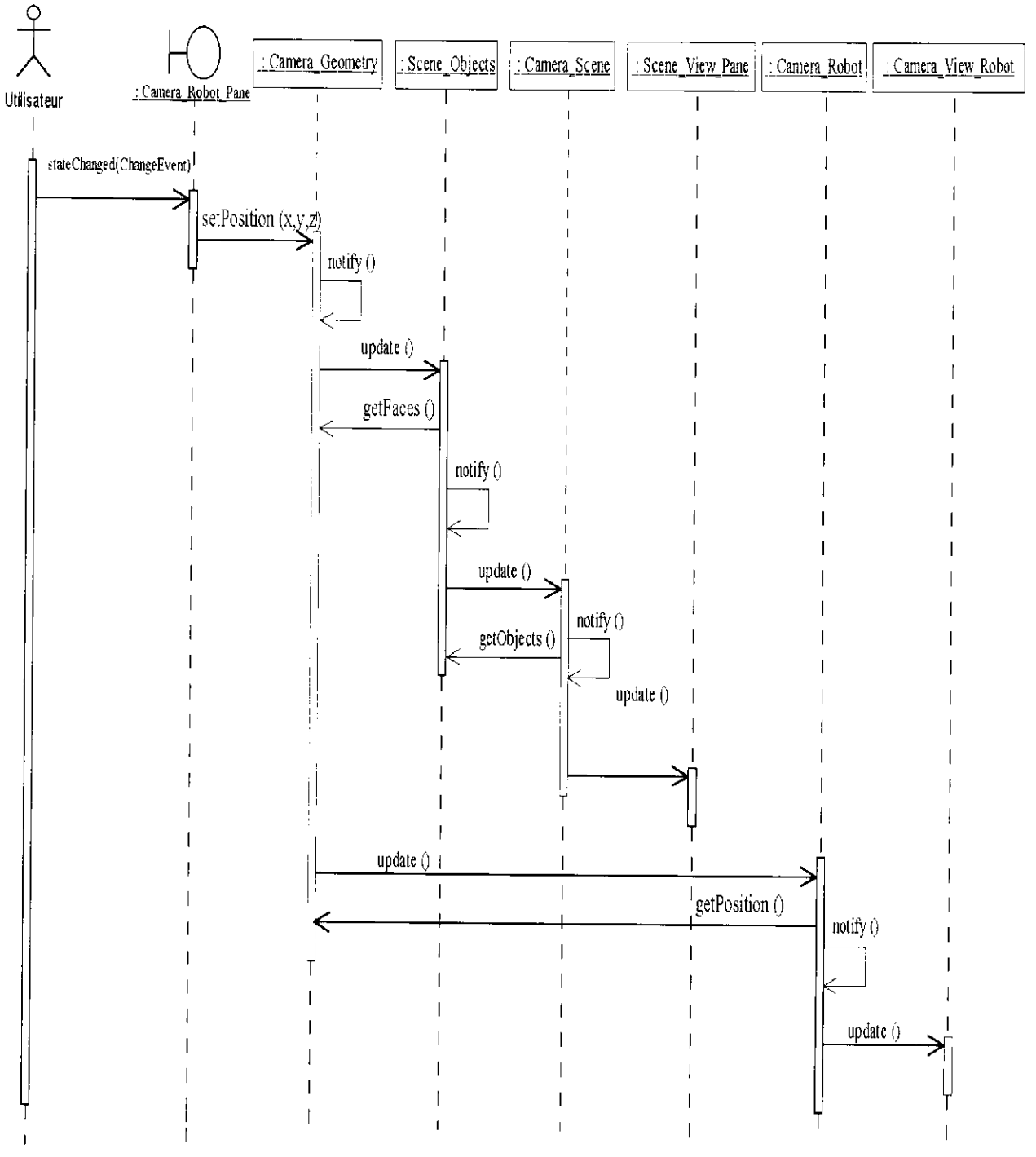

Utilisateur
: Camera_Robot_Pane
: Camera_Geometry
: Scene_Objects
: Camera_Scene
: Scene_View_Pane
: Camera_Robot
: Camera_View_Robot
stateChanged(ChangeEvent)
setPosition (x,y,z)
notify ()
update ()
getFaces ()
notify ()
update ()
notify ()
getObjects ()
update ()
update ()
getPosition ()
notify ()
update ()

Figure IV.15

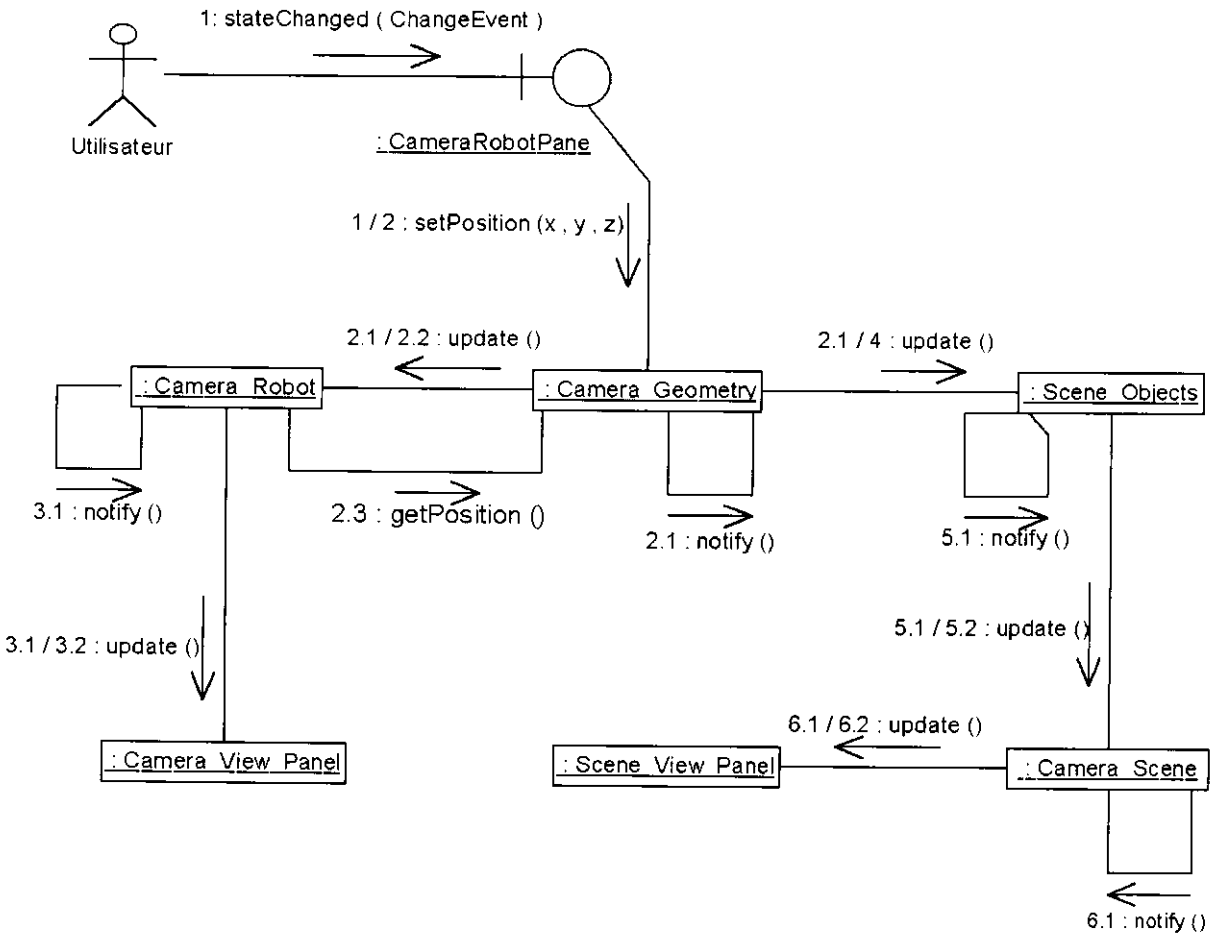


Figure IV.16

- **Le choix du modèle :**

L'utilisateur choisit un model de Robot puis il valide son choix en cliquant sur le bouton associé dans l'interface utilisateur : *ADM_Models_Frame*, ce changement va générer un évènement *actionPerformed(ActionEvent).* **(Figure IV.17)**

A la réponse de ce dernier, *ADM_Models_Frame* envoie le message *getConnection()* à *DataBase.*

Cette dernière crée une connexion avec la base de données puis exécute les requêtes SQL nécessaires.

Après la récupération des données, elle les envoie à *ADM_Models_Frame* qui les envoie à son tour à *RobotGeometry* pour mettre à jours sa structure[1].

- ***La gestion de la base de données :***

L'administrateur effectuent les mises à jours de la base de données (*Ajout, Modification, Suppression*) puis il valide son choix en cliquant sur le bouton associé dans l'interface utilisateur *(UI)*: *ADM_Tabbed_Frame*, ce changement va générer un évènement *actionPerformed(ActionEvent)*. **(Figure IV.18)**

A la réponse de ce dernier, *ADM_Models_Frame* envoie le message *getConnection()* à *DataBase*.

Cette dernière crée une connexion avec la base de données puis exécute les requêtes SQL nécessaires.

[1] Cette mise à jours déclenche le scénario qui est expliqué dans la **Figure IV.10**.

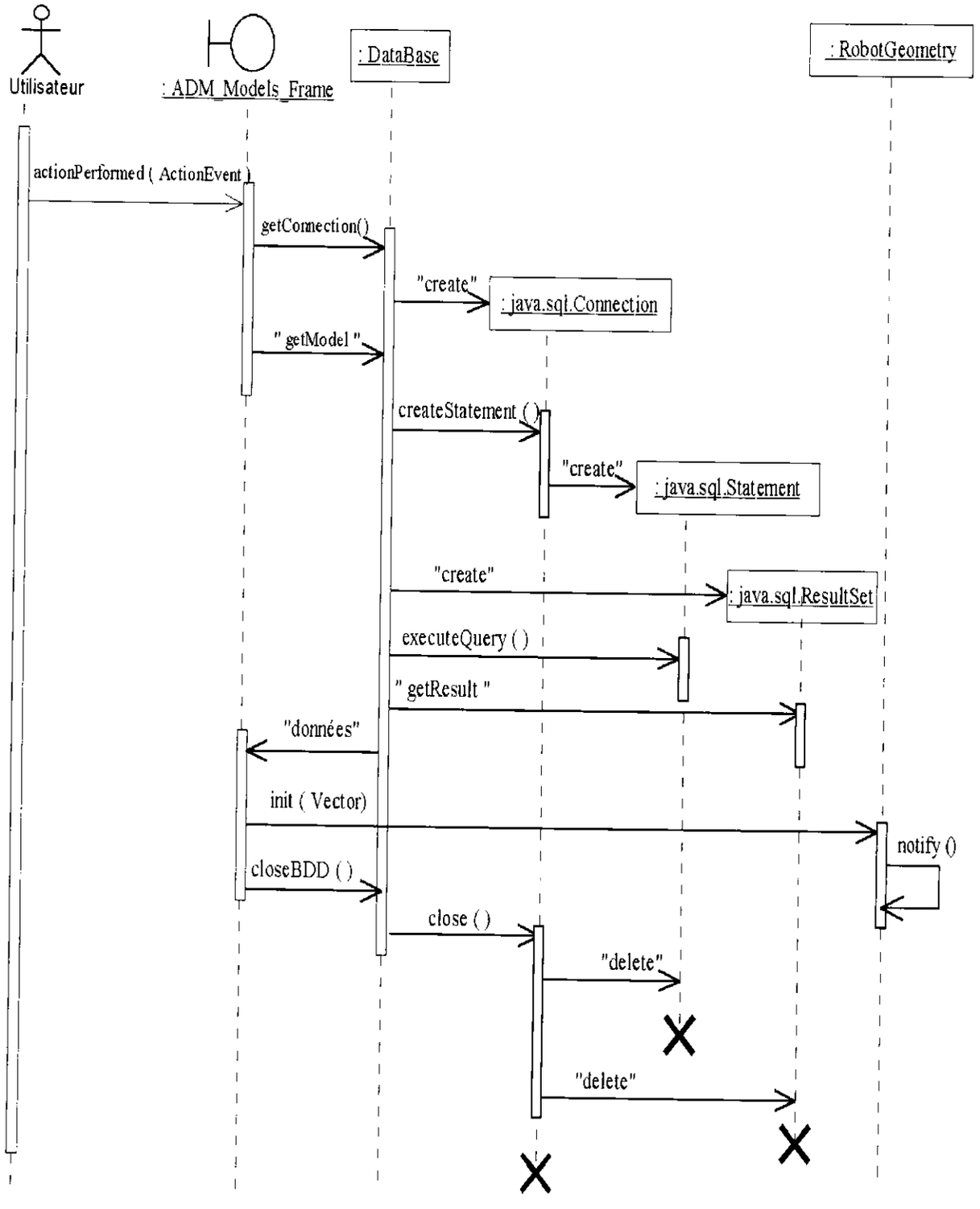

Utilisateur
: ADM_Models_Frame
: DataBase
: RobotGeometry
actionPerformed (ActionEvent)
getConnection()
"create"
: java.sql.Connection
" getModel "
createStatement ( )
"create"
: java.sql.Statement
"create"
: java.sql.ResultSet
executeQuery ( )
" getResult "
"données"
init (Vector)
notify ()
closeBDD ( )
close ( )
"delete"
"delete"

Figure IV.17

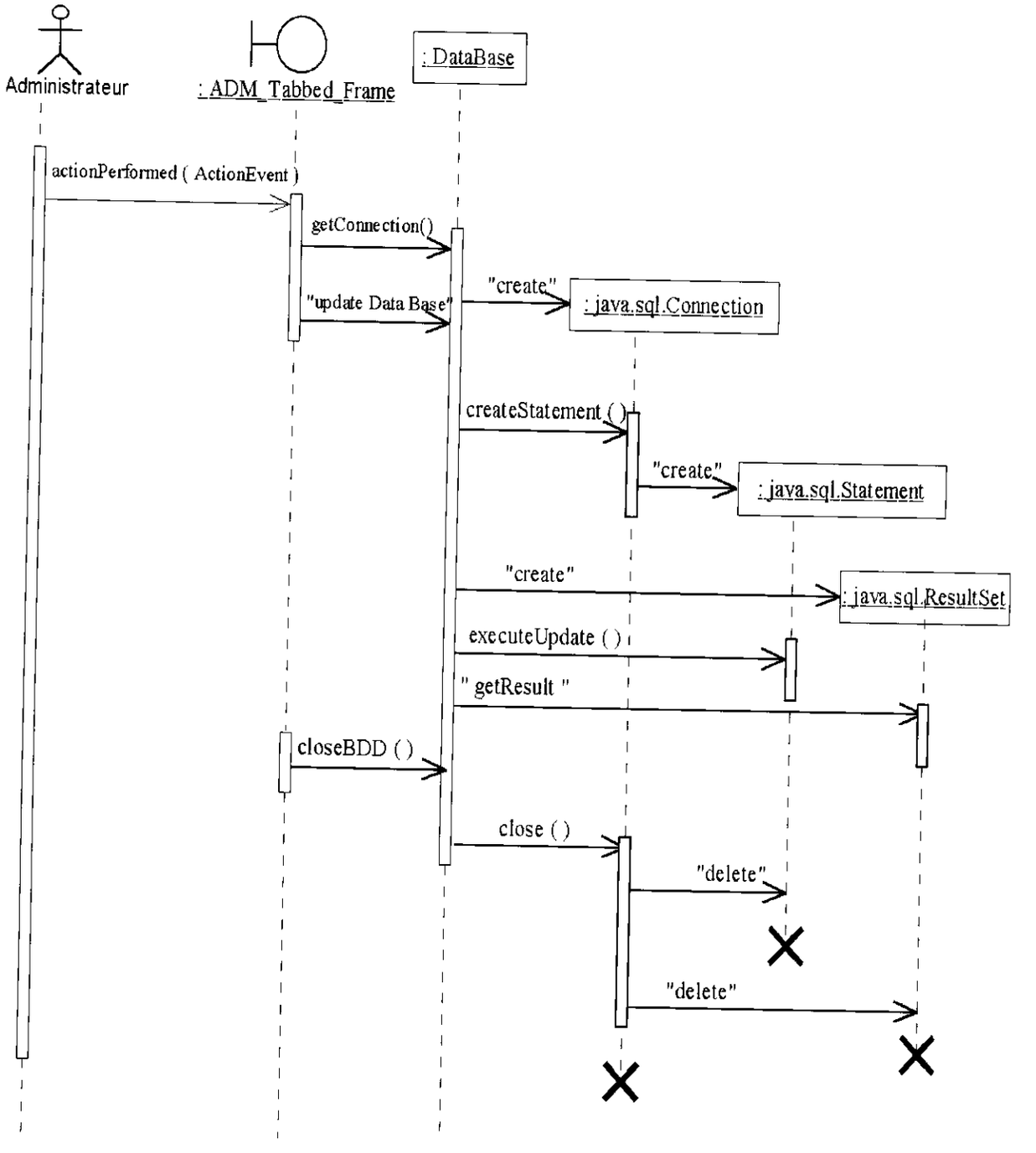

Administrateur
: ADM_Tabbed_Frame
: DataBase
actionPerformed (ActionEvent)
getConnection()
"update Data Base"
"create"
: java.sql.Connection
createStatement ( )
"create"
: java.sql.Statement
"create"
: java.sql.ResultSet
executeUpdate ( )
" getResult "
closeBDD ( )
close ( )
"delete"
"delete"

Figure IV.18

CHAPITRE V

TESTS

V-1. INTRODUCTION

Les tests dans une méthode XP sont effectués à la fin de chaque itération; lors du développement de notre système nous sommes passés par plusieurs étapes, chacune d'elles nous a donné un résultat qu'on a évalué selon nos besoins qui sont définis à chaque étape. Aussi nous avons fait des choix d'implémentation selon des tests de performances.

Dans ce chapitre nous allons exposer un ensemble de résultats obtenus pendant le développement de notre système.

V-2. TESTS INTERMEDIAIRES

Dans cette première étape, notre but (besoin) était la visualisation des objets à trois dimensions (3D) sur l'écran, pour cela la première composante conçue était la caméra. La **Figure V.1** montre la première version de camera affichant une vue d'un modèle graphique (cube multicolore).

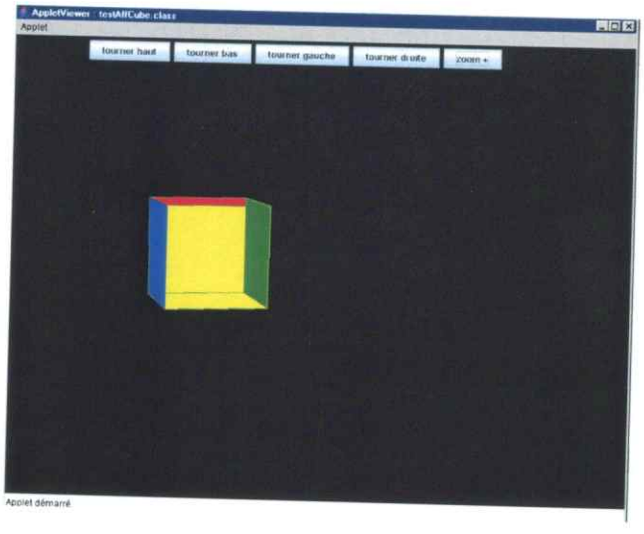


Figure V.1

Dans cette deuxième étape, notre but (besoin) était la résolution de la cinématique directe d'un bras manipulateur, pour cela la deuxième composante

conçue représente le robot (avec une architecture basée sur Observer et MVC) et la troisième composante est l'outil permettant la résolution de la cinématique directe appliqué à ce robot. La **Figure V.2** montre la première version du robot qui est représenté par une série de cubes, ce bras manipulateur (robot) peut être commandé (contrôlé) par les boutons qui font appel à la résolution de la cinématique directe.

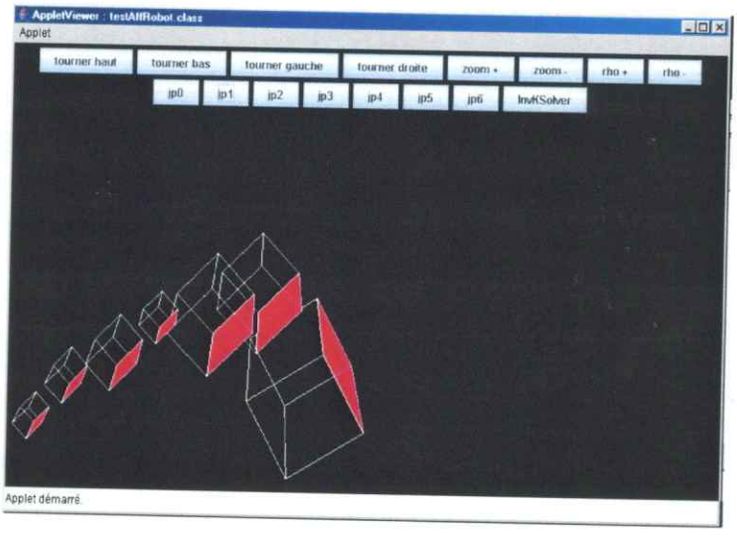


Figure V.2

Cette version consiste à transformer les objets de la scène en pixels dans le but de les afficher sur l'écran. La **Figure V.3** montre la première version du double buffer qui permet d'accéléré l'animation et d'utiliser une image d'arrière plan, il est basé sur le remplissage d'un tableau (buffer) contenant les couleurs des pixels.

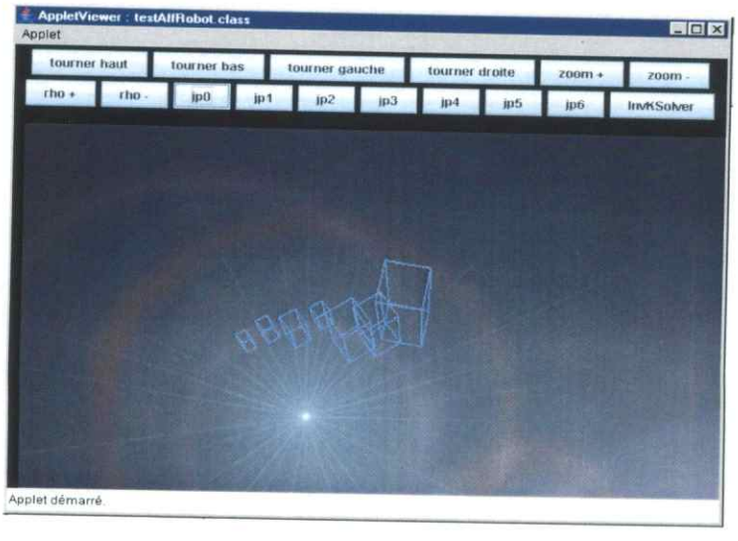


Figure V.3

La version présentée par la **Figure V.4** représente le modèle (tridimensionnel) final du robot qui est satisfaisant. On a utilisé un modèle surfacique en triangles, ces derniers sont remplis par une couleur transparente.

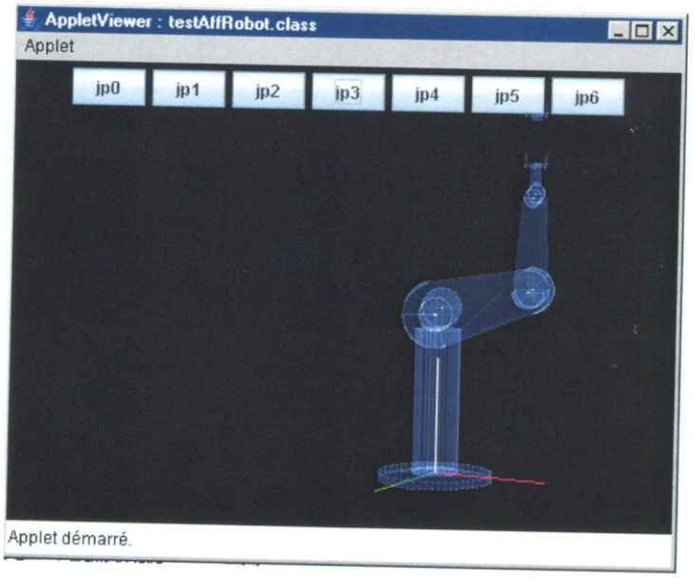


Figure V.4

La dernière version du système est présentée dans la **Figure V.5**. On a ajouté une camera de la scène pour la visualisation de tous les objets de la scène et une caméra du robot (celle qui est dans la scène et qui visualise juste le robot), les vues de ces deux caméra sont affichées dans deux panneaux d'affichage différent. On a ajouté aussi des panneaux de contrôles pour faire déplacer les deux caméras, pour changer les angles d'articulations du robot et pour changer l'apparence du robot (affichage en couleurs transparentes ou pas, affichage du plan sur lequel le robot est fixé et l'affichage du système de coordonnées du robot).

Cette version correspond aux besoins architecturaux (utilisation des patrons de conception et de l'orienté objet) mais le module de modélisation graphique doit être complété (ajout de sources lumineuses, et l'élimination des faces cachées).

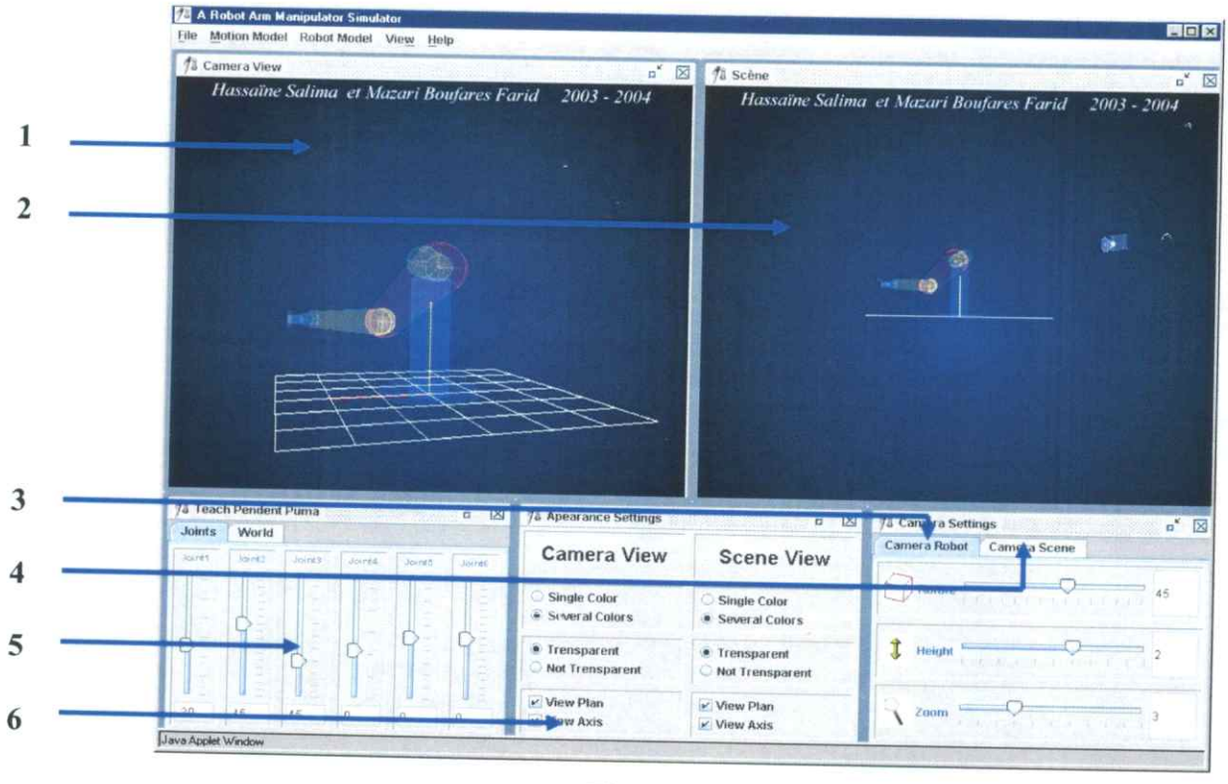


Figure V.5

1 : Le panneau d'affichage de la vue de la caméra du robot.

2 : Le panneau d'affichage de la vue de la caméra de la scène.

3 : Le panneau de contrôle de la caméra du robot.

4 : Le panneau de contrôle de la caméra de la scène.

5 : Le panneau de contrôle du robot.

6 : Le panneau de contrôle de l'apparence des objets affichés dans les deux panneaux.

V-3. TESTS DE PERFORMANCES

Pour choisir la meilleure implémentation nous avons fait des tests de performances dans un environnement de développement java 1.5 (Sun).

Les tests ont été effectués sur une machine **Intel** Pintium4 ® d'une fréquence de 1.7 MHz et d'une mémoire vive (RAM) DDR de 128 Mo. Le système d'exploitation tournant pendant les tests est : Microsoft Windows XP ® professionnel.

Le calcul du temps d'exécution a été effectué en utilisant l'API java : « System.currentTimeMillis() », qui retourne le temps courant en millisecondes. Le calcule du temps d'exécution s'effectue comme suit :

Temps d'exécution = Temps de début d'exécution – Temps de fin d'exécution

Au cours du développement de la simulation plusieurs choix se sont présentés :

- Choisir l'implémentation des matrices de transformations : notre système effectue en permanence une multitude de multiplication matricielles, il faut donc choisir une structure de données qui optimise le temps d'exécution.

Matrices implémentées par variables :

```
class Matrix4x4{
      private double m00,m01,m02,m03;
      private double m10,m11,m12,m13;
      private double m20,m21,m02,m23;
      private double m30,m31,m32,m33;
      //...
}
```

Matrices à base de tableaux :

```
Class Matrix4x4{
      private double m[][];
      //...
}
```

On remarque dans la **Figure V.6** que les multiplications de matrices tableau ont un temps d'exécution qui augmente de manière rapide par rapport au temps d'exécution de multiplication de matrices par variables. On à alors utilisé une implémentation de matrices par variables.

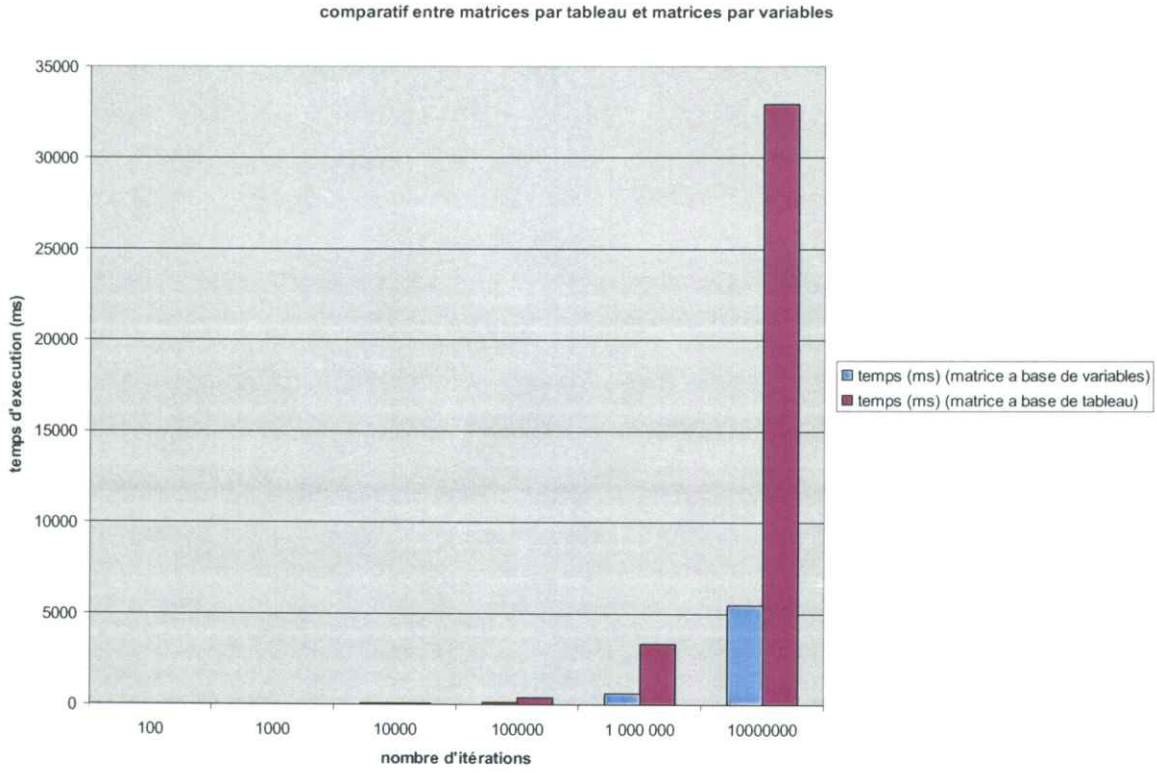


Figure V.6

- le deuxième choix porte sur les algorithmes de tris; doit on utiliser un tris simple ou un tris rapide dans le traitement des faces cachées ?

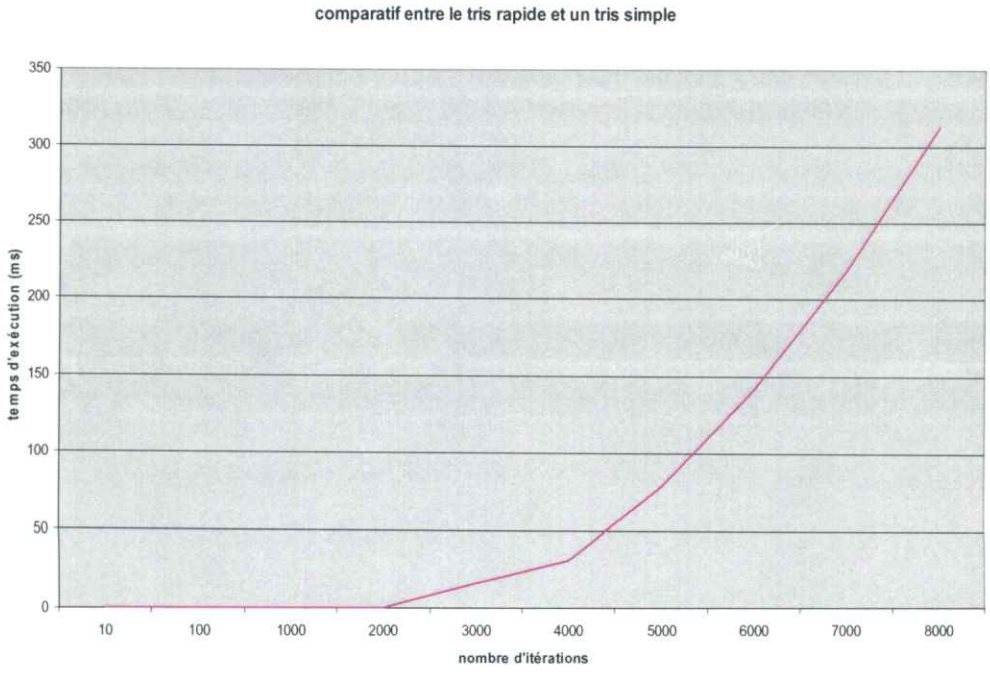


Figure V.7

Le temps d'exécution du tri rapide est de l'ordre des nanosecondes, sa courbe normalement en bleu n'est pas visible; le temps d'exécution dans tous les cas n'atteint pas la milliseconde. Alors que la courbe d'un tri simple monte en flèche. On opté par conséquent pour une solution d'élimination des faces cachées utilisant un tri rapide.

V-4. CONCLUSION

On a suivi une méthode itérative et incrémentale (XP), qui est basée sur les tests incrémentaux ; on peut effectuer plusieurs itérations sur le même composant logiciel, donc on peut avoir plusieurs versions jusqu'à arriver à la version finale de ce composant, pour valider une version on l'a teste ; lorsqu'on obtient des résultats satisfaisants on passe à un autre composant logiciel qui peut être réaliser après un certains nombre d'itérations et ainsi de suite jusqu'à arriver à la version complète et finale de notre système.

Cette manière de procédé nous a permis d'éviter les problèmes d'intégration de l'ensemble des composants en phase finale qui sont : le risque de la génération des erreurs importantes, des problèmes d'optimisations, le risque d'impossibilité d'assembler correctement le logiciel, des problèmes du fonctionnement de l'ensemble des composants logiciels,...etc.

CONCLUSION

Dans notre travail on a utilisé les concepts de l'Orienté Objet et les patrons de conception « patterns » en utilisant une méthode de développement agile. On garantit par l'utilisation de ces principes la réduction de la rigidité, de la fragilité, de l'immobilité et de la viscosité de notre système.

Les patrons de conception, précisément Observer, nous ont permis de réduire les calculs des matrices de transformations (de chaque lien du robot). Le patron de conception *Observer* nous a permis de séparer les différents composants du système (le couplage abstrait entre les différents modules nous facilite la maintenance et la réutilisation de ces composants logiciels); le modèle géométrique, modèle cinématique du robot, les camera, la scène et l'interface graphique sont tous séparés.

Dans ce travail on est arrivé à utiliser *Observer* en dehors de son application standard (*MVC*); la conclusion est que le patron de conception *Observer* peut être utilisé pour la communication entre les différents modules du système.

L'utilisation de l'*Observer* fournis par java, nous a permis de comprendre sa structure et son fonctionnement, on a pus donc implémenter notre propre patron de conception *Observer* qui permet de changer la structure de donnée interne de la classe *Observable* (liste d'observateurs «Observers », tableau d'observateurs ...etc.) et qui permet à un observable de notifié un observateur bien précis, ce qui n'est pas possible avec le patron de conception *Observer* de java

L'utilisation du patron de conception *Strategy,* nous a permis de tester différentes implémentations des différentes interfaces (*Camera_Interface, Sort_Interface, Collection, Matrix_Interface, Node_Interface,...*).

La structure en liste chaînée, du modèle cinématique du robot, nous a permis de communiquer facilement entre les différents liens lors du changement des matrices de transformations.

Le langage java nous a offert un ensemble d'API pour le développement d'interfaces graphiques « Swing ».

Reste à faire

L'objectif principal de notre travail est atteint; on a pus appliquer les concepts de l'Orienté Objet et les patrons de conception « patterns » en travaillant selon les principes de la méthode XP. Néanmoins, il reste à introduire certaines fonctionnalités dans la simulation :

- Nous avons conçu une base de donnée afin de charger la géométrie et les propriétés du robot, il reste à intégrer ce composant dans le logiciel.
- Implémenter le traitement des faces cachées.
- Implémenter un module d'illumination.
- Concevoir un *solver* de cinématique inverse.

On n'a pas pus terminer les caractéristiques et les modules cités ci-dessus faute de temps.

Extensions possibles

On peut imaginer plusieurs extensions de notre travail :

- Evitements d'obstacles ;
- Intégrer de nouveaux objets dans la scène, que le bras pourrait manipuler ;
- Introduire la dynamique des manipulateurs dans le système ;
- Ajouter un module pour la communication avec le matériel « Hard » qui permet de commander un vrai bras de robot ;
- Utiliser des techniques plus élaborées dans le rendu et l'animation du robot ;
- Utiliser d'autres patterns dans le développement de notre système (qui peuvent avoir une relation avec les patterns déjà utilisés),...etc.

ANNEXE A

DÉVELOPPEMENT ORIENTÉ OBJETS

1. ORIENTÉ OBJETS :

L'orienté objet consiste à modéliser informatiquement un ensemble d'éléments d'une partie du monde réel (que l'on appelle domaine) en un ensemble d'entités informatiques. Ces entités informatiques sont appelées objet. Il s'agit de données regroupant les principales caractéristiques des éléments du monde réel (taille, couleur, comportement, ...).

1.1 Objet

Un objet est une entité qui comporte des données et des méthodes. Les données représentent l'état de l'objet; et les méthodes représentent le comportement qu'il peut adopter. L'état d'un objet peut être uniquement modifié par l'envoi de message à l'objet, ce qui déclenche l'exécution d'une méthode.

1.2 Classe

Une classe permet de regrouper des objets ayant des caractéristiques semblables. La classe n'est qu'une représentation abstraite d'un objet. Dans un programme, l'objet est créé à partir de cette classe, on parle alors d'instance.

Une classe est composée de deux parties :

- **Les attributs** (parfois appelés données membres) : il s'agit des données représentant l'état de l'objet
- **Les méthodes** (parfois appelées fonctions membres) : il s'agit des opérations applicables aux objets

1.3 Encapsulation

L'encapsulation est un mécanisme consistant à rassembler les données et les méthodes au sein d'une structure en cachant l'implémentation de l'objet, c'est-à-dire en empêchant l'accès aux données par un autre moyen que les services proposés. L'encapsulation permet donc de garantir l'intégrité des données contenues dans l'objet.

Le masquage des informations

L'utilisateur d'une classe n'a pas forcément à connaître la façon avec laquelle sont structurées les données dans l'objet, cela signifie qu'un utilisateur n'a pas à connaître

l'implémentation. Ainsi, en interdisant à utilisateur de modifier directement les attributs, et en l'obligeant à utiliser les fonctions définies pour les modifier (appelées interface), on est capable d'assurer l'intégrité des données.

L'encapsulation permet de définir des niveaux de visibilité des éléments de la classe. Ces niveaux de visibilité définissent les droits d'accès aux données. Il existe trois niveaux de visibilité :

- **publique :** Les fonctions de toutes les classes peuvent accéder aux données ou aux méthodes d'une classe définie avec le niveau de visibilité public. Il s'agit du plus bas niveau de protection des données.
- **protégée :** l'accès aux données est réservé aux fonctions des classes héritières, c'est-à-dire par les fonctions membres de la classe ainsi que des classes dérivées.
- **privée :** l'accès aux données est limité aux méthodes de la classe elle-même. Il s'agit du niveau de protection des données le plus élevé.

1.4 Héritage

Cette notion permet de décrire des objets de plus en plus précis, héritant des caractéristiques d'objets plus généraux. Elle est très puissante, permet la réalisation de logiciels extensibles, et surtout favorise la réutilisation de modules déjà créés. L'héritage (*inheritance*) est un principe propre à la programmation orientée objet, permettant de créer une nouvelle classe à partir d'une classe existante. Le nom d'"héritage" provient du fait que la classe dérivée contient les attributs et les méthodes de sa superclasse (la classe dont elle dérive). L'intérêt majeur de l'héritage est de pouvoir définir de nouveaux attributs et de nouvelles méthodes pour la classe dérivée, qui viennent s'ajouter à ceux et celles héritées. Par ce moyen on crée une hiérarchie de classes de plus en plus spécialisées. Cela a comme avantage majeur de ne pas avoir à repartir de zéro lorsque l'on veut spécialiser une classe existante.

1.5 Polymorphisme

Le nom de polymorphisme vient du grec et signifie : qui peut prendre plusieurs formes. Cette caractéristique essentielle de la programmation orientée objet caractérise la possibilité de définir plusieurs fonctions de même nom mais possédant des paramètres différents (en nombre et/ou en type), si bien que la bonne fonction sera choisie en fonction de

ses paramètres lors de l'appel. Le polymorphisme rend possible le choix automatique de la bonne méthode à adopter en fonction du type de donnée passée en paramètre.

2. PROBLÈMES LIÉS AU DÉVELOPPEMENT :

2.1. Architecture et dépendances :

Certaines applications ont tendance à maintenir la pureté du design (conception) à travers le développement initial, dès la première version. Un risque de pourrissement du logiciel est alors envisageable depuis le début. En effet pour éviter la dégradation de la conception on a tendance à rajouter du code qui devient difficilement maintenable ; Les dépendances entre modules et classes s'accumulent et deviennent difficilement gérables.

2.2. Symptômes de pourrissement d'un système :

Les quatre (4) symptômes suivants représentent des indicateurs de l'état de dégradation d'un système :

2.2.1. Rigidité : Un logiciel est dit rigide si il est difficile d'effectuer un changement. Chaque modification implique une cascade de changements dans des modules dépendants, cela prend de l'ampleur avec un manque de communication, des bugs non critiques sont négligés en raison de la pression des managers sur les développeurs et d'une connaissance moindre du délai de développement.

2.2.2. Fragilité : Quant un logiciel présente des cassures (bugs et problèmes de dépendances) et dans différents endroits, après un changement, on dit qu'il est ***fragile***. Souvent les cassures se produisent dans des zones qui n'ont aucune relation conceptuelle avec les zones ayant changés.

2.2.3. Immobilité : L'incapacité de réutiliser du software issu du même projet ou d'un autre rend le développement du système immobile.

2.2.4. Viscosité : La viscosité se présente sous deux formes

- Viscosité du design,
- Viscosité de l'environnement.

En face d'un problème les ingénieurs trouvent souvent plusieurs solutions pour résoudre un problème, certaines préservent le design (correspondent a la conception initiale) certaines non. Quand un ingénieur constate que, parmi les solutions trouvées, celle qui ne préserve pas le design est la plus facile, il risque de l'utiliser donc d'augmenter la viscosité du système.

La viscosité de l'environnement est due à la lenteur et l'inefficacité de l'environnement de développement.

En conclusion, ses quatre (4) symptômes reflètent une architecture pauvre, leur présence dans une application exhibe un problème dans la conception.

2.4. Les causes de la dégradation du développement :

2.4.1 Changement des besoins :

Le changement des besoins peut poser un problème lorsque la conception initiale ne peut pas anticiper. Souvent les changements doivent être rapides alors que les ingénieurs ne sont pas assez familiers de la méthode conceptuelle originelle.

2.4.2 Gestion des dépendances :

Le type de changements qui provoquent la dégradation du développement logiciel est celui qui introduit de nouvelles dépendances non planifiées (imprévues). Les 4 symptômes cités précédemment sont des résultats directs ou indirects de la présence de dépendances inappropriées entre les modules du software.

La dégradation de l'architecture des dépendances rend le logiciel difficile à maintenir ; pour résoudre un tel problème une gestion des dépendances est nécessaire, elle consiste à utiliser des firewalls anti-dépendances, leur réalisation est permise grâce aux techniques de ***l'orienté objet*** et des ***patrons de conception***.

3. PRINCIPES DU DESIGN ORIENTÉ OBJET :

3.1 L'OCP (Open-Closed Principle) :

Ce principe prône l'écriture des modules tel qu'ils puisent être étendus sans changement ; en d'autres termes : changer le comportement sans changer le code.

Le but principal de l'OCP est d'éviter la propagation des changements dans le code.

3.2 L'abstraction :

Les techniques qui tendent à atteindre l'OCP sont basées sur l'abstraction.

Les programmes qui se composent d'une multitude d'expressions similaires au « if/else » ou au « switch », lors de l'ajout d'un nouvel objet ou d'un changement de stratégie objet, toutes les expressions du type cité doivent être scannés et convenablement modifiées.

Cet exemple montre la structure d'un tel code :

```
if (Object.type = ObjectType1)
    Object.ObjectFunction1() ;
else
        if(Object.type = ObjectType2)
            Object.ObjectFunction2() ;
```

Les Structures de se type compliquent la maintenance et sont des sources d'erreur. Pour éviter de telles séquences on utilise les techniques d'abstraction, plusieurs types d'objets ont une même fonction ou méthode mais chacune effectue un traitement différent selon la classe d'appartenance de l'objet.

3.3 Polymorphisme dynamique :

Dans l'OCP le fonctionnement ne dépend que de l'interface. Des classes additionnelles ne provoqueront pas de changement dans le fonctionnement. Ainsi, on aura crée un module extensible, avec une nouvelle classe concrète, sans aucune modification du comportement.

3.4 Le principe de substitution de Liskov (LSP) :

La classe de base peut être substituée par ses sous classes. Les violations de se principe sont

difficiles a détecter a temps, si la conception est fortement présente dans le développement, le coût d'une réparation sera très important.

3.5 Le principe d'inversion de dépendances (DIP) :

Le principe est simple : dépendre de l'abstraction et on du concret ; dépendre des interfaces ou des fonctions et classes abstraites au lieu de dépendre des fonctions et classes concrètes.

3.6 Dépendre d'une Abstraction :

Chaque dépendance doit cibler une interface ou une classe abstraite.

3.7 Création D'objet :

Pour créer une instance, on doit dépendre d'une classe concrète. La création d'instances apparaît dans pratiquement toute l'architecture du design. Ainsi, il semble qu'aucune échappatoire a se problème n'est possible, le système comportera une multitude de ces dépendances. Néanmoins, le patron « *Abstract Factory* » est une solution élégante pour se type de problèmes.

3.8 L'ISP (Interface Segregation Principle) :

Plusieurs interfaces client au lieu d'une interface a but général. Si une classe a plusieurs clients, plutôt que de charger une seule classe avec toutes les méthodes dont le client a besoin, il est préférable de créer une interface spécifique pour chaque client.

Les clients (classes qui utilisent un service) doivent être classées selon leur type, on crée alors pour chaque type une interface. Si au moins deux types de clients ont besoins d'une même méthode, cette dernière doit être ajoutée à leurs interfaces.

ANNEXE B

LES PATRONS DE CONCEPTION « DESIGN PATTERNS »

DÉFINITION DES PATRONS DE CONCEPTION

En prenant un dictionnaire comme référence, le mot « *patron* » tel qu'il est traduit de l'anglais : « *pattern* », veut dire modèle d'un ouvrage, d'un objet ...etc.

Les patrons de conception *(Design Pattern)* ont pour objectif de conserver l'expérience d'experts. Ils représentent les connaissances d'experts en orienté objet dans une forme pouvant être distribuée aux autres développeurs. Ils le font en nommant, décrivant et validant les solutions qui sont le plus souvent reproduites.

Les patrons de conception sont des solutions éprouvées à des problèmes spécifiques et récurrents « *Un patron décrit un problème devant être résolu, une solution, et le contexte dans lequel cette solution est considérée. Il nomme une technique et décrit ses coûts et ses avantages. Il permet à une équipe d'utiliser un vocabulaire commun pour décrire leurs modèles* » **[8].**

A partir des différents points de vue traités dans les (§) précédents, on peut déduire que dans le domaine du génie logiciel Les patrons de conception identifient et nomment un problème récurrent proposent une solution élégante à ce problème et ils permettent aussi de caractériser la qualité de l'implantation et de la conception des programmes avec les motifs utilisés. Il faut noter que le terme " *Design Patterns* " est souvent utilisé. Cependant il ne faudrait pas en déduire que L'approche *pattern* intervient seulement dans la partie conception (*Design*), mais elle peut intervenir dans les différentes phases de cycle de vie d'un logiciel.
On distingue : **[9], [10]**.

- **les *patterns d'analyse :*** Ils ont pour but de guider les étapes d'analyse d'un cycle de vie d'un logiciel. Ils permettent d'identifier les problèmes récurrents dans l'expression des besoins des applications et de transformer ces expressions en des modèles réutilisables.

- **les *patterns architecturaux :*** Ce sont des schémas d'organisation structurelle fondamentaux pour les logiciels.

- **les *patterns de conception* (*design patterns*) :** Ils sont reconnus comme des bonnes techniques du génie logiciel à objets. Cette technique améliore le cycle de vie du logiciel en facilitant la conception, la documentation, la maintenance .Ils sont considérés comme des micro-architectures qui visent à réduire la complexité, à promouvoir la réutilisation et à fournir un vocabulaire commun aux concepteurs.

- **les *frameworks (cadres d'applications) :*** Ce sont des sous systèmes prêt à être instancier avec des possibilités bien définies d'adaptation et d'extension.

- **les *idiomes* (*patterns d'implantation*) :** Ce sont des patterns de bas niveau dans un langage de programmation donné. Ils sont destinés à montrer comment réaliser, dans un langage donné, un trait absent de ce langage. Par exemple comment réaliser l'héritage multiple en Java, comment représenter des évolutions multiples d'objets...

En général, chaque pattern est caractérisé par quatre éléments essentiels : un ***nom***, un ***problème***, une ***solution*** et des ***conséquences***. **[11]**

- ***Le nom*** offre la possibilité de communiquer des concepts plus abstraits entre concepteurs.
- ***Le problème*** décrit les situations dans lesquelles le patron devrait être utilisé.
- ***La solution*** décrit les éléments constituants la solution (classes) et leurs rôles, relations, responsabilités et coopération.
- **Les *conséquences*** décrivent les implications de l'utilisation du pattern. Elles traitent souvent des principaux facteurs de qualité du logiciel, entre autres les détails d'implantation, les possibilités d'extension, la réutilisabilité et la portabilité.

1. DESCRIPTION ET DOCUMENTATION DES PATTERNS

Gamma et al ont adopté une représentation uniforme et structurée. Chaque pattern est décrit et documenté de la même façon selon les propriétés ci-dessous :

- **Nom :** le nom significatif du pattern.
- **Intention** : décrit ce que fait le pattern, son but et quel problème particulier il résout.

- **Alias :** énumère, s'il en existe, d'autres noms communs du pattern.
- **Motivation :** donne un exemple de problème de conception pouvant être résolu par application du pattern. L'illustration par un exemple facilite la compréhension de la description abstraite (donnée par la suite) du pattern.
- **Indications d'utilisation :** décrit les situations où on peut utiliser le pattern.
- **Structure :** décrit la structure du pattern en utilisant des diagrammes de classes. Des diagrammes de collaboration sont aussi utilisés pour la description de l'interaction entre les classes du pattern.
- **Participants :** définit les classes (ou objets) intervenantes dans le pattern et leurs responsabilités.
- **Collaborations :** décrit comment les participants interagissent pour assumer leurs responsabilités.
- **Conséquences :** décrit comment le pattern atteint ses objectifs, quels sont les compromis et les résultats de son utilisation.
- **Implémentation :** présente des astuces, techniques et pièges qu'il faut connaître lors de l'implémentation du pattern. Cette partie propose aussi, quand il en existe, des solutions typiques du langage utilisé.
- **Exemple de code :** donne des fragments de code pour illustrer la façon dont on peut implémenter le pattern dans un langage orienté objet.
- **Utilisations connues :** contient des exemples d'utilisation du pattern dans des systèmes réels.
- **Patrons apparentés :** cite les patterns qui sont reliés avec celui en cours de traitement et leurs différences.

2. LES PROBLÈMES D'UTILISATION ET D'APPLICATION DES PATRONS

Il est difficile de les apprendre les patrons de conception, certains d'entre eux sont complexes, ce qui rend leurs implémentation et utilisation encore plus complexes. **[12]**

Les patrons de conception ne sont pas tous au même niveau de complexité. En effet, l'application, de certains d'entre eux, n'est pas un processus évident. Il faut cerner le problème pour pouvoir identifier la solution adéquate et donc le patron adéquat. Une fois que le patron à utiliser est identifié, il faut l'appliquer au modèle concerné. Pour cela, le

concepteur doit pouvoir définir le rôle de chaque élément du modèle pour établir une correspondance avec les éléments du patron. **[11]**

Une fois le patron appliqué, il faudra vérifier si la sémantique du modèle est toujours respectée, comme il faudra s'assurer que de futures modifications ne violent pas les contraintes sémantiques et structurelles imposées par le patron. **[11]**

L'implémentation des patrons dans un langage de programmation nécessite une mise en correspondance entre les éléments de conception du patron et les constructions du langage de programmation. Cela n'est pas toujours aussi évident, dans le cas du langage Java, par exemple, on ne peut pas implémenter l'héritage multiple. **[11]**

Les patrons permettent d'améliorer la flexibilité dans une application en introduisant des niveaux d'adressage indirect supplémentaires. Il faut donc utiliser les patrons avec une certaine discrétion puisqu'ils peuvent entraîner une complexité inutile et une perte de performance injustifiée. **[13]**

3. CLASSIFICATION DES PATTERNS DE CONCEPTION

Gamma et al. **[14]** ont classé les patrons qu'ils proposent selon leur rôle et leur domaine d'application (classe vs objet). Ils ont distingué entre patrons Créateurs, Structurels et Comportementaux. (**Figure. B1**)

- ***Les patrons créateurs* (*Creational patterns*)** concernent la création de classes ou d'objets. On peut citer : Factory method, Abstract factory method, Builder, Singleton, Prototype,...

- ***Les patrons structurels* (*Structural patterns*)** s'intéressent à la composition d'objets ou classes pour réaliser de nouvelles fonctionnalités. On peut citer : Adapter, Bridge, Composite, Flyweight, Decorator, Façade, Proxy,...

- ***Les patrons comportementaux* (*Behavioral patterns*)** concernent les interactions entre classes et l'affection des responsabilités. On peut citer : Observer, State, Strategy, Mediator, Chain of Responsibility, Template, Interpreter, Visitor, Iterator, Memento,....

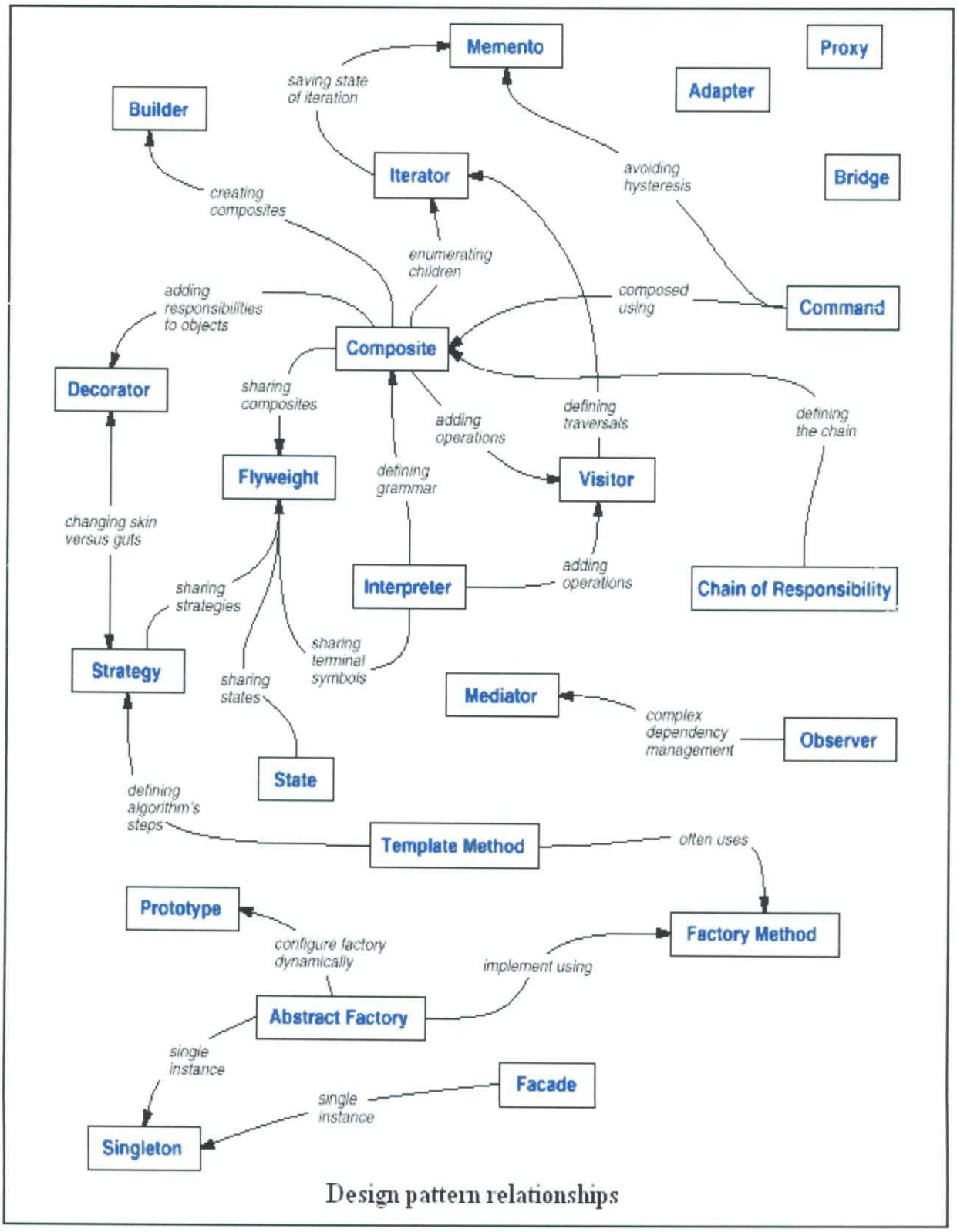


Figure B.1

4. LE PATTERN D'ARCHITECTURE *MVC (MODEL VIEW CONTROLLER)*

Parmi les patrons architecturaux, qui se rattachent à la problématique de la conception architecturale, on peut prendre en exemple le (*Model-View-Controller* ou *MVC*) a été créé en 1980 par A.Goldberg à Xerox PARC pour le langage Smalltalk-80. **[15]**. Son but est de faciliter la programmation dans le cas des systèmes ou on a besoin de présenter une même donnée de manière synchrone et multiple.

Le Modèle MVC est une architecture qui décompose un composant logiciel en trois parties séparées : les objets applicatifs (*Model*), les vues externes (*View*), et les contrôleurs (*Controller*) qui lient événements sur les vues et services sur les objets **(Figure B.2)**.

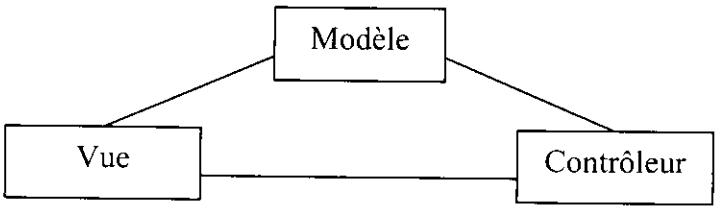


Figure B.2 : L'architecture MVC

- Le ***modèle (model)*** est la partie qui représente le comportement et l'état du composant logiciel.
- La ***Vue (View)*** est la partie qui permet la visualisation de l'état représentant le model.
- Le ***contrôleur (controller)*** est la partie qui gère l'interaction de l'utilisateur avec le model, il permet de changer l'état du model.

Il est important de noter que :

- Le model n'as aucune connaissance particulière sur ses vues et contrôleurs, c'est le système qui avertit les vues lors d'un changement d'état dans le model.
- Un model peut avoir plusieurs vues et plusieurs contrôleurs.

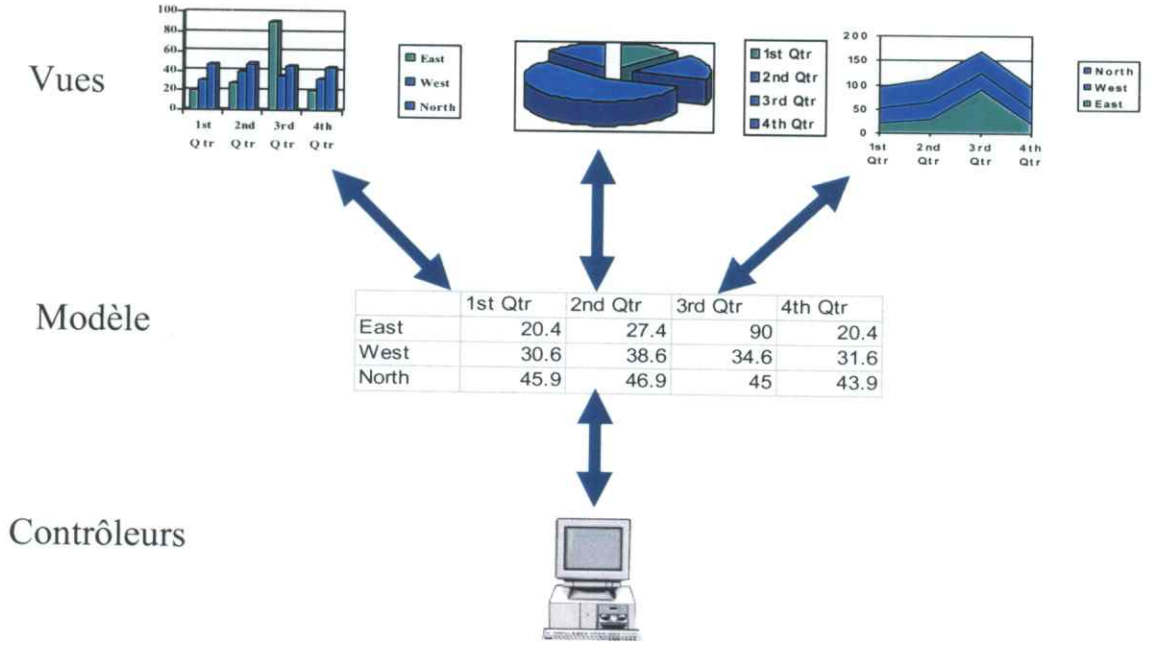


Figure B.3 : Exemple d'application de MVC

La figure **B.3** présente un exemple d'application du MVC,où le modèle est une table d'Excel et les vues sont différentes représentations graphiques de ce modèle.

5. LE PATTERN DE CONCEPTION OBSERVER

Le patron **observer** ou **observateur** est classé dans la catégorie des patrons comportementaux, sa description et sa documentation est donnée comme suit :

Intention : Le patron observer définit une dépendance ***un à plusieurs*** entre objets, lorsque un objet change d'état tous les objets qui en dépendent sont informés et sont automatiquement mis à jour. **[16]**

Alias : Dependents, Publish-Subscibe.

Motivation : Deux parties composent le patron de conception **Observer**, le **Sujet** (***Subject***) et l'**Observer** (***Observer***), elles sont liées par une relation **un à plusieurs** découplée (les deux parties sont distinctes) facilitant leur réutilisation.

Le schéma ci-dessous illustre l'application du pattern de conception observer sur l'exemple précédent.

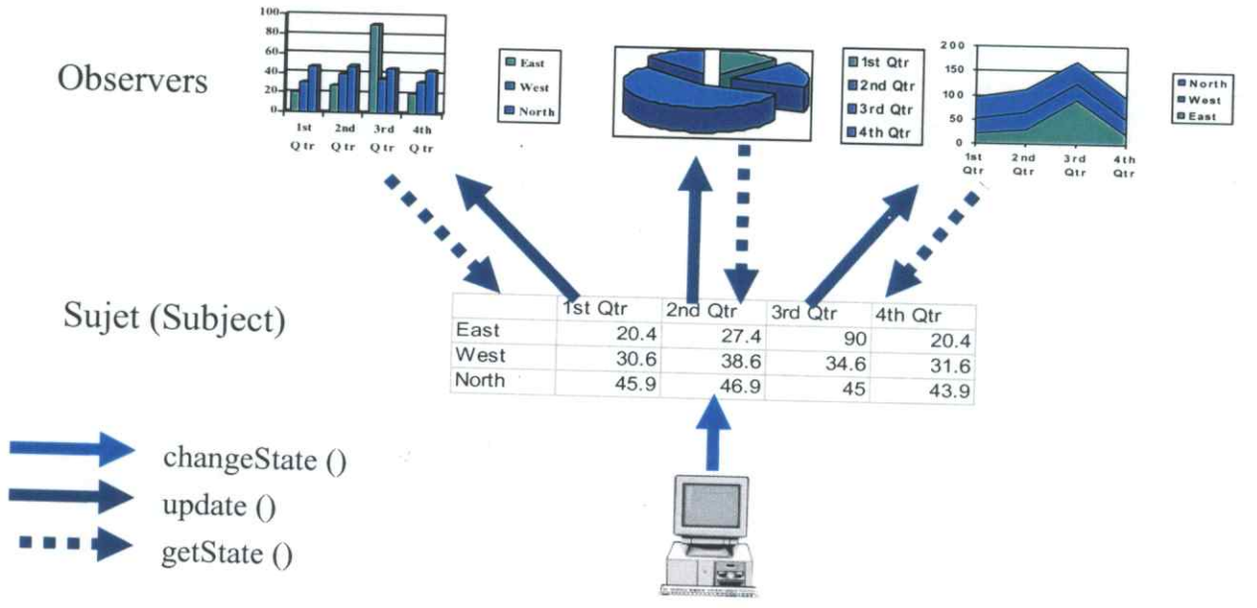


Figure B.4 : Exemple d'application du pattern Observer

Indications d'utilisation (applicabilité) : Ce patron peut être utilisé dans les cas suivants **[17]** :

- Lorsqu'une abstraction présente deux aspects l'un dépendant de l'autre.
- Lorsque l'objet *Subject* ne connaît pas le nombre de ses *Observers*.
- Lorsque l'objet *Subject* peut être capable de notifier ses *Observers* sans les connaître.

Structure : La structure du patron observer est donnée par le diagramme de classe suivant :

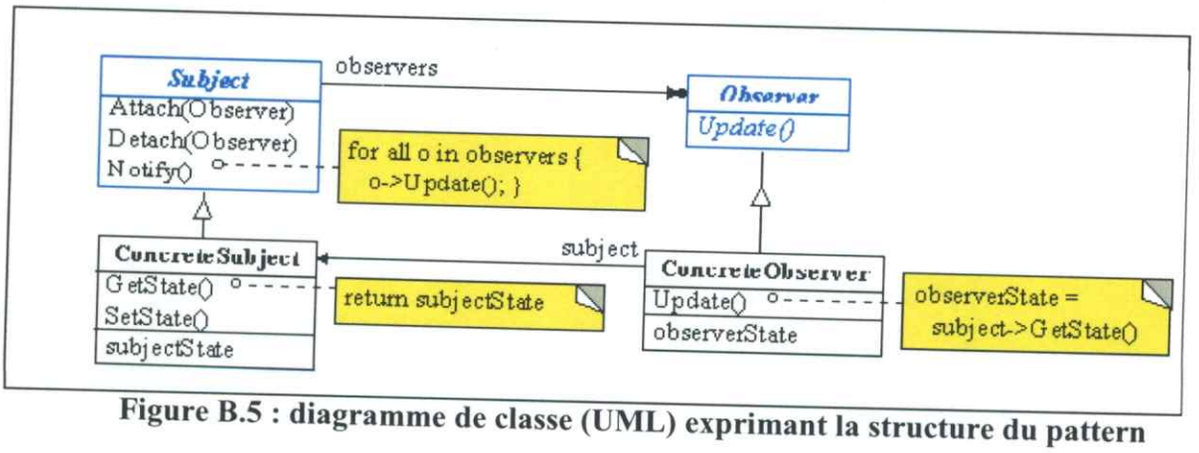


Figure B.5 : diagramme de classe (UML) exprimant la structure du pattern Observer.

Participants :

- **Subject**

  Fournit une interface pour attacher et détacher les objets observers.

- **Observer**

  Définit une interface pour la fonction update () qui doit être exécutée après la notification

- **ConcreteSubject**

  Représente l'objet observé.

  Sauvegarde l'état intéressé par l'objet de **ConcreteObserver**.

  Envoie une notification à ses Observers lorsqu'il change d'état.

- **ConcreteObserver**

  Sauvegarde l'état de l'objet observé.

  Implémente la fonction update () de l'interface **Observer** pour récupérer le nouveau état de l'objet observé.

Collaborations :

- L'objet **ConcreteSubject** notifie (informe) ses observers d'un changement dans son état.
- Après la notification, l'objet **ConcreteObserver** récupère l'état de l'objet **ConcreteSubject**. Ensuite utilise cette information pour changer à son tour son propre état interne.

Le diagramme de séquence suivant montre l'interaction entre les deux parties du pattern:

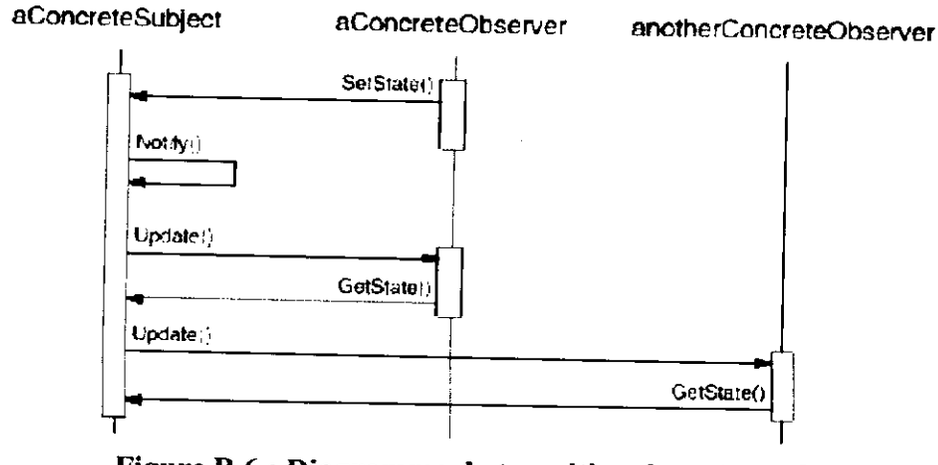


Figure B.6 : Diagramme de transition du patron observer.

Utilisations courantes :

L'une des utilisations les plus connues du pattern observer est L'architecture MVC (*Model View Controller*) ; où la Vue est l'*Observer* et le Modèle est le *Subject.*

Patrons Apparentés : Médiator, Singleton ;

Conséquences :

Avantages :

- Supporte la communication par **Diffusion Générale** (Broad-cast).
- Le couplage est abstrait entre le *Subject* et l'*Observer*, chacun peut être réutilisé individuellement
- Le rapport entre le *Subject* et l'*Observer* est dynamique, il peut être établi au temps d'exécution. Cela donne beaucoup plus de flexibilité de programmation. **[17]**

Inconvénients :

- Un simple changement d'état peut provoquer de nombreuses mises à jours **[16]** coûteuses en temps et en ressources.
- La possibilité de lenteur des mises à jours peut provoquer la suspension du sujet tant que tous ses observers ne sont pas encore mis à jours. **[16]**

Implémentation :

- Subject doit connaître ses observers. Le subject doit avoir une référence sur ses observers. Par exemple : un tableau, une liste chaînée, ou autre,...
- Observation de plusieurs subject. On peut implémenter le rapport *plusieurs à plusieurs* entre le *Subject* et l'*Observer*. L'interface Update dans observer doit savoir quel *Subject* a changé (a envoyer la notification). Une des implémentations possible est de passer le *Subject* en paramètre dans l'opération Update.
- Qui déclenche l'événement update (l'opération Notify dans *Subject*) ? Les opérations de changement d'état dans *Subject* déclenche Notify.
- Il faut s'assurer que l'état de *Subject* est cohérent avant la notification. Autrement, un *Observer* doit demander au *Subject* à travers l'opération GetState().
- Eviter d'envoyer update à un Observer spécifique:dans les deux modèles push et Poll.
  - **Push model**: Subject envoie les détails sur le changement au Observer.

- **Poll model** : Subject envoie le minimum sur le changement au Observer et c'est Observer qui demande le reste de l'information.

- Spécifier explicitement les modifications aux intéressés. Un observer peut être intéressé par un événement spécifique. Ceci peut améliorer l'efficacité de update.

Exemple de code :

Le pattern observer en java :

L'API java offre deux packages implémentant le patron observer **[18]** :

Java.util.Observer : c'est une interface qui comporte la fonction ***update()***, cette fonction est exécutée quant un observer est notifié d'un changement.

Java.util.Observable : comporte la classe Observable qui sert de classe mère pour toute classe qui veut avoir les fonctionnalités du model.

Elle comporte les fonctions et procédures suivantes :

- public void addObserver(Observer obs) : ajoute un observer a la liste interne d'observers.
- public void deleteObserver(Observer obs) : supprime un observer de la liste.
- public void deleteObservers() : vide la liste.
- public int countObservers() : retourne le nombre d'observers présents dans la liste.
- protected void setChanged() : modifie l'indicateur interne de changement pour qu'il indique que l'objet a changé.
- protected void clearChanged() : efface les indicateurs internes de changement.
- public boolean hasChanged() : retourne « true » si cet observable a changé.
- public void notifyObservers() : avertir les observers d'un changement.
- public void notifyObservers(Object obj) : avertir les observers d'un changement. Passe l'objet en paramètre.

Voici un simple exemple qui explique l'utilisation de ce pattern en Java :

Dans cet exemple nous avons un modèle (MyModel) , une vue (MyView) et un contrôleur (MyController) et la class principale MainFrame.

La class MainFrame est utilisée pour attacher les observers (`model.addObserver(mv) ;`)

Le contrôleur change le **counter** du modèle lors du click sur le bouton (par l'exécution de la fonction `increaseCounter()` du modèle).

Ce changement d'état doit être remarqué par la vue, c'est pourquoi le modèle fait appel à la fonction `setChanged()` puis `notifyObservers(new Integer(counter))` pour avertir la vue en lui passant en paramètre le nouveau état.

```
/*
 * MyController.java
 * Cette class représente le contrôleur
 */

import java.awt.event.ActionEvent ;
import java.awt.event.ActionListener ;

import java.util.Observable;
import java.util.Observer ;

import javax.swing.JButton ;
import javax.swing.JPanel ;

public class MyController extends JPanel implements ActionListener
{
   private JButton push ;  // le boutton de contrôle
   private MyModel model ; //le model controlé

   public MyController(MyModel m){

      model = m ;
      push = new JButton("Add Value");
      push.addActionListener(this);
      add(push);
   }

   public void actionPerformed(ActionEvent e)
   {
      model.increaseCounter();

   }

}
```

```
/*
 * MyModel.java
 * Cette class représente le modèle ( ConcreteSubject )
 */

import java.util.Observable;

class MyModel extends Observable
{
   private int counter ;

   public MyModel(){
       counter = 0 ;
   }

   public void increaseCounter()
   {
      counter = counter + 1 ;
      setChanged(); //indique que son état a changé

      notifyObservers(new Integer(counter)); //avertit les observers
   }

   public int getCounter()
   {
      return counter ;
   }
```

```
/*
 * MyView.java
 * Cette class représente la vue( ConcreteObserver )
 */

import java.util.Observable;
import java.util.Observer ;
import javax.swing.JLabel ;
import javax.swing.JPanel ;

public class MyView extends JPanel implements Observer
{
   private JLabel label ;

   public MyView(){
      label = new JLabel("0");
      add(label);
   }

   public void update(Observable ob , Object arg ){

      if(ob.instanceof MyModel)
         m = (MyModel) ob ;

      // récupérer la nouveau état du modèle
      label.setText(" "+ m.getCounter());
      repaint();
   }
}
```

```
/*
* MainFrame.java
* Cette class représente la class Main dans laquelle on attache les
*  observers
*/

import java.awt.Container;
import java.awt.GridLayout;
import javax.swing.JFrame ;

public class MainFrame extends JFrame {

public MainFrame(){

Container c = getContentPane();
c.setLayout(new GridLayout(2,1));

MyModel    model = new MyModel();
MyView       mv = new MyView();
MyController mc = new MyController(model);

c.add(mc);
c.add(mv);
setSize(100,100);

model.addObserver(mv);//attacher l'observer (MyView)

}

public static void main(String args[]){

(new MainFrame()).setVisible(true);
}
}
```

6. LE PATTERN DE CONCEPTION STRATEGY

Intention : Un programme ayant besoin d'un service ou d'une fonction, qui peut être exécuté de plusieurs manières, est candidat au pattern Strategy. Les programmes choisissent un algorithme précis; soit par un calcul d'efficacité soit à travers un choix utilisateur. **[15]**

Alias : Policy.

Motivation : Le pattern Strategy est composé de plusieurs algorithmes encapsulés dans une classe appelée ***le Contexte***. Le programme client choisit l'un des algorithmes, ou dans certains cas le meilleur est sélectionné par le Contexte, cela se fait dynamiquement pendant l'exécution. Ce pattern encapsule des algorithmes qui font plus ou moins la même chose.

Indications d'utilisation :

- De nombreuses classes associées ne diffèrent que par leur comportement. Stratégie offre un moyen de configurer une classe avec un comportement parmi plusieurs.

- On a besoin de plusieurs variantes d'algorithme.

- Un algorithme utilise des données que les clients ne doivent pas connaître. Employez le modèle de Stratégie pour éviter d'exposer un algorithme complexe spécifique aux structures de données.

- Une classe définit beaucoup de comportements et ceux-ci apparaissent comme des déclarations multiples conditionnelles dans ses opérations. Au lieu d'avoir beaucoup de conditionnels, déplacer ces branches conditionnelles dans leur propre classe de Stratégie.

Structure :

Le comportement général est implémenté par une classe abstraite, les sous-classes (concrètes) définissent les comportements qui diffèrent. Le diagramme de classe suivant exprime l'implémentation la plus générale du pattern Strategy :

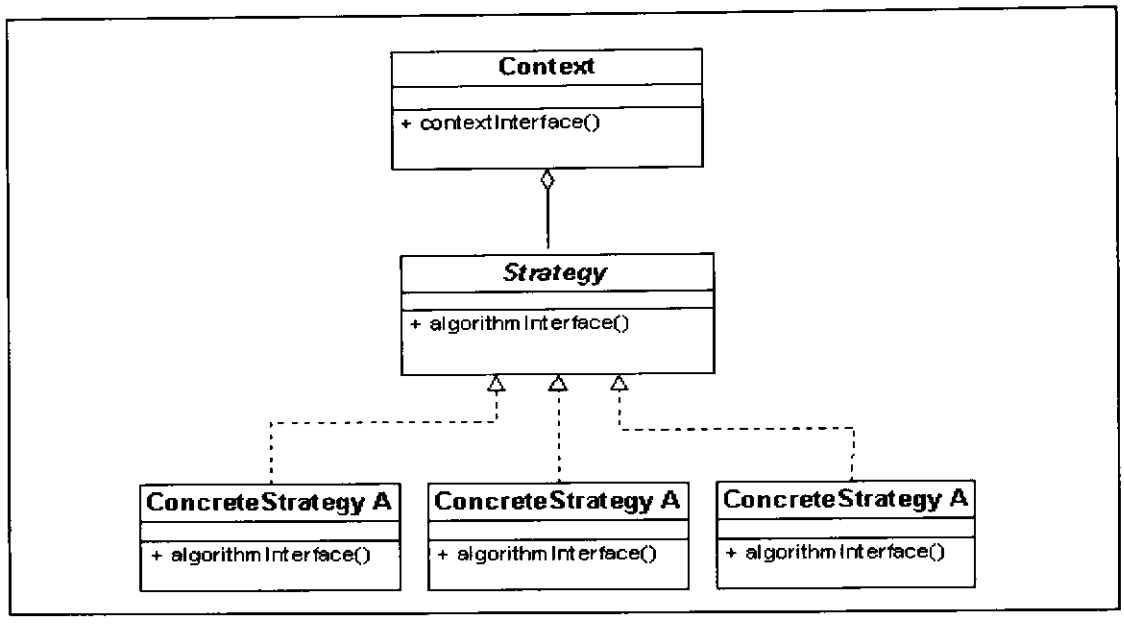


Figure B.7 : diagramme de classe (UML) exprimant la structure du pattern Strategy

Participants :

- **Strategy :**

  Déclare une interface commune à tout algorithme supporté.

- **ConcreteStrategy :**

  Utilisée par le **Context** pour appeler l'algorithme définie par **ConcreteStrategy**.

  Implémente l'algorithme en utilisant l'interface **Strategy**.

- **Context :**

  Configuré par une instance de **ConcreteStrategy**.

  Doit définir une interface qui permet à **Strategy** d'accéder à ses données.

Collaborations :

- La stratégie et le Contexte interagissent pour implémenter l'algorithme choisi. Un contexte peut passer toutes les données exigées par l'algorithme à la stratégie quand

l'algorithme est appelé. Alternativement, le contexte peut se passer comme un argument aux opérations de Stratégie.

Utilisations courantes :

On peu citer quelques cas d'utilisation du pattern Strategy :

- Sauvegarder des fichiers dans des formats différents,
- Compression de fichiers dans des formats différents,
- Capture vidéo selon des schémas de compression différents,
- Tracé d'un ensemble de données dans différents formats : graphe, histogramme, ...etc.

Patrons apparentés : Flyweight.

Conséquences :

- Les familles d'algorithmes liés
  - Les algorithmes sont arrangés dans une hiérarchie d'héritage.
  - Les ressemblances entre des algorithmes peuvent être déplacées dans la classe de base de Stratégie.
- Une alternative à sous classes.
  - Obtenir une conception plus claire, plus extensible que les sous classes du contexte.
  - L'algorithme peut varier indépendamment du contexte.
  - L'algorithme peut être changé dynamiquement au temps d'exécution.
- Les stratégies éliminent des déclarations conditionnelles
  - Quand les stratégies sont encapsulées dans une classe il est difficile d'éviter les conditionnels.
  - Des déclarations conditionnelles sont difficiles à maintenir,encombrantes et créent des dépendances inutiles.

- Un choix d'implémentation
  - Les clients peuvent choisir parmi différentes implémentations celle qui leurs convient.
- Les clients doivent connaître les différentes implémentations des différentes stratégies
  - Un inconvénient consiste en ce que les clients doivent connaître les effets de chacune des stratégies.

- Communication entre Stratégie et Contexte
  - Méfiez-vous du passage de beaucoup de données entre le Contexte et la Stratégie dans la quel la plupart des classes de **ConcreteStragy** n'emploie pas ces données.
  - Employez l'accouplement plus serré pour éviter cette situation.

Implémentation :

Définir la Stratégie et interfaces de Contexte (Voir collaborations) :

- Dans ce cas vous pouvez passer des données dont la ConcreteStrategy n'a pas besoin.

Exemple de code source :

Voici un simple exemple qui explique l'utilisation de ce pattern en Java :

L'interface **MainClass** représente le **Context.**

L'interface **Sort_Interface** représente la **Strategy**.

La classe **Sort** représente la **ConcreteStrategy.**

```
public Interface Comparable {

   public boolean lessThan(Object X);
   public boolean greaterThan(Object X);
   public boolean equal(Object X);
}
```

- Un choix d'implémentation
  - Les clients peuvent choisir parmi différentes implémentations celle qui leurs convient.
- Les clients doivent connaître les différentes implémentations des différentes stratégies
  - Un inconvénient consiste en ce que les clients doivent connaître les effets de chacune des stratégies.

- Communication entre Stratégie et Contexte
  - Méfiez-vous du passage de beaucoup de données entre le Contexte et la Stratégie dans la quel la plupart des classes de **ConcreteStragy** n'emploie pas ces données.
  - Employez l'accouplement plus serré pour éviter cette situation.

Implémentation :

Définir la Stratégie et interfaces de Contexte (Voir collaborations) :

- Dans ce cas vous pouvez passer des données dont la ConcreteStrategy n'a pas besoin.

Exemple de code source :

Voici un simple exemple qui explique l'utilisation de ce pattern en Java :

L'interface **MainClass** représente le **Context.**

L'interface **Sort_Interface** représente la **Strategy**.

La classe **Sort** représente la **ConcreteStrategy.**

```
public Interface Comparable {

   public boolean lessThan(Object X);
   public boolean greaterThan(Object X);
   public boolean equal(Object X);
}
```

```
public class Student implements Comparable {
   // ...
   private double note;

   public boolean lessThan(Object X){

            return (note < ((Student)X).getNote() );
   }
   public boolean greaterThan(Object X){

            return (note > ((Student)X).getNote() );
   }
   public boolean equal(Object X){

            return (note == ((Student)X).getNote() );
   }

   public double getNote(){
            return note;
   }
   // ...
}
```

```
public class Sort implements Sort_Interface {

   public void sort(Comparable [] S){

         boolean sorted  ;

         do{
             sorted = true;

              for(int i=0 ; i< S.length()-1 ; i++ ){


                  if( S[i].greaterThan(S[i+1]) ){
                     swap(S , I , i+1);
                     sorted = false ;
                  }
             }

         }while ( sorted );

     }//end of sort

   private void swap(Object a[], int i, int j)
     {
         Object T;
         T = a[i];
         a[i] = a[j];
         a[j] = T;
     }
}
```

```
public class MainClass {
    // ...
    private Comparable[]    array;
    private Sort_Interface method;

    public MainClass(){

          array  = new Student[10];
          method = new Sort();
    }

    public void test(){
           // ...
              method.sort(array);
          // ...
    }

    // ...
}
```

La classe **MainClass** représente la class principale du programme, cette class possède deux attributs, method et array .

L'attribut **method** est une référence sur l'interface **Sort_Interface** et qui peut être initialisé par n'importe quelle instance d'une classe qui implémente **Sort_Interface**, pour changer l'algorithme de tri, il suffit de changer cette initialisation et la suite du programme reste inchangé.

Dans la classe **Sort** la méthode **sort(Comparable[])** est appliquer sur un tableau d'instances de classes qui implémentent l'interface **Comparable.**

ANNEXE C

LES MÉTHODES DE DÉVELOPPEMENT AGILES

« XP – EXTREM PROGRAMMING »

De nombreux projets sont basés sur le cycle en V (Figure C.1). Cependant, cette méthode est souvent critiquée **[19]**. Le premier reproche qui peut être fait au modèle en V est d'écrire d'abord et de réaliser les jeux de Tests Unitaires après la réalisation des Objets à tester. Cela signifie qu'on n'effectue pas de tests sur un objet en fonction des caractéristiques qu'il doit posséder. On peut avoir l'idée d'adapter les tests a un objet qui ne satisfait pas tout les besoins . On parle alors de tests avec *a priori* **[19]**. Les tests sont aussi souvent sacrifiés ou stoppés dans le cas de dépassement des délais, des objets non fiables ou incorrects ne sont pas détectés. L'intégration finale des composants développés demande dans ce cas un très grand effort, il devient difficile de construire un système convenable à partir d'éléments en partie incorrects.

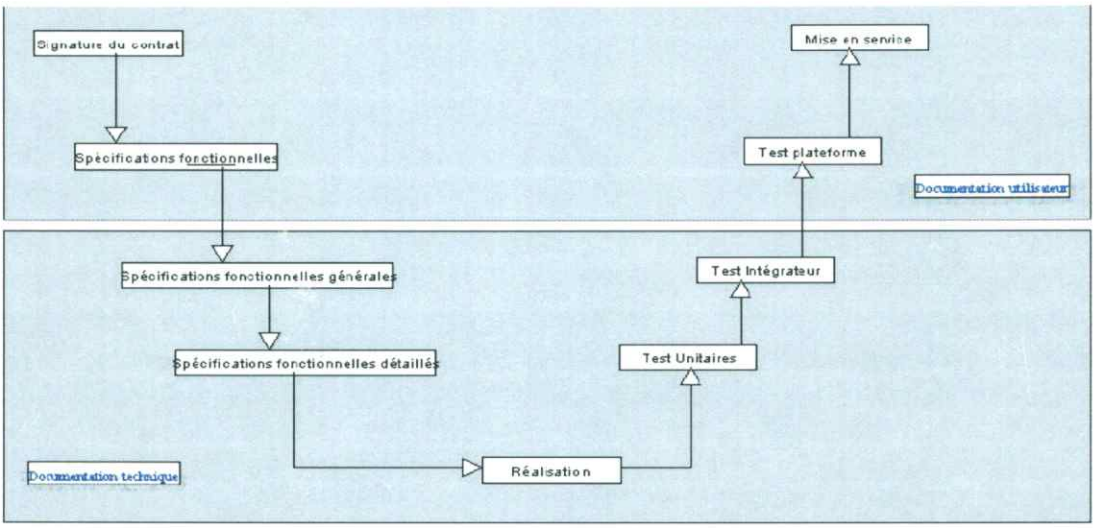


Figure C.1: schéma du modèle de développement en V

L'intégration finale dans un tel model est génératrice de bugs importants et de problèmes d'optimisation. Il n'est pas sûr que le système puisse fonctionner immédiatement, et au mieux en utilisant toutes les capacités des parties intégrées. Le risque de ne pas livrer le système dans les délais devient très important.

Dans le système en V, le client ne reçoit le logiciel qu'à la fin du projet. Il ne peut donc constater les dérapages, tant fonctionnels que techniques, de l'équipe de développement, il ne peut pas corriger ses spécifications dans le cas ou où il aurait a fait une erreur d'appréciation

de ses besoins.

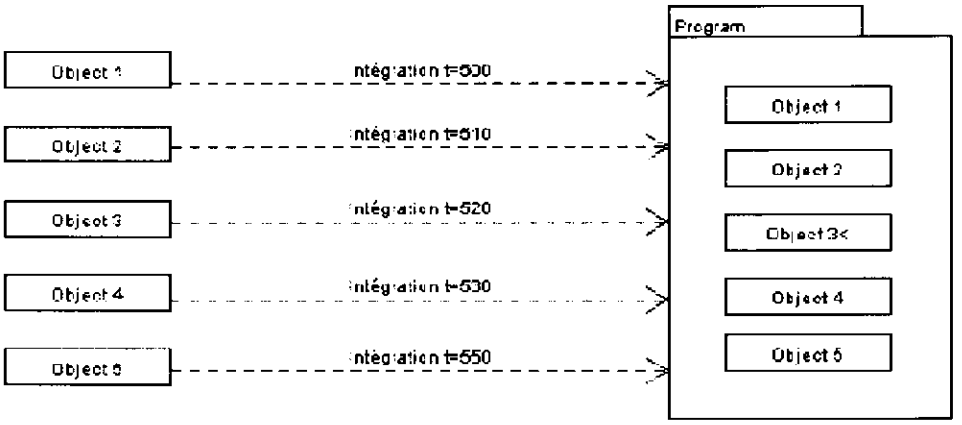


Figure C.2 : schéma d'intégration dans une méthode non itérative

Afin d'éviter ce genre de problèmes il nous faut explorer de nouvelles manières d'organiser, de diriger et de planifier le développement des systèmes informatiques. Les méthodes itératives agiles nous permettent d'éviter de tomber dans des problèmes, tels que décrites plus haut.

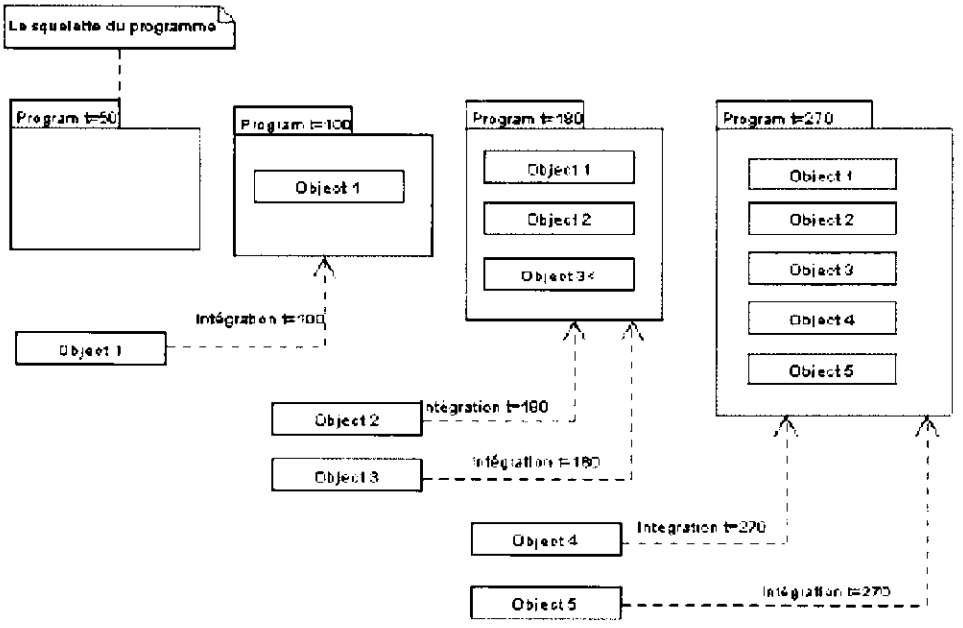


Figure C.3: schéma d'intégration dans une méthode itérative

3.2. Historique :

En février 2001, un groupe initial de 17 méthodologues issus de différents horizons ont formé l' « *Agile Software Development Alliance* ». Ce groupe a définit un manifeste qui

encourage d'autres manières de développer du software, puis se basant sur ce dernier, il a établit une collection de principes qui définissent un critère d'agilité pour le processus de développement de logiciels ; tel que la Modélisation Agile. **[20]**

3.3. Les valeurs de l'Alliance Agile :

3.3.1 individualités et interactions à travers le processus et les outils :

Le facteur le plus important a considérer est le coté humain (employés, membres de l'équipe) et la manière avec la quelle les personnes travaillent ensemble, si cet aspect n'est pas pris en considération, les meilleurs outils et processus ne seront d'aucune utilité.

3.3.2 Du software avec une documentation compréhensible :

Une documentation, écrite proprement, est un guide précieux, qui permet de comprendre pourquoi et comment le système a été développé ainsi que son utilisation. Cependant, il ne faut pas oublier que le but principal du développement de logiciels est de créer du software et non de la documentation.

3.3.3 La collaboration du client à travers la négociation d'un contrat :

Pour avoir du succès dans un développement, les développeurs investissent dans un effort de découverte, cela pour anticiper les besoins des clients.

3.3.4 Répondre aux changements à travers un plan :

Le changement est une réalité dans le développement de logiciels. Un plan de projet doit être malléable, il doit donc facilement changer avec le changement de situation. **[20]**

3.4. L'AM (Agile Modéling) :

La méthode AM est une collection de pratiques, guidées par des valeurs et des principes. L'AM n'est pas un processus prescriptif, en d'autres termes, la modélisation agile ne définit pas les procédures de création d'un modèle donné, mais, elle fournit des conseils pour être efficace en tant que modeleur **[21]**.

3.5. Principes fondamentaux de l'AM :

- Assumer la simplicité : partir du principe qui veut que la solution la plus simple est la meilleure.
- Accepter le changement,
- La réalisation de l'effort suivant est le but secondaire,
- Changement incrémental,
- Maximiser l'investissement des managers dans le projet,
- Identifier pourquoi et pour qui un model est crée,
- Modèles multiples **[21]** : Les modèles ne se limitent pas aux diagrammes UML seulement,
- Un travail de qualité : les développeurs doivent fournir un effort pour établir des artifices permanents tels du code source, documentation utilisateur et support technique de qualité,
- Retour arrière rapide,
- Produire un logiciel de haute qualité est le but principal,
- Créer le minimum de documentation possible,
- Le contenu est plus important que la représentation,
- Apprendre des autres tout en travaillant avec eux,
- Connaissance des différents models : connaître les forces et les faiblesses de chaque technique avent de l'applique **[21]**,
- Connaissance des différents outils,
- Communication honnête et ouverte,
- Adaptation locale : modifier l'AM selon l'environnement du projet.

3.6. Les buts de l'AM:

La modélisation Agile poursuit trois (3) buts :

- Définir et montrer comment mettre en pratique une collection de valeurs et de principes.

- comment appliquer des techniques de modélisation dans le développement de logiciels prenant une approche agile tel qu'XP, DSDM ou SCRUM.
- Comment peut on améliorer la modélisation sous un processus prescriptif tel que RUP (Rational Unified Process).

3.7. Les pratiques de l'AM :

Les principales pratiques de l'AM sont les suivantes :

- Participation active des managers,
- Appliquer le(s) artifice(s) appropriés,
- Propriété collective,
- Le plus de tests possibles,
- Créer plusieurs models en parallèle,
- Description simple des models,
- Présenter les models publiquement,
- Itérer vers un autre artifice,
- Modélisation par petits incréments,
- Modéliser avec les autres,
- Démontrer les models avec du code,
- Utiliser les outils les plus simples,
- Utiliser des standards pour la modélisation,
- Utilisation élégante des patrons,
- Détruire les models temporaires,
- Modéliser pour communiquer,

- Modéliser pour comprendre,
- Réutiliser des ressources existantes,
- Mettre à jour seulement si nécessaire.

3.8 Développement agile de logiciels :

Le développement agile consiste à prendre une approche itérative et incrémentale, incluant différents models. Les développeurs Agile utilisent des itérations en avant et en arrière entre les taches telles que :

- Modélisation de données,
- Modélisation objet,
- Refactoring,
- Implémentation, et
- Amélioration et test de performances.

L'algorithme suivant explique de manière claire, précise et concise, la marche a suivre lors d'un développement agile :

Pour chaque tache faire :

Tant que le problème reste à résoudre
Si on peut avancer d'une étape **alors** attaquer cette étape
Sinon - se replier jusqu'au point où on pourrait résoudre le problème
- modifie ou incorporer les nouveaux outils

Les besoins doivent tracer le développement du schéma objet, qui a son tour guide la construction du modèle des donnée et du code source. Les chalenges de performances et les caractéristiques des plateformes doivent motiver des changements conceptuels évolutifs du schéma objet.

L'une des principales caractéristique des développeurs Agile est qu'ils sont répartis sur plusieurs petits groupes, travaillant à résoudre des problèmes, prêts à changer (nouveaux arrivants, départs, regroupements, ...etc.), avec un coût réduit. **[22]**

4. LA METHODE DE DEVELOPPEMENT XP (EXTREME PROGRAMING)

1. Introduction :

Le processus actuellement utilisé, se focalise sur le papier plutôt que sur le côté humain. Le papier ne pense pas, ne résout pas de problèmes et ne s'adapte pas aux changements **[23]**.

2. Description générale d'XP :

XP est un processus agile de développement de logiciels qui accorde une grande d'importance au coté humain (membres de l'équipe de développement). Ainsi, il fournit un cadre dans lequel les personnes communiquent de manière efficace.

XP procède avec une approche différente en ce qui concerne l'analyse et la conception. Les diagrammes produits par UML sont souvent ignorés dès qu'on crée du code. XP se concentre donc sur le code et les cas d'utilisation, en ignorant les étapes intermédiaires.

XP est un processus pour des équipes ambitieuses, qui veulent être présentes sur le marché de manière rapide, tout en gérant les risques liés a la rapidité en utilisant des méthodes humaines. Les risques engendrés par la rapidité d' XP sont atténués en travaillant par paire, en faisant beaucoup de tests et en communiquant, pratiquement quotidiennement, avec le client. XP n'est pas un processus haut risque ; il va rapidement mais sûrement. **[23]**

Kent Beck, auteur d' "extreme programming explained" a dit qu'XP est une méthode légère pour de petites et moyennes équipes de développement **[24]**. La philosophie d'XP est qu'il faut payer pour nos besoins, quand on a besoin, et pas avant. **[23]**

3. Les valeurs d'XP :

XP est une collection de pratiques guidées par des valeurs et des principes, cela découle du fait qu'XP est un processus agile de développement. Les valeurs d'XP sont les suivants :

1 La communication,
2 La simplicité,
3 Les retours arrière,
4 Le courage.

4. Les principes d'XP :

L'ingrédient clé dans la formule qui permet de rassembler les deux buts (principes) conflictuels, développement rapide et développement sur, est d'utiliser une méthode telle qu' XP. **[25]**

L'élément qui manque dans pratiquement tous les processus de développements, lourds ou autres, est une forte concentration sur l'architecture du software.

Les compagnies qui déploient un processus lourd ont intérêt d'utiliser des processus légers qui se concentrent sur la production de software avec une architecture solide. L'avantage de ce processus et qu'il permet d'atteindre le marché de manière rapide avec un produit de haute qualité. **[25]**

5. Les pratiques d'XP :

X. P. est une collection des meilleures pratiques :

5.1 Les rôles XP :

une équipe soudée, bien organisée et capable de créer un environnement de communication entre ces membres est une clé principale dans la réusite d'un projet logiciel.

chaque me"mbre de l'équipe se voit attribué un rôle bien présis, qu'il doit assumer, XP déffinit trois rôles dans une équipe de développement :

1. le rôle Programmeur :

Le programmeur doit écrire, modifier et avoir une bonne connaissance du code. Afin d'éviter les erreurs il doit effectuer des tests unitaires et des tests "scenarios" **[19]**. le programmeur communique de manière directe et permanente avec le Client, cela lui permet de reverifier les besoins, de modifier son analyse et son design, de verifier les tests et de reformuler son code.

Le programeur doit :

1. savoir ce qui est demandé, avec des priorités clairement déclarées,
2. fournir un travail de qualité en toute occasion,
3. travailler a un rithme de travail durable.

Le progrtammeur a le droit:

4. de demander et de recevoir de l'aide de la part de ses collègeus et du *Client*
5. d'émettre et de réviser ses propres éstimations de coûts,
6. d'accepter des responsabilités, qui ne peuvent pas lui être imposées,

2. le rôle Client:

Le rôle *Client* est atrribué au membre de l'équipe qui reteprésente les interets et les besoins du client effectif. Il n'est pas forcement le payeur, il peut être un consultant; un ingénieur détaché auprès du dépositaire **[19]**.

Le *Client* a la responsabilité de définir les fonctionnalités et les caractéristiques du système a concevoir ainsi que de la manière avec la quelle elles doivent être conçues, il doit rester en communication permanente avec les programmeurs, tout en faisant preuve de simplicité et de précision dans ses explications. La communication directe et franche entre ces deux catégories de membres (*Clients*, *programmeurs*), permet de faire des retours arrières rapides dans les cas ou le Client se rend comte que les besoins qu'il a formulé au début étaient inadaptées (le *Client* peut se tromper).

Le client as le droit **[19]** :

- à un plan d'ensemble, montrant ce qui peut être accompli, quand et à quel coût.
- de voir les progrès sur une application , en permanence, et qui passe l'ensemble

des jeux de tests.

- de changer d'avis, de substituer des fonctionnalités sans en payer un prix exorbitant.
- 4. d'être informé, en temps réel, des modifications faites au calendrier de réalisation, afin de pouvoir réduire la charge de travail pour retomber sur la date de livraison initiale.
- d'annuler le projet à tout moment et de disposer d'une application utile et utilisable (mais limitée en fonctionnalités).

3. le rôle Coach:

Le *Coach* est l'élément coordinateur dans un projet XP. il doit s'assurer de la bonne application de la méthode XP, il doit aussi avoir une connaissance des différentes méthodes de gestion de projets, cela lui permettera de peufiner la mise en oeuvre XP et de trouver des solutions de recherche à cxertaines parties de la méthode.

le *Coach* doit être présent partout lors du début pour s'assurer que le projet démarre sur de solides bases. son but principal est que la gestion du projet puisse fonctionner sans lui le pus rapidement possible. le *Coach* doit s'éfacer, au fur et a mesure, afin de laisser à l'équipe une autonomie maximale.[Déschalier2003]
Le c*oach* doit savoir **[19]** :

- communiquer sans imposer ses idées
- écouter, motiver son équipe et renforcer l'ésprit de groupe.
- être un pédagogue convaincu,
- se fâcher quant il le faut.
-

5.2 Le planning :

XP est un processus itératif de développements qui nécessite une planification des taches et des itérations. Au cours des plannings, les manager et les programmeurs déterminent la portée de chaque version (release), les programmeurs estiment le coût de chaque caractéristique. Les managers partitionnent alors le développement des caractéristiques en petites itérations.

5.3 Les scénarios utilisateurs:

Un scénario représente une caractéristique du système. Le manager construit des scénarios selon les besoins du client, puis le soumet aux développeurs.

5.4 De petites versions :

Les programmeurs construisent un système en définissant plusieurs versions. Une version est un ensemble d'itérations qui fournissent des caractéristiques évaluables aux utilisateurs du système.

5.5 Exécution de tests fréquents :

Les tests sont un ensemble de donnée et de scénarios de départ, ainsi que les résultats, correspondants à cet ensemble, qui doivent apparaître après l'utilisation de la partie du système à tester. Ils déterminent qu'un scénario est complet ou non. Les managers écrivent un ensemble de tests que les programmeurs peuvent exécutent plusieurs fois par jour.

5.6 Environnement de travail ouvert :

Pour faciliter la communication, les membres de l'équipe travaillent dans un environnement ouvert avec toutes les personnes et les équipements facilement accessibles.

5.7 Les tests orientés design :

Les programmeurs écrivent du software en plusieurs petites étapes vérifiables :

- premièrement, on écrit un petit test,
- ensuite, on écrit plus de code pour satisfaire le test,
- puis, un autre test est écrit,
- est ainsi de suite.

5.8 La métaphore du système :

Une métaphore du système fournit une idée ou un modèle du système. Elle offre un contexte pour la dénomination des choses dans le software, rendre le software communicatif avec les programmeurs.

5.9 Une conception simple :

Dans XP la conception doit être le plus simple possible pour l'implémentation du scénario courant. Les programmeurs ne construisent pas des cadres d'application et des infrastructures pour des caractéristiques qui peuvent apparaître.

5.10 Refactoring :

Même s'il est très expérimenté dans un domaine, l'esprit humain ne peut exécuter une production parfaite dès le premier jet. Cette assertion est d'autant plus vraie s'il s'agit :

- D'une nouvelle expérience,
- D'un projet trop complexe pour pouvoir être perçu dans son ensemble, par l'esprit humain,
- Si la personne ne possède pas les capacités ou la 'fibre' la mieux adaptée à ce type de problématique.

Lorsque l'on découvre un nouveau milieu, il est difficile, faute d'une expérience concrète dans le domaine, de faire immédiatement les bons choix. La recherche de la solution passera par une phase d'apprentissage et de recherche de solutions moyennes mais faciles à mettre en oeuvre. **[19]**

Le refactoring est basé sur le principe que : toute production informatique n'est pas vue comme un produit fini et parfait, mais, comme un élément susceptible d'être remanié pour la raisons suivantes : **[19]**

- Chacun a pour responsabilité d'améliorer le code source en fonction de ses compétences (responsabilité commune de code source).
- Un Objet n'est valide que s'il répond aux règles d'échanges avec son environnement. Si l'environnement évolue, l'objet doit s'adapter à ses nouveaux interlocuteurs. Le programme évoluant sans cesse, les objets qui le compose ne doivent pas être figés mais savoir s'adapter aux nouveaux challenges.

Dès qu'on ajoute des caractéristiques au projet, le design commence à être perdu. Si ça continue, le design se détériorera. Le refactoring est un processus qui rend le design clair de manière incrémentale.

5.11 Intégration continue :

Les programmeurs intègrent et testent le software plusieurs fois par jour. De grandes branchements et fusions de code sont alors évitées.

5.12 Propriété collective :

L'équipe possède l'intégralité du code. Les programmeurs modifient n'importe quelle partie du code si nécessaire.

5.13 Les standards de code :

Le code a besoin d'avoir un style commun pour faciliter la communication entre programmeurs. L'équipe possède l'intégralité du code ; donc possède le style du code.

5.14 Programmation par paires :

Deux programmeurs collaborent à résoudre des problèmes. Si deux personnes arrivent à lire un même code source, il est plus probable qu'un troisième programmeur pourrait aussi le lire. En plus, avec la propriété collective du code, n'importe qui peut effectuer des changements dans ce code si nécessaire.

5.15 Allure Durable :

L'équipe doit rester fraîche pour produire du logiciel de manière efficace. Une façon de s'assurer qu'une équipe ne fasse pas beaucoup d'erreurs est de ne pas augmenter le nombre d'heures supplémentaires (éviter le surmenage).

5.16 De courtes Itérations :

Des itérations courtes (deux semaines) permettent des retours arrière rapides. Deux semaines semblent vraiment courtes si on n'as pas fait XP auparavant. On peut choisir une itérative de quatre semaines. Le travaille est meilleur que dans des itérations de deux ou trois mois, mais des itération

s de deux semaines donnent beaucoup plus de retours arrières.

6. XP et architecture du software :

X. P. est un processus léger qui accorde une attention particulière à l'architecture du système. Elle se base fortement sur les techniques de gestion de projets : estimation, gestion de l'ordonnancement. Mais son but principal et de produire du software avec une architecture solide. **[25]**

En résultat, le logiciel reste :

- facilement modifiable,

- facilement extensible,
- facilement développable,

7. L'analyse et la conception dans XP :

Malgré que dans XP on produit des tests incrémentaux, l'analyse et la conception restent des étapes présentes dans le développement d'un système. En effet, XP accorde une importance considérable à l'analyse et à la conception. Cependant, la réalisation d'une analyse externe importante et d'une grande documentation de conception n'est pas nécessaire. La documentation d'analyse est réduite à son minimum, ceux-là en écrivant des scénarios utilisateurs revus continuellement par les managers. La documentation du design est maintenue par les ingénieurs à travers le code, en effet le code lui-même est une documentation pour le design.

Dans XP la structure de code à une importance extrême. Un code qui ne représente pas sa propre documentation de conception subit une factorisation (refactoring) jusqu'à atteindre ce but. Un code obscur, dupliqué ou qui demande beaucoup de commentaires pour être compris n'est pas toléré. **[25]**

Le design d'un programme est l'architecture du code, c'est la décomposition du code en méthodes, classes, paquetages... Etc. ainsi que les relations entre ces différents éléments.

On peut représenter ces notions par des diagrammes ; mais ces derniers ne font pas le design. Ils ne sont que des proxys du design. Le design est en réalité dans le code.

X. P. n'évalue pas les "design -proxys", elle évalu leur expression directe dans le code.

7. Développement de logiciels suivant XP :

XP est une méthode itérative et incrémentale dominée par le code et basée sur la conception. On oriente continuellement le code suivant la meilleure conception.

Chaque version débute par une phase d'exploration, et chaque itération par une réévaluation de la conception courante et de l'architecture.

Un projet X. P. est rempli de milliers de petites micros itérations, chacune d'elles

contiennent un composant : d'analyse, de tests, de codage et de conception. L'ordre est très important ; l'analyse vient logiquement au début (pour concevoir il faut comprendre), elle est mélangée à un peu de design. En suite, on écrit des cas de tests qui décrivent la compréhension acquise pendant l'analyse. Le fait d'écrire des tests implique la présence d'une conception dans nos esprits. Puis on écrit du code qui permet aux tests d'être concluant. Bien sûr, il ne faut pas oublier que la conception est toujours présente dans cette étape. Enfin en fait le refactoring du code pour le rendre aussi simple est propre que possible. **[23]**

Ont choisi le refactoring plutôt que d'appliquer une attention initiale (analyse et conception initiale lourde) simplement par ce que le refactoring est moins coûteux et plus fiable.

Les utilisateurs XP n'ont pas peur du refactoring car :

- la présence de tests permet de prouver qu'aucune cassure ne peut passer inaperçue.
- le travail en paire apporte à chaque étape les avantages de deux esprits.
- la propriété collective du code permet aux membres de l'équipe d'être familier avec le programme.

La ressource la plus importante que doit gérer un ingénieur manager est le temps dont disposent ces ingénieurs. XP réduit le temps de développements en éliminant des rencontres, des documents, des vérifications... Etc. non nécessaires.

Le developpement dans une méthode XP suit la logique suivante:

Pour chaque tâche :

- Conception de la tâche
- s'assurer qu'elle s'accorde avec l'architecture du système,
- écrire des tests,
- écrire le code qui permet aux tests d'être concluants,
- faire le refactoring du code, en petites étapes, jusqu'à ce que sa conception soit jugée bonne.

Un haut niveau de développements dans X. P. est fait au :

- début du projet,
- début de chaque version,
- début d'échec itérations,
- début de chaque tâche,

Dans des projets en cascade, tous les besoins doivent être définis avant le départ du processus de conception. Dans XP aussitôt que vous avez un couple de semaines pour satisfaire un ensemble de besoins utilisateur, vous pouvez commencer le développement. Dans les premières itérations nous commençons par définir un produit simple. Si on utilise XP, on peut ignorer les détails sans être pénalisé plus tard. Dés le début du développement, la conception nous offre une certaine liberté dans la construction du système aussitôt qu'un certain nombre de fonctionnalités est identifié.

Le but principal dans la première itération était :

- construire une petite partie du produit,
- acquérir une expérience et

acquérir des compétences

BIBLIOGRAPHIE

[1] Robert C. Martin, « *Design Principles and Design Patterns* », 2000

[2] Garlan, Shaw, « *Patterns of Software Architecture* »

[3] Robert C. Martin, « *Designing Object Oriented Applications using UML* », 2d ed, Prentice Hall, 1999.

[4] Claude VIBET, « *Robots : principes et contrôles* », Edition Ellipses, 1987

[5] Paul, R. P., « *Robot Manipulators: Mathematics Programming and Control* », Cambridge, MA: MIT Press, 1981

[6] Frank L. Lewis, Darren M. Dawson, Chaouki T. Abdallah, « *Robot Manipulator Control Theory and Practice* », Edition MARCEL DEKKER,INC. 2004

[7] Grady Booch, James Rumbaugh, Ivar Jacobson, « *Le guide de l'utilisateur UML* », EYROLLES 2003

[8] R.E. Johnson, «*Frameworks = (components + patterns)* », Communications of the ACM, vol.40 , No.10, pp. 39-42, Octobre 1997

[9] Olivier BOISSIER , « *Analyse, Conception Objet (Design Pattern Introduction)* », SMA/G2I/ENS Mines Saint-Etienne, , Avril 2004

[10] Jacques Lonchamp, « *Génie Logiciel Quatrième partie:les techniques de conception* », CNAM - CRA Nancy,2003

[11] François Martel, « *Documentation Structurée du Design des Cadres D'application* », Mémoire présenté à la Faculté des études supérieures de l'Université de Montréal (M.Sc) . Octobre 1999.

[12] Khashayar Khosravi, « *Design Pattern-enabled object-oriented metrics* », Séminaire GÉLO (Laboratoire de génie logiciel, Université de Montréal) – 18 février 2004.LATECE Technical Report 2004.

[13] Stephen Lam « *Observer design pattern* », SENG 609.04 Design Pattern, 2 Avril 1998.

[14] E. Gamma, R. Helm, R. Johnson and J. Vlissides. « *Design Patterns: Elements of Reusable Object-Oriented Software* », Addison-Wesley, Reading, MA, 1995

[15] JAMES W. COOPER « *The Design Patterns : Java Companion* »,
Addison-Wesley 1998

[16] SDSU & Roger Whitney, « *Observer* »,
San Diego State University, CS 635 Advanced Object-Oriented Design & Programming,
Spring Semester, 2002.

[17] Stephen Lam, « *Observer design pattern* »,
SENG 609.04 Design Pattern, 2 Avril 1998

[19] Laurent DESÉCHALLIERS, « *Thèse Professionnelle Architecture et Reverse-ingeneering appliqués à la plate-forme SIGMA* »,
CESI DE ROUEN MASTERE SPECIALISE EN MANAGEMENT DE PROJET INFORMATIQUE EN MILIEU INDUSTRIEL LABORATOIRE P.S.I : PERCEPTION . SYSTEME . INFORMATION,
AVRIL-AOUT 2003.

[20] Ambler W. Scott, «*Introduction to Agile Modéling (AM)* »,
Ronin International Inc. White paper, 2002

[21] Ambler w. Scott , « *Artifacrts for Agile Modéling The UML and Beyond* »,
Ronin International Inc. 2004

[22] Michael Feathers, « *Agile Workflow*»,
Object Mentor, 2001

[23] Interview with Robert C. Martin on *eXtreme Programming*
The Objective View Journal Issue 2004

[24] Robert C. Martin *Object Mentor - Extreme Programming*
Object Mentor, Inc.2003

[25] Robert C. Martin *Object Mentor – why Object Mentor,*
Inc. 2003.

[26] Dr.-Ing. Andreas Jochheim, Dr.-Ing. Michael Gerke and Dipl.-Ing. Andreas Bischo,
« *Modeling and simulation of kinematic systems* »,
Control Systems Engineering group, Department of Electrical Engineering, University of Hagen, D-58084 Hagen, Germany,1999

[27] Christopher Alexander, « *A Pattern Language* »,
New York Oxford University, Press, 1977

[28] Christopher Alexander, « The timeless way of building »,
Oxford University Press, 1979

[29] G. Florijn, M. Meijers et P. van Winsen, « *Tool support for object-oriented patterns* »,
in Lecture Notes in Computer Science, vol. 1241, pp. 472-495, 1997.
(ECOOP97, Finland).

[30] BRICE CHARVIER, « *Editeur de diagrammes de séquence UML Rapport - version 2.0* »,
Organisme ESSI. 2004

[31] Ghizlane El Boussaidi et Hafedh Mili, « *Les patrons de conception: Représentation et mise en oeuvre* »,
Laboratoire de recherche sur les Technologies du Commerce Electronique Université du Québec à Montréal

[32] J. –P Gourret , « Modélisation d'images fixes et animées »,
Masson 1994