République Algérienne Démocratique et Populaire

Ministère de L'enseignement Supérieur et de la Recherche Scientifique

Université SAAD DAHLEB de BLIDA
Faculté des sciences
Département Informatique

MEMOIRE

Présenté pour l'obtention du diplôme de Master

Spécialité:

Informatique

Option :

Ingénierie du logiciel

Par : **DJILLALI Nora**

Thème

Amélioration du SMA To Aspect mobilité, Et optimisation

Promoteur :
Dr. Mahieddine Mohamed.

Soutenu le :

Devant le jury composé de:

Présidant

Examinateur

Examinateur

Année universitaire 2008-2009.

Aux deux personnes qui m'ont enseigné les premiers savoirs de ma vie
A mes parents

A mes Sœurs

A mes deux frères

Remerciement

En premier lieu, permettez-moi de remercier monsieur Mohammed Mahieddine, mon promoteur, pour m'avoir si judicieusement guidée, appuyée et pour cette confiance qu'il m'a accordée. Votre professionnalisme est pour moi une intarissable source d'inspiration.

Je tiens à remercier aussi très sincèrement pour leurs judicieux conseils et pour avoir gracieusement accepté de réviser cet ouvrage, monsieur le président et messieurs les membres du jury.

Mes hommages les plus respectueux vont également à tout le personnel du Service de soutien pédagogique et technologique de l'université de Blida.

A Mme Oukid-Khouas la directrice du laboratoire LDRSI (Laboratoire Des Recherches et Systèmes Informatiques) de l'Université de Blida, pour m'y avoir accueilli et avoir facilité le bon déroulement de mes travaux.

Je remercie les membres de l'équipe de recherche GLOODO(LRDSI) avec qui les discussions furent très constructives.

J'exprime enfin ma gratitude à ma famille pour leur soutien, leur présence et leurs encouragements. À ma mère qui m'a transmis ses valeurs. Elle vit à travers nous.

À mon père qui a fait de ma vie un paradis. Merci de ta compréhension.

Plusieurs autres personnes sont intervenues tout au long de ce projet. Quelques-unes pour de plus courts moments, d'autres plus longuement, mais toutes resteront dans ma mémoire.

Merci à vous !

Mes plus grands remerciements vont À Dieu !

RESUME

Le thème de recherche central de ce travail est l'application des outils de maint enabilité, comme la re-fabrication et la re-conception pour l'optimisation d'un framework à base de patrons pour la conception d'agents intelligents et mobiles sur téléphones portables.

Le travail a permis de réaliser un framework pour la sérialisation sécurisée pour l'envoi d'objets sur réseaux, ainsi que de son utilisation la migration de messages pour la destruction et la création d'agents à distance.

Nous avons aussi réalisé la mobilité puis la persistance d'une base de règles de production dans une base de données relationnelle distante, en utilisant un parsing objet vers schéma XML, et son envoi du schéma XML sur réseau, ensuite sa récupération à distance par une Servlet, puis sa transformation en un schéma objet, pour arriver à sa persistance par une méthode appropriée en tables, dans une base de données relationnelle.

Pour conclure, un bilan est adressé et des orientations futures sont exposées.

Mots Clés :

Maintenabilité de logiciel, Re-fabrication (Refactoring), Re-conception (Reengineering), Sérialisation, Mobilité, Framework, patron (pattern), Tests (tests unitaires, tests d'intégration,…), J2ME, Agent.

TABLE DES MATIERES

LISTE DES FIGURES

CHAPITRE I

INTRODUCTION

Savoir, c'est se souvenir.
ARISTOTE

INTRODUCTION

Il est indéniable que la tendance vers le développement à base de *Frameworks* basés sur des patrons s'impose, de plus en plus, depuis un certain nombre d'années.

Quant aux bénéfices attendus de l'utilisation de *Frameworks* à base de patrons, il s'agit essentiellement de réduire les coûts de production des logiciels, mais aussi d'en réduire les coûts de maintenance et d'en favoriser la réutilisation, par des utilisateurs, aussi bien que par des développeurs.

La conception et la réalisation des systèmes multi agent n'est pas un domaine facile [Fer95], surtout quand il s'agit d'un Framework développé pour téléphones portables, il n'est pas évident de résoudre les problèmes de dépendances et d'architecture. Le risque de développer des *Frameworks* rigides, fragiles et non maintenable (n'évolue pas avec les changements dans son environnement) est présent en permanence (conception initiale lourde, réalisation n'exprimant pas le design …etc.). Néanmoins, il existe des solutions qui permettent de résoudre se genre de problèmes ; Le design Orienté Objet et les patterns (patrons) permettent de concevoir des *Framework* plus fiables et plus flexibles.

Les bons *Frameworks* sont généralement le résultat de plusieurs itérations de conception et de beaucoup de travail impliquant des changements structurels. Ces changements peuvent impliquer une série de re-fabrications (refactoring).

La conception Orientée Objet est un ensemble de techniques se basant sur l'abstraction, pour créer le modèle d'un domaine, et de définir les relations et les interactions entre les éléments de ce domaine.

Le principal but de ce travail est de permettre l'analyse de "*To*", et la re-conception de ses côtés qui le nécessitent, à travers une bonne re-fabrication, pour permettre sa maintenabilité.

En basant sur un développement par patrons, avec un certain nombre d'autres techniques de conception (re-conception, re-fabrication, maintenabilité, réutilisabilité, …) on pourrait arriver à réaliser des *Frameworks* moins rigides et moins fragiles, en éliminant les dépendances imprévues entre les composants du système.

Les patrons fournissent un ensemble de techniques facilitant l'analyse, la conception et la réutilisation et la maintenabilité des *Frameworks*. Ils réduisent l'immobilité d'un système en permettant la création de modules réutilisables. La rigidité et la fragilité d'un *Framework* sont aussi atténuées du fait de leur utilisation.

Le but de notre travail ici, est de réaliser une analyse d'un *Framework* et d'améliorer les aspects conception et de réalisation, en utilisant les concepts de l'orienté objet, des patrons, et des *Frameworks*, tout en suivant une méthode de développement et de conception et de développement choisies (maintenabilité, ré-engineering et refactoring). Cela vient du fait que les bons *Frameworks* ont en toujours besoin d'améliorations.

Le succès d'un système d'agents dans le futur, en notre point de vue, dépend des principes de construction et de structuration et des méthodes de conception du logiciel.

Cependant le J2ME a des restrictions fortes, qui ne permettent pas de développer, comme avec le Java standard.

Pour faciliter la conception et éviter de tomber dans des problèmes de dépendances, "*TO*" a opté, dès sa création, et ce malgré les restrictions du J2me, pour une solution de conception orienté objet par patrons et *Frameworks*.

Mais tous les logiciels ont besoin d'évoluer et de s'améliorer, et en même temps d'intégrer les nouvelles techniques de développement ou de conception.

Dans le travail d'optimisation des côtés faibles de *"To"* qui nous a été demandé, nous avons, nous aussi, choisi d'adopter les techniques actuelles de maintenance, de re-fabrication et de re-conception.

Le domaine d'utilisation des téléphones portables prend un grand essor de nos jours [Mah01], et cela va en en crescendo, ce qui fait qu'il est temps de commencer à adapter toutes les techniques de conceptions classiques pour les modèles réduits.
Notre travail, vient comme un travail précurseur, dans le domaine de la maintenabilité aux moyens techniques de développements « lourdes », dans le domaine des logiciels conçus pour téléphones portables.

I.1. PRÉREQUIS

Pour réaliser une maintenabilité d'un Framework sur téléphone portable, basée sur les patrons de conception, en utilisant une approche orienté objet, on a besoin d'avoir des connaissances dans les domaines suivants :

- Notions générales sur les Framework,
- Les patrons de conceptions,
- Le langage J2ME,
- Les Système Multi Agents,
- Les Agents,
- L'Orienté Objet,
- Les techniques de maintenabilité , telles que
- Le ré-engineering, et
- Le refactoring.

I.2. ETAT DE L'ART

Quelques simulations des agents sur téléphones portable ont été déjà développées. Leur problème principal est la technique de conception ou d'implémentation utilisées, comme la non-maîtrise des concepts de l'OOD.

Alors, qu'un certain nombre de *Frameworks* dans le domaine des agents sur ordinateur, existe comme AgentSpeak(L), ConGolog, AGENT0, 3APL, et AgentLight [URL].

Nous avons remarqué qu'il y'a très peu de travaux publiés dans le domaine de la conception d'agent (ou Systèmes Multi Agents) sur téléphones portables.

Les travaux portant sur le développement par patrons et *Frameworks* ou les travaux sur la re-fabrication, le re-conception, ou la maintenabilité sont eux quasiment inexistants.

Les travaux qui se rapprochent, mais très peu, du domaine de notre étude est celui, réalisé en 2002, par **Eva Indal** [Ind02],

Indal a développé un agent entre téléphone mobile et ordinateurs, son travail est peu développé quant à la conception, et son utilisation des moyens techniques évolués est très peu entreprise dans son mémoire, surtout en ce qui concerne l'absence de l'utilisation des concepts du développement orienté objets. On pourrait aussi dire, que la méthode de sérialisation qui a été utilisée est assez similaire avec ce qui existait déjà dans "*To*".

Un autre travail sur la sérialisation est celui de **Road**. Mais la méthode de sérialisation qui est décrite est la même que celle qui est utilisée dans To, sauf que dans To, elle a été appliquée dans la mobilité.

I.3. CONTENU DU MÉMOIRE

Le problème que nous sommes tenu de résoudre dans ce mémoire est de restructurer le *Framework*, qui s'appelle ***TO***, par les procédés d'ingénierie du logiciel, c'est-à-dire, la

reconception et la re-fabrication (refactoring) pour le rendre plus maintenable et faciliter , par delà, sa réutilisation.

L'organisation de principaux chapitres dans ce mémoire peut être brièvement résumée comme suit :

Le deuxième chapitre expose brièvement les notions théoriques utilisées pour réaliser ce mémoire. Ce dernière est divisée en 12 chapitres ; Tel que, le sous-chapitre II.1 définit le langage de programmation pour réaliser ce travail (J2ME). Sous-chapitre II.2 des notions sur les *Frameworks* sont introduit, le chapitre qui suit c'est des notions sur les agents et les SMA (System Multi Agent). L'objectif de ce chapitre est de présenter les agents, leurs caractéristiques, les avantages de leur utilisation et leurs applications. I1 définit les caractéristiques principales des agents dans un *SMA*. Nous étudions particulièrement les caractéristiques des agents mobiles.

Les sous-chapitres (II.5, II.6, II.7) concernant les notions de base qui vont être utilisées dans ce mémoire (sérialisation, sécurité, test logiciel).

Les sous-chapitres (II.8, II.9, II.10) trait les techniques de développement de logiciels ; qu'on va suivre dans notre mémoire (maintenabilité, re-fabricatuion, re-conception).

Également, les sous-chapitres (II.11, II.12) consacrent aux applications Web (*Servlet, JDBC, XML*).

Remarque : du fait de la consistance du mémoire, nous avons dû déplacer physiquement ce chapitre(II) en Annexe. Le lecteur est avisé qu'il n'y a aucun changement concernant la numérotation.

Le chapitre III sera consacrée au travail effectué (re-conception du ***TO***). Cette partie est également divisée en 6 chapitres :

Tout d'abord nous allons, dans le chapitre III.1, étudier et analyser le logiciel *Framework* ***To*** existant, pour faire ressortir tous ses côtés négatifs, ainsi que ceux positifs.

Ensuite, nous allons proposer les côtés à améliorer (côté mobilité). D'abord en exposant avec un certain détail comment a été réalisé le travail, à faire ressortir les faiblesses et insuffisances du ***To*** initial. Ensuite nous allons proposer notre solution (Sous-chapitre III.2 dans le même chapitre). Le sous-chapitre III.3 présente le côté la manière dont nous avons conçu la sécurité de ***To***.

On exposera, ensuite, dans le sous-chapitre III.4, le côté migration de la base de faits. Nous donnerons une solution basée sur la mobilité, des règles transformées sous la forme d'un schéma XML, puis leur migration à travers un serveur http et Une *Servlet* sous *Tomcat*, pour aboutir enfin à leur persistance, sous la forme d'un schéma relationnelle, dans une base de données sous le JDBC *PostgreSQL*.

Dans le sous-chapitre III.5, nous allons expliquer comment nous sommes arrivé à améliorer, et résoudre le problème de l'affichage des interfaces graphiques de ***To***, et par delà, comment nous avons amélioré ces dernières. Dans le sous-chapitre III.6 présente comme ***To*** a été testé

A travers, tout le mémoire, nous avions utilisé, quand c'était possible ou intéressant de faire, le langage de visualisation de la conception UML, pour représenter soit le côté statique, soit le côté dynamique d'une re-conception, re-fabrication, patrons, ou frameworks.

Le chapitre IV est consacré à la présentation de l'Application. Dans ce chapitre nous allons montrer les différentes évaluations, et tests réalisés pendant le travail de la re-conception ou de la re-fabrication.

A la fin, dans la conclusion, nous allons faire une évaluation de notre solution, et fournir des directives pour des travaux futurs.

CHAPITRE III

ANALYSE, CONCEPTION ET REALISATION

III.1 ETUDE ET PRESENTATION DU "To" ORIGINEL

1. Introduction:

Dans cette partie nous allons aborder notre réalisation en ce qui concerne les techniques du génie logiciel utilisées dans l'élaboration de l'optimisation demandée.

Les techniques d'optimisation utilisées sont basées sur des techniques de l'orientée objets, des patrons, des frameworks, de la re-conception (re-engineering), de la re-fabrication (refactoring), et de la maintenabilité des logiciels.

La re-conception, la re-fabrication, et la maintenabilité des logiciels sont toutes des méthodes itératives et incrémentales.

IL s'agit dans cette partie d'étudier, de décortiquer, d'analyser et de décomposer la conception à la base du logiciel *To* fourni, pour en faire ressortir les éléments forts et faibles de logiciel.

Les étapes de la démarche adoptée dans notre travail sont les suivantes :

- Analyse des documents fournis,
- Analyse du logiciel *To*,
- Prise de contacts avec les développeurs de *To*,
- Cerner les parties qui seront sujets éventuelles d'un re-factoring et d'un re-engineering,

- Réfléchir aux parties qui pourraient être sujets éventuels à la maintenabilité.

2. Présentation de *To*:

To (True Object, ou autre ...) est un Framework multi-agents à base de patrons, qui permet de transformer n'importe quel objet (au sens de l'orienté objets) en un agent intelligent, sociable, et mobile sur téléphones portables.

To:

- est un Framework à base de patrons.
- est un Framework multi-agent développé pour téléphone portable
- est un système multi agents intelligents, réactifs, sociables, et mobiles,
- utilise les technologies des agents mobiles pour la migration;
- utilise les technologies des systèmes experts pour instancier son moteur de raisonnement,
- pourrait servir de base pour d'autres développeurs.

On utilise les méthodes de développement re-engineering pour décrire ***To***

On dit conception inverse ou reverse-engineering car on va découvrir la conception de l'application à partir des classes qu'elle contient

3. Description du framework To

Le Framework To est organisé en 9 packages et une *MIDlet* d'affichage ; qui sont :

1. **agent :** composé de 5 classes et interfaces de bases ;
2. **controller :** contient une seule classe ;
3. **expertSystem :** 15 classes;
4. **message :** 14 classes ;
5. **mobility :** 6 classes ;
6. **test :** 15 classes ;

7. **to :** 8 classes ;
8. **util :** 19 classes ;
9. **views :** 11 classes ;
10. **la MIDlet :** la classe *ToMIDlet* est la classe principale ;

On peut décrire ces différents packages par un diagramme de déploiement; comme suit:

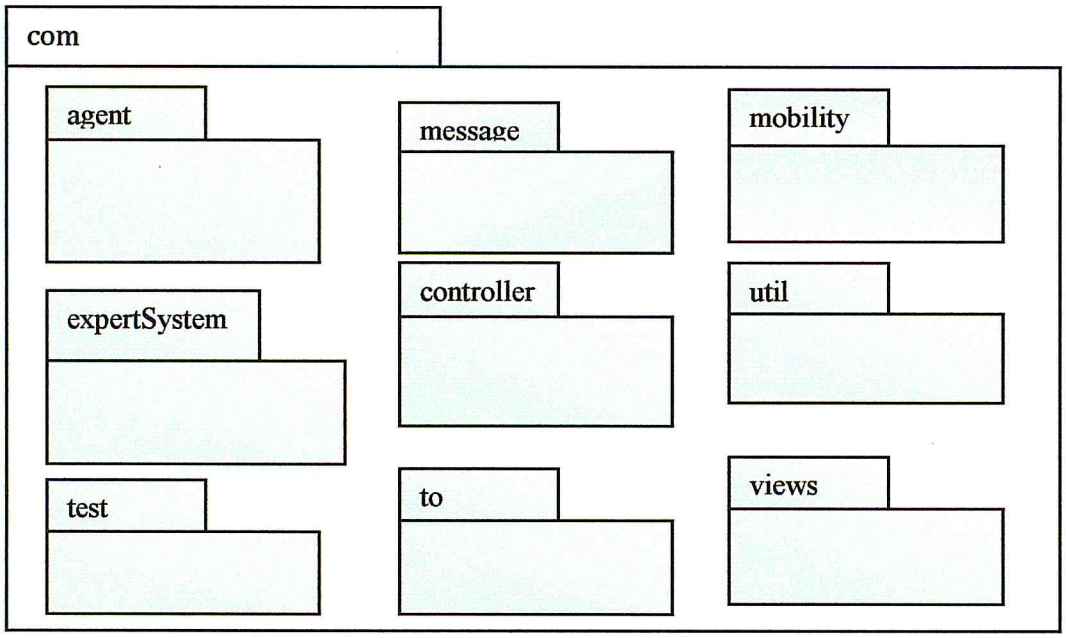


Figure3.1 : diagramme de déploiement des packages de to.

On va détailler, par la suite, chacun des packages par un diagramme de classes, et faire ressortir ensuite les différents patrons utilisés dans chaque package.

3.1. Le package agent

Le package *agent*, dont le diagramme de classes est comme suit:

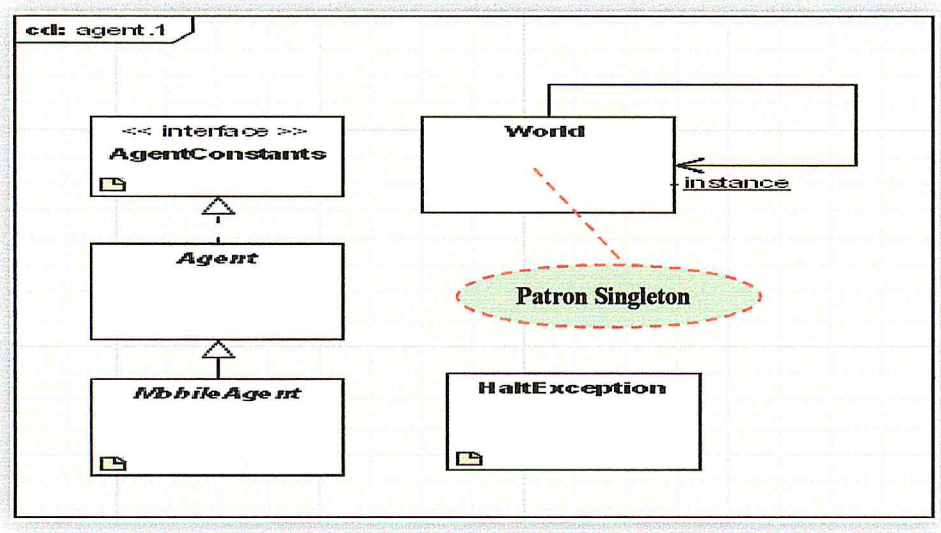


Figure3.2 : diagramme de classe montrant le package agent.

La classe *World* :

Comme on pourrait s'en apercevoir, cette classe joue un rôle très important dans le domaine de la conception de *To*. On pourrait dire que c'est la poste par laquelle passe tous les messages de communication entre les différents agents.

Elle est conçue comme un patron *Singleton*, pour la simple raison, qu'on n'a prévu qu'une seule poste de messages pour une même plateforme (ou un même environnement). Elle est à la base d'une caractéristique importante du domaine des agents, qui est la communication. Donc la classe *World* est à la base de la communication entre les *To*s
On peut dire donc, que pour tous les *To*s d'un même environnement, il ne doit y avoir qu'une seule poste utilisée pour poster des messages à destination d'autres Tos.
Cette poste, boite de messages ou boite à lettres est appelée *World*.

La classe *World* est conçue au moyen du patron *Observer*.

Comme on l'avait expliqué auparavant, ce patron permet d'avoir un système fortement découplé, donc c'est une approche bien plus intéressante que l'approche qui consiste à utiliser

une boîte à lettres, à la manière de la poste restante où sont stockés les messages à destination d'un agent qui en prend connaissance périodiquement.

La différence réside dans le fait que dans cette approche la notification du destinataire est faite par la boîte aux lettres elle-même, au moment de la réception d'un message.

L'avantage d'utiliser l'*Observer* est son couplage faible va nous donner des possibilités d'une véritable réutilisation.

La classe *Agent*:

C'est une classe abstraite qui a toute les caractéristiques d'un agent, intelligent, et sociable.

La classe *Agent* définit donc les principales caractéristiques d'un agent.

La classe *MobileAgent*:

C'est une classe abstraite qui hérite toutes les caractéristique d'un Agent, et en plus la caractéristique de mobilité, et cela en raison de son implémentation de *l'interface Migratable* qui se trouve dans le package *mobility*.

La classe *HaltException*: une spécialisation de la classe *Exception* de java.

3.2. Le package *mobility*

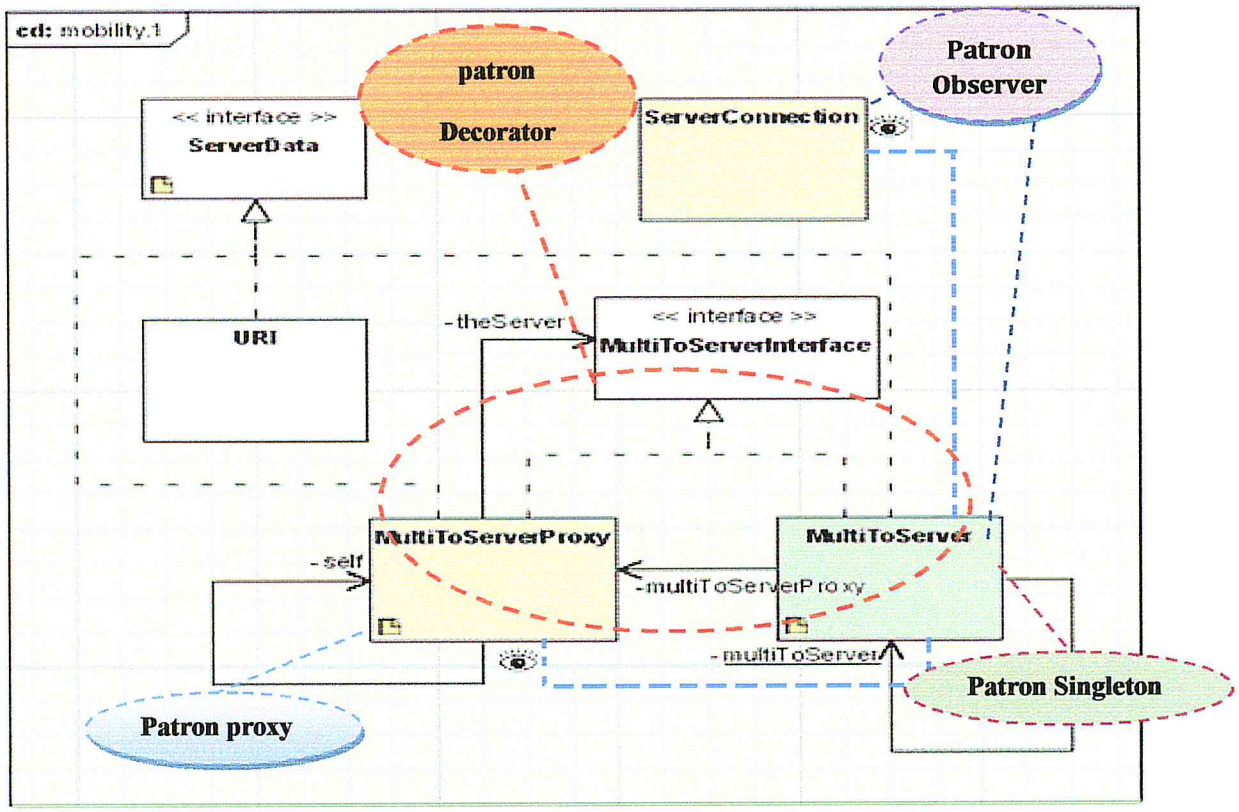


Figure3.3 : diagramme de classe présentant le package mobility.

Ce package englobe les classes qui rentrent dans la mobilité des messages et autres (base de faits, base de règle du raisonneur) qu'on déplace sur réseau, entre les environnements:

La classe *MultiToServer*:

C'est le serveur de l'application, il reçoit les requêtes à appliquer sur les agents, il tourne à l'attente d'une connexion d'un client, dès qu'il en reçoit, il notifie les classes *ServerConnexion*, et *MultiToServer*.

Elle est conçu comme un *Singleton* et un *Observable* en même temps.

La classe *MultiToServerProxy*:

Il joue le rôle d'un proxy. Il implémente la même interface que *MultiToServer*, et il a le même rôle que ce dernier.

En tant qu'observer de *MultiToServer*, dans son *update()*, il va va profiter de stocker les connexions dans une liste, qu'il mettra à la disposition du Serveur pour un quelconque traitement. (c'est une sorte de secrétaire général).

la classe *ServerConnexion*:

Cette classe permet de traiter chaque connexion réseau, dans un thread unique.
Elle s'occupe de lire le *Stream* d'entrée (flux), d'en créer avec un Receiver, et de notifier le *World* et l'*Environnement*, dans le cas où un message vient d'arriver.

3.3. Le package *expertSystem*:

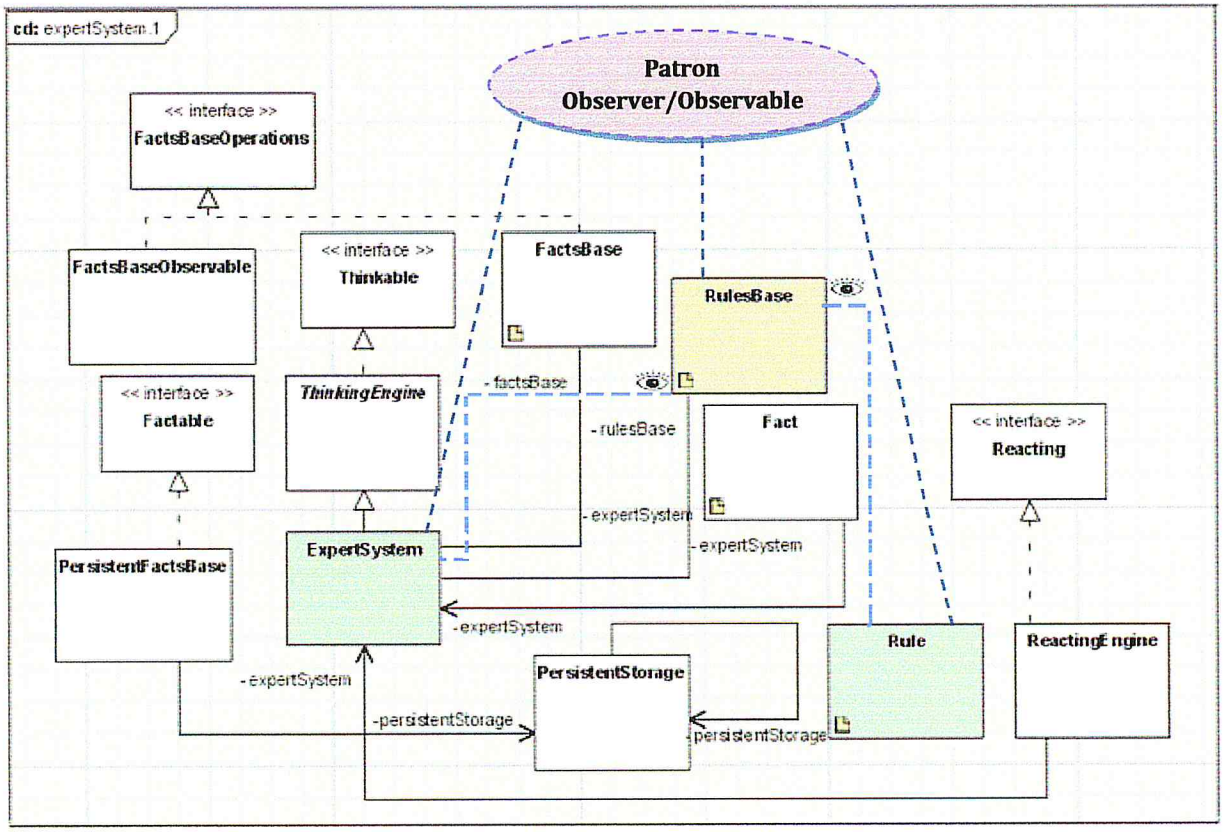
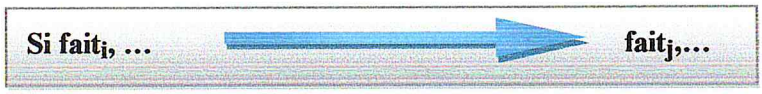


Figure3.4 : diagramme de classe montrant le package expertSystem.

Le raisonneur conçu, pour attacher un côté intelligence à la base de To.

Dans la première application du framework To, ce raisonneur a été instancié par un raisonneur, moteur d'inférences par chaînage avant, à base de règles de production, du type :



La classe *Rule* : C'est la classe qui modélise une règle de production.

La classe *Fact* : C'est la classe qui définit un fait.

La classe *ExpertSystem* :

C'est le système expert d'un agent, il englobe une base de faits *FactsBase* et une base de règles, et dès qu'on opère une mise à jour dans la base de règles *RulesBase*, ou dans la base de faits (the working memory) FactsBase le Framework, grâce à l'*Observer*, déclenchera automatiquement le moteur d'inférence.

FactsBase et ***FactsRule*** sont, respectivement, la base de faits et base de règles.

Elles sont conçues, comme de conteneurs de faits et de règles, respectivement, toujours à la base de la même *LinkedList*.

3.4. Le package *util*

Il englobe les outils utilisés pour réaliser, implémentationnellement, certains côtés du *Framework* *To*.

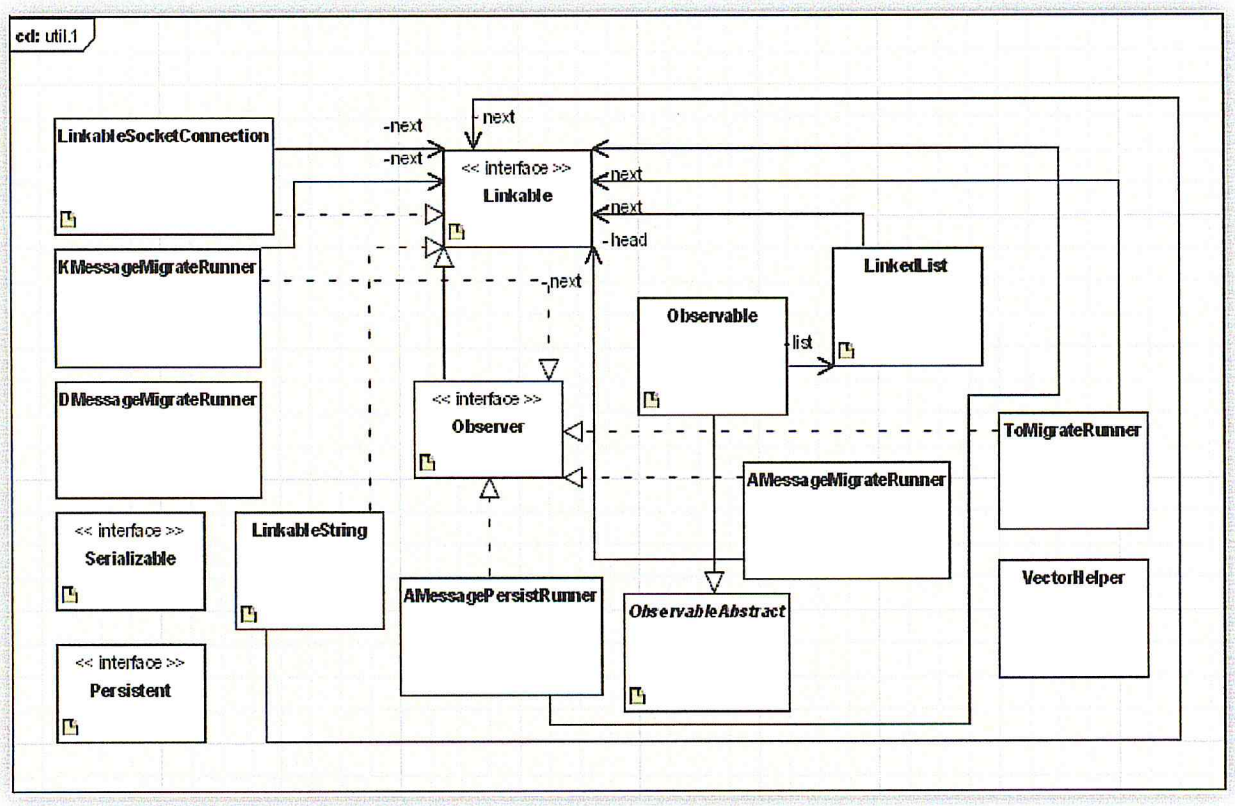


Figure 3.5 : diagramme de classe présentant le package util.

La classe *Observable* :

Dans cette section on va détailler l'architecture du patron[1] Observer de java.

La classe *Observable* doit avoir une référence sur une liste d'*Observers*. Lors d'une notification cette dernière est parcourue pour envoyer à chaque *Observer* la méthode *update()*.

Dans le cas de ***To*** cette liste est une liste de *Linkable*, elle peut contenir tout élément qui implémente qui implémente *LinkedList*.

L'*Observable* hérite de la classe abstraite *ObservableAbstract* (voir Figure 3.5). L'utilisation d'une classe générale *ObservableAbstract* permet aux utilisateurs d'implanter l'*Observable* à leur manière, ils peuvent par exemple utiliser un tableau ou une autre structure

[1] Voir sous-chapitre II.3.

de données au lieu de cette liste. On peut remarquer qu'on peut utiliser une interface au lieu d'une classe abstraite, mais dans ***To*** ce ne f^t pas le cas, car toutes ses méthodes sont *public* et la méthode *notifyObservers (Observable)* doit être *protected*.

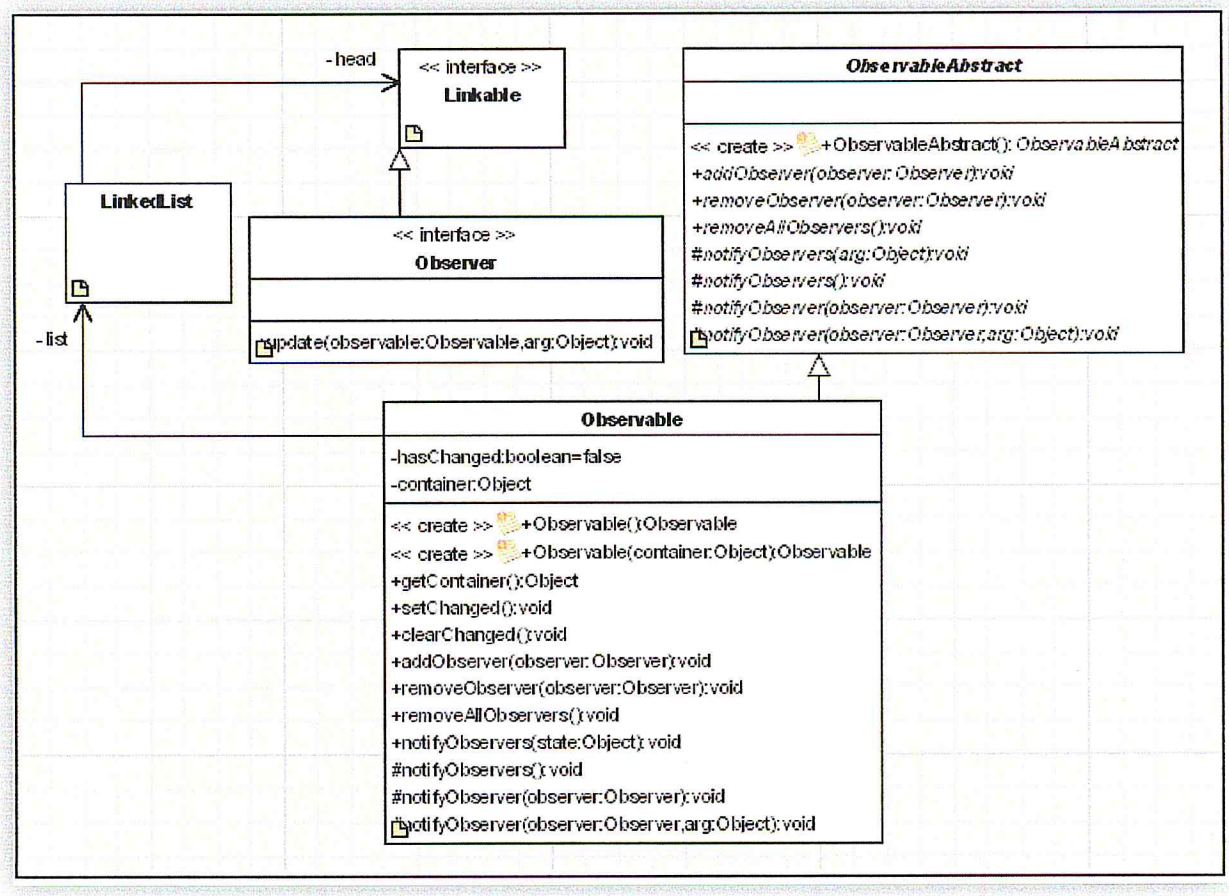


Figure3.6 : diagramme de classe présentant le patron Observer/Observable.

La classe *LinkedList* :

To a crée son propre conteneur unique et général, il est basé sur une liste chaînée.

Elle est utilisé par le patron *Observer/Observable* comme cela est expliqué plus haut.

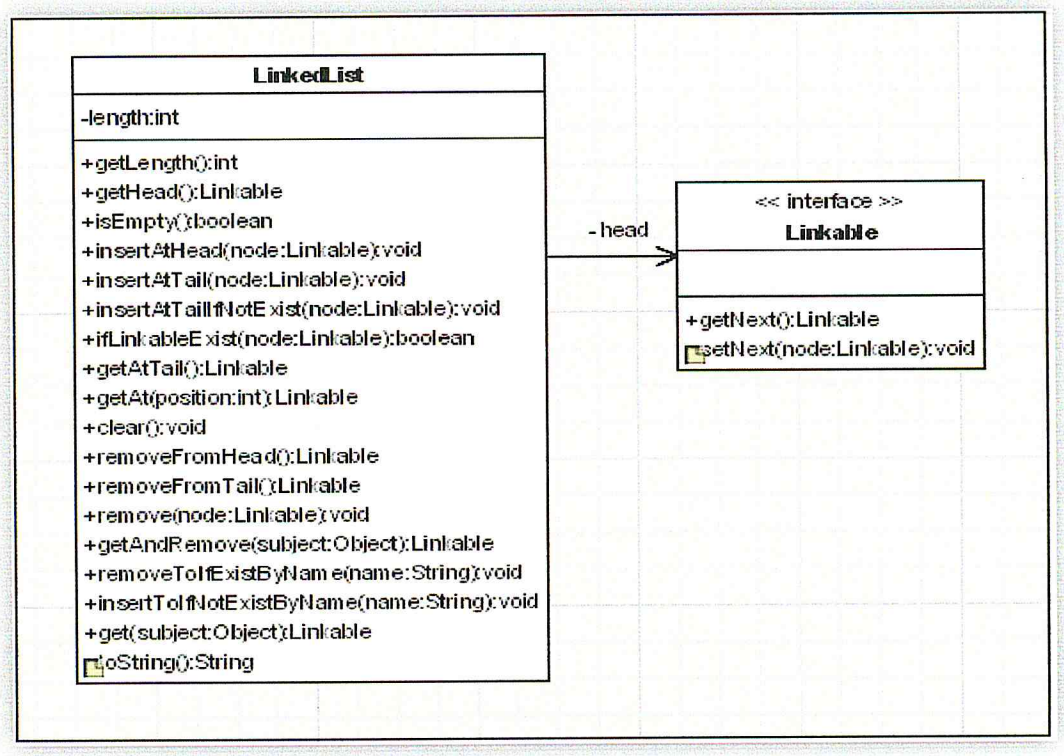


Figure3.7 : diagramme de classe décrivant la **LinkedList** *utilisée dans* **To.**

3.4. Le package *to*

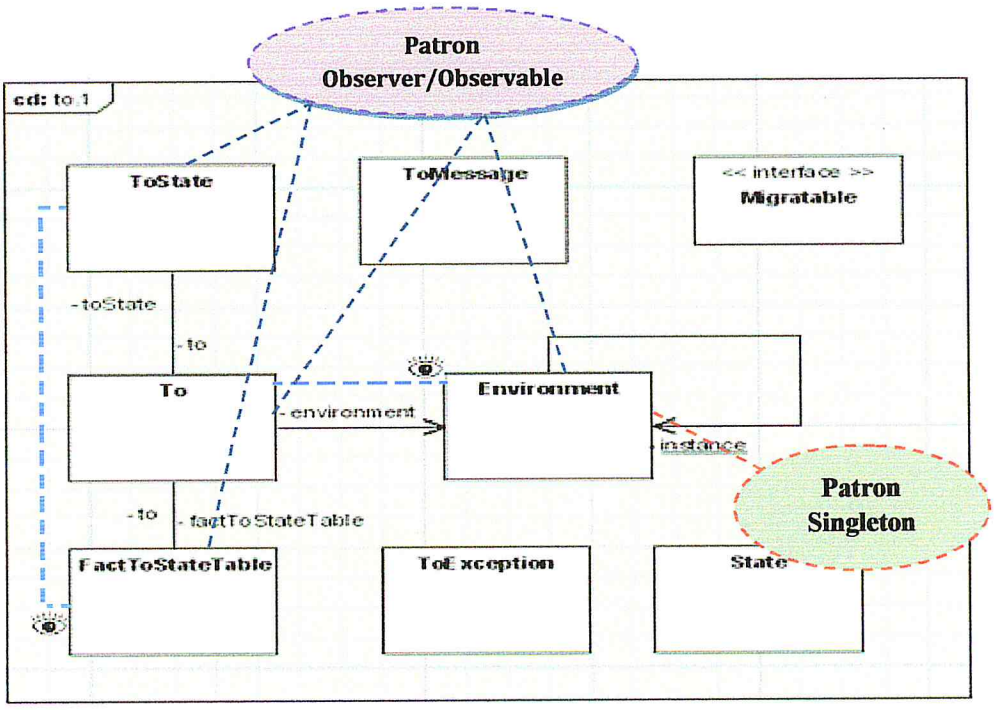


Figure 3.8 : diagramme de classe présentant le package to.

La classe *Environment* :

C'est le moyen, par lequel, les *To*s vivent et coopèrent ensembles.
Il est basé sur un conteneur, une *LinkedList* où en peut insérer les *To*s dès leur création, et une *Hashtable* faisant la correspondance entre un nom de *To* et un *To* de l'environnement
C'est l'*Environment* qui crée le système de coopération multi *To*s (SMA).

La classe *To*:

C'est la classe qui décrit les agents To, c'est un *Agent* intelligent et mobile, il hérite la classe *Agent* et implémente les interfaces *Thinkable, et Methodable,* pour lui adjoindre le côté intelligence.

.

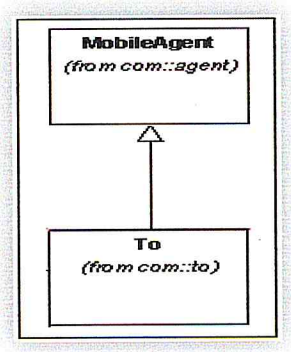


Figure 3.9 : To* « est un » *mobile agent

La classe *ToState* :

C'est le tableau de bord du To, c'est une sorte de conteneur indexé (*Hashtable*).
La *Hashtable* fait la correspondance, entre le nom de l'état et l'état du *To*.
Il comporte tout les états émotionnels, psychologiques, physiques, physiologiques et autres du To, comme:

"estMalade", false
"aTropMangé", true
"estFatigué", false
"estFort", true
"eatIntegerString", true
,...

Elle est Observer de *FactsBase* de *ExpertSystem* du *To*, et *Observable* de *FactToStateTable*.

La classe *State* :

C'est une sorte de fait, représentant l'état émotionnel, psychologique, et physique interne d'un *To*.

Elle est *Observable* de *FactsBase* pour lui permettre de se mettre à jour par rapport à ce qu'il y'a dans la base de faits.

Un *Fact* peut agir sur un *State*, et vice versa.

L'interface Migratable:

C'est elle qui contient la fonction spéciale de migration des messages et des agents;

3.5. Le package *message*:

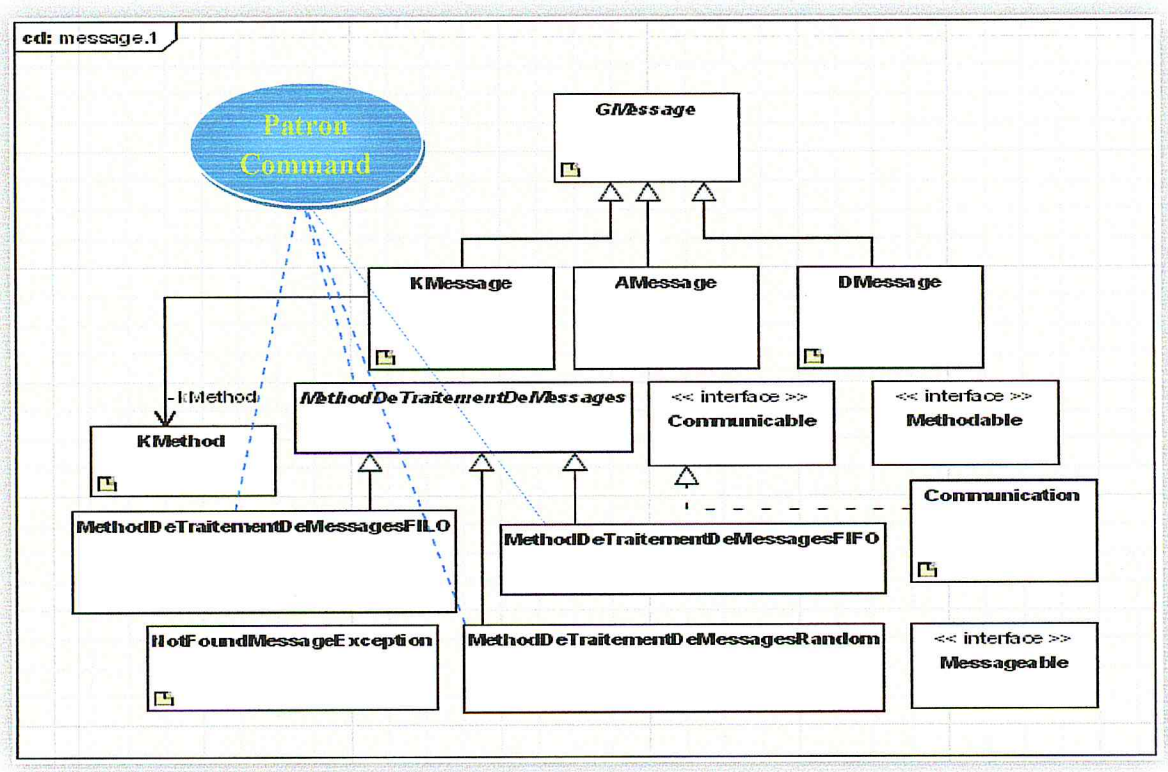


Figure3.10 : diagramme de classe de package message.

Ce package englobe les type des messages qui sont utilisés dans le framework *To*.

To a choisi l'échange des messages comme le principe de la communication pour les raisons suivants.

Les avantages de la communication basée sur l'échange des messages sont:

- ⇨ La communication d'agent (partage de la connaissance) doit être encapsulée comme il ne peut pas faire partie d'une langue,
- ⇨ Un seul type de message doit être compris par tous les types d'agents (i.e. une façon standard de communiquer),
- ⇨ Le message doit être de taille minimal pour réduire la charge du réseau,
- ⇨ Le message peut être de formats différents.

Si un objet envoie un message à un autre objet, le receveur n' a aucun contrôle sur l'exécution de la méthode correspondante; la méthode va toujours s'exécuter. Ce contrôle est fourni par le langage de base, et non pas par l'objet.

Avec les agents, la communication est vue comme une demande par un agent à un autre, de faire un certain traitement, s'il le désire. L'agent récepteur peut exécuter n'importe quelle action à la réception de la demande, qui va être entièrement sous son contrôle.

Mazari [Maz09] a jugé utile d'abstraire un message par un message **GMessage** général, dont trois autres classes de messages vont découler :

- **les messages *KMessage* :** demandant au destinataire l'exécution d'une de ses méthodes,
- **les messages *DMessage* :** pour la destruction du *To* destinataire,
- **les messages *AMessage* :** pour la migration d'un *To* vers un environnement hôte.

Un message ***GMessage*** est conçu comme un message qui pourrait migrer.

Les messages ***KMessage*** a échangé entre agents pour exécuter une méthode contiennent:

- les identités de l'émetteur (source du message) et
- du receveur (destination) du message,
- la méthode pour exécuter par l'agent receveur, et
- les arguments de la méthode.

L'agent ***To*** réagi à un ***KMessage*** avec son propre comportement déterminé après que le message soit reçu.

La classe *MethodDeTraitementDeMessage*:

- **Traitement de Messages**

Chaque agent TO a un conteneur pour stocker les messages reçus.

Les messages sont traités d'après *MethodDeTraitementDeMessages* et sont utilisés pour activer des capacités ou exécuter un comportement spécifique de l'agent.

Cette méthode de traitement a été conçue au moyen du patron *Command* de telle sorte qu'on pourrait avoir plusieurs sortes de méthodes de traitement de messages, et cela selon l'ordre d'arrivé des messages :

- l'ordre premier arrivé, dernier traité, FILO,
- l'ordre premier arrivé, premier traité, FIFO, ou
- un ordre aléatoire, le traitement se fait selon une certaine stratégie.

Pour résumé :

Nous allons donner un exemple d'échange de message entre deux agents.

l'agent *To1* veux envoyer un message *Message3* a l'agent *To2*.

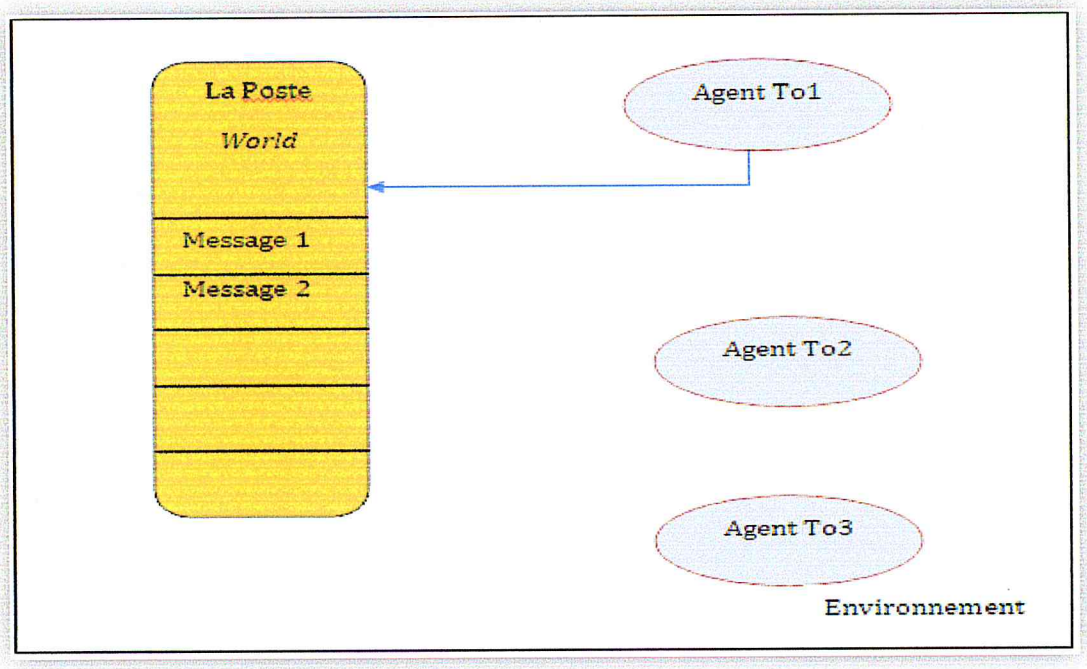
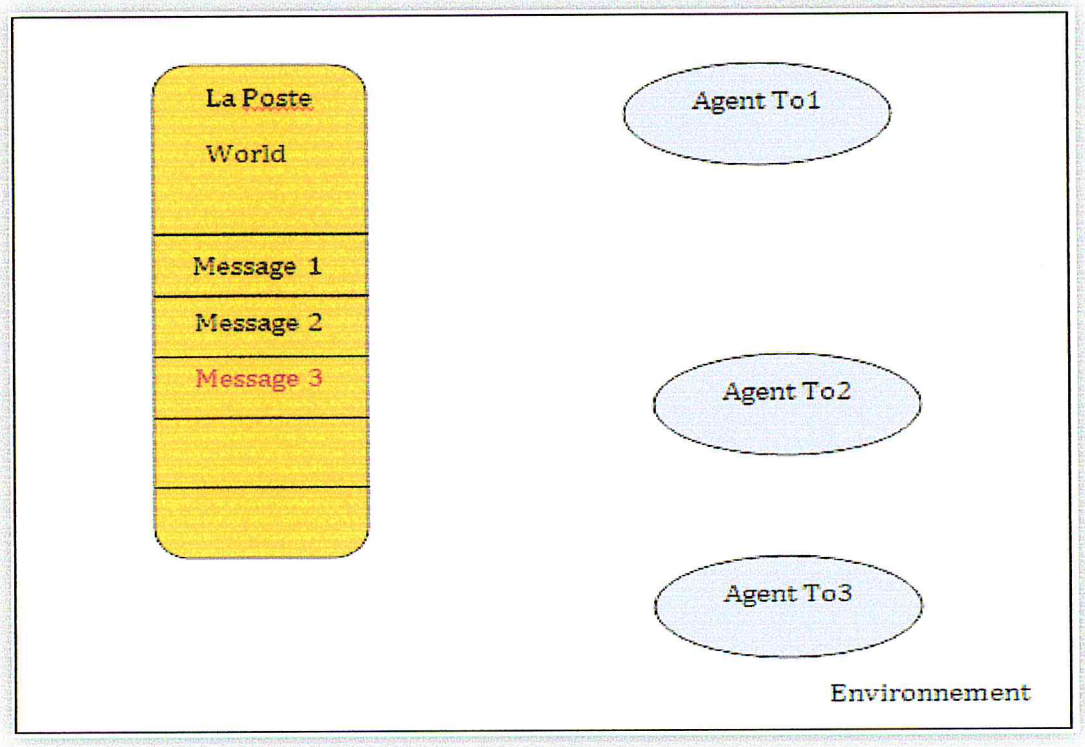


Figure3.11 : Messages envoyés par les* Tos, *à la poste* World *[Maz09] :

Dans première étape l'agent To1 envoie un message vers le *World* (La poste) pour que cette dernière s'occupe de délivrer ce message au *To* destinataire.



Quand le message arrive à la poste, elle le stocke chez elle et notifie les autres agents qu'il y a un message qui arrive. Pour concevoir cette solution Mazari a utilisé les *To*s comme des *Observer* de *World*.

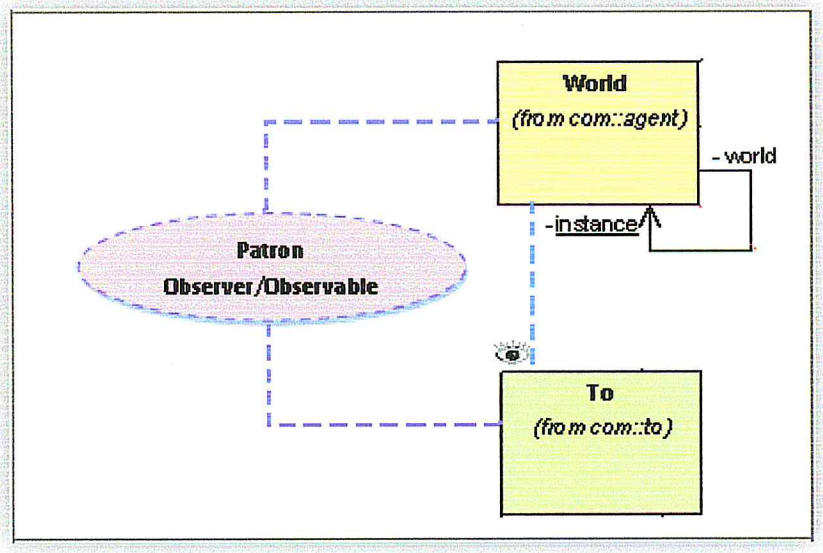
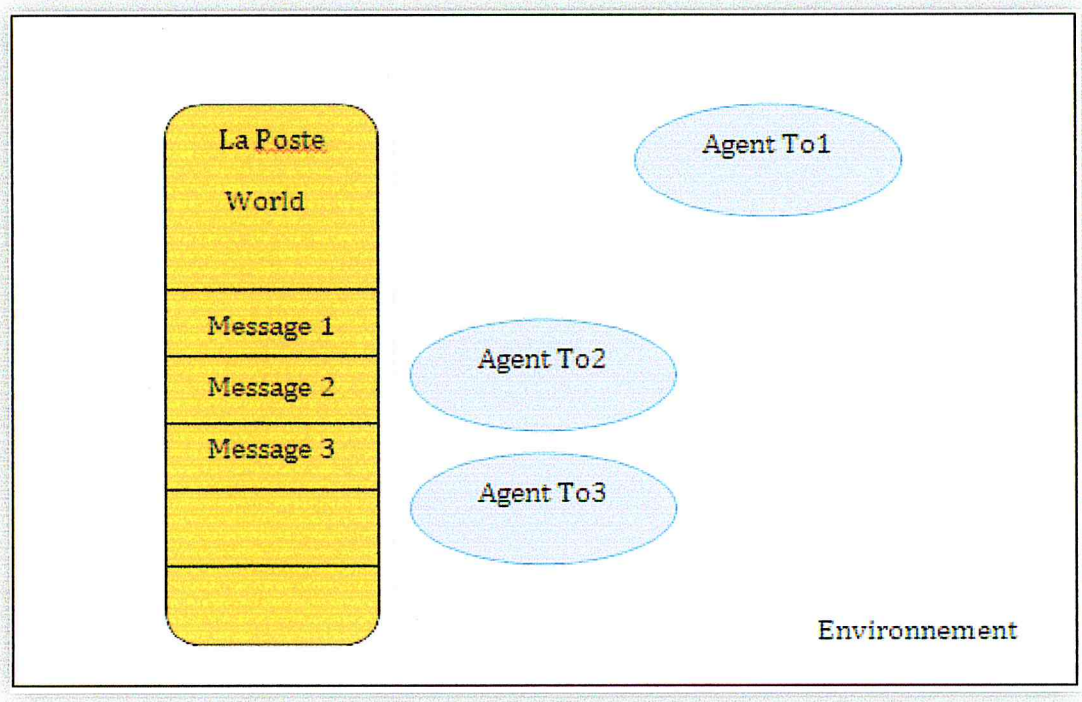
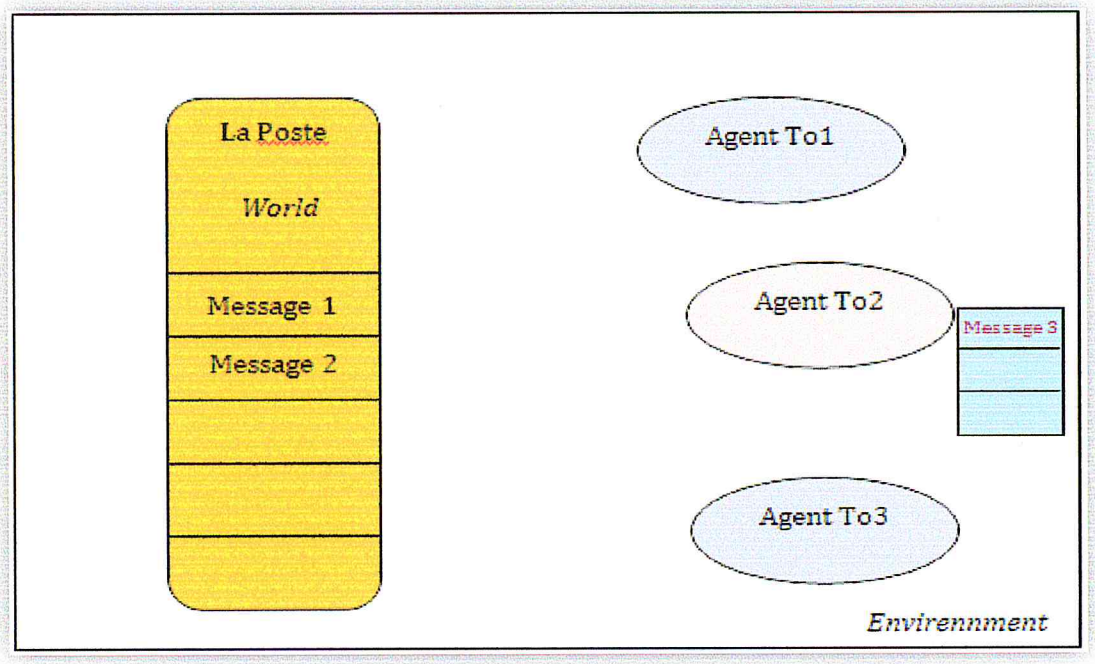


Figure3.12 : To et World sont reliés par patron Observer/Observable.



Chaque agent doit vérifier la liste des messages de *World*, si un des messages lui est destiné.



Après que l'agent retire le message il le stocke dans une liste propre pour qu'il puisse le traiter ultérieurement.

En peut voir que l'envoi des messages est conçu de manière asynchrone, ce qui nous donne un grand avantage, dans le cas où le To destinataire est occupé, l'agent émetteur n'attend pas que le receveur soit libre pour lui envoyer le message.

Mais l'inconvénient de cette solution apparaît lorsque le *World* a plusieurs *To*s, et que tous ces *To*s doivent vérifier la liste de messages quand un message arrive au *World*, ce que rend le module de communication un peu lourd.

3.6. Le package *views*

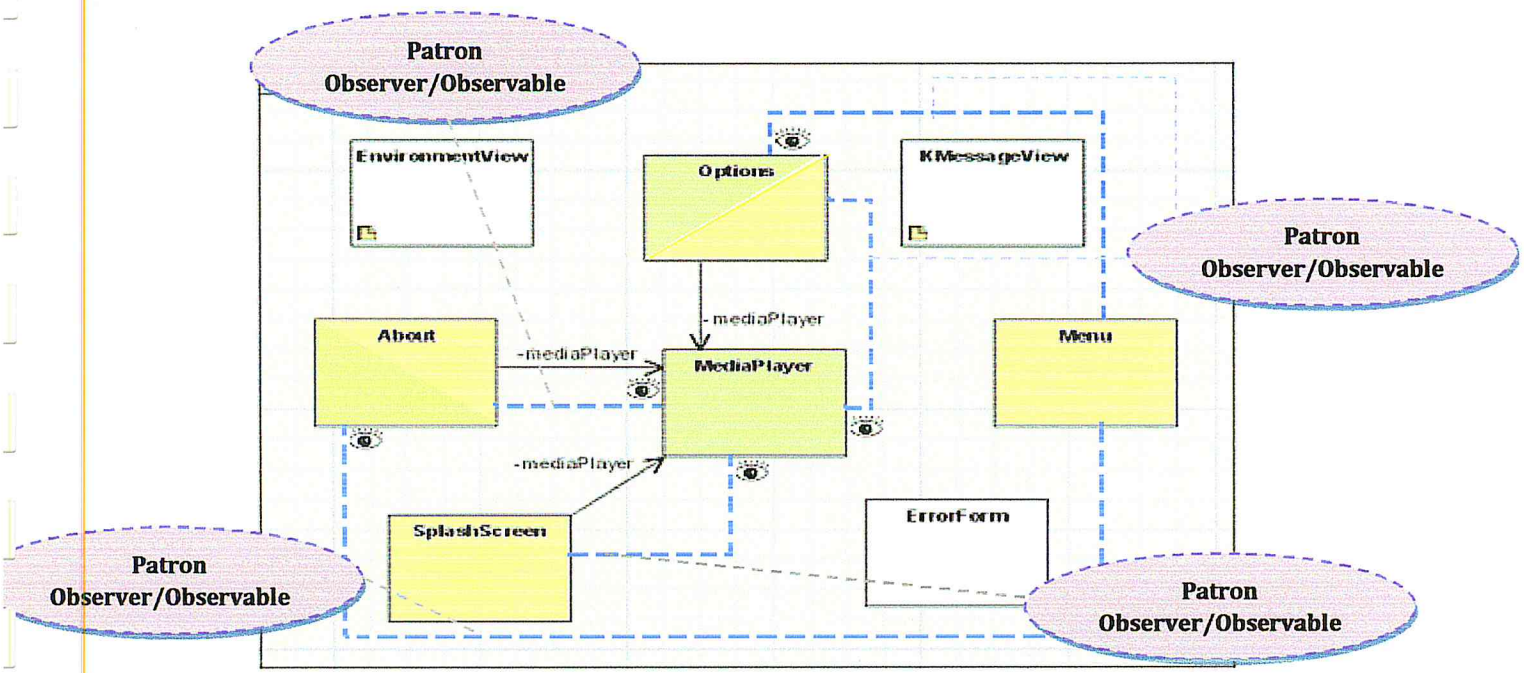


Figure3.13 : diagramme de classe présentant le package views.

Se sont les vues d'affichage, au sens MVC du terme, comme on en voit sur l'écran d'un téléphone portable.

Elles indiquent, par affichage, qu'un événement vient de se produire, et peuvent donner des informations supplémentaires sur cet événement.

La classe *MediaPlayer*:

C'est une classe qui définit une sorte de vue musicale pour chaque notification d'un *Observable*s.

Cette belle idée est celle de Mazari.

La classe *SplashScreen*:

C'est une sorte d'écran de bienvenue, de démarrage de l'application.

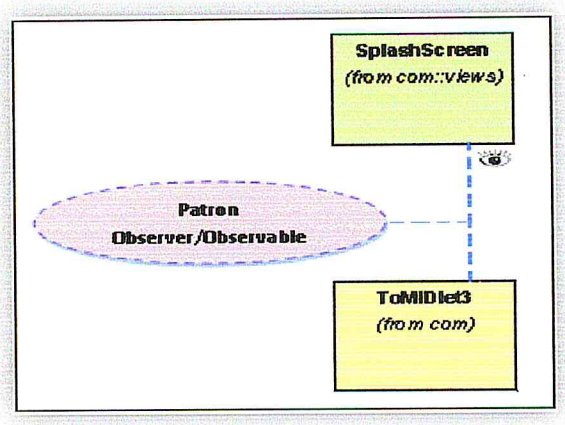


Figure3.14 : ToMIDlet et SplashScreen sont reliés par patron Observer.

La classe *Menu*:

C'est une liste d'affichage, qui nous permet de choisir l'un de ses éléments, elle est *Observer* de *SplashScreen*.

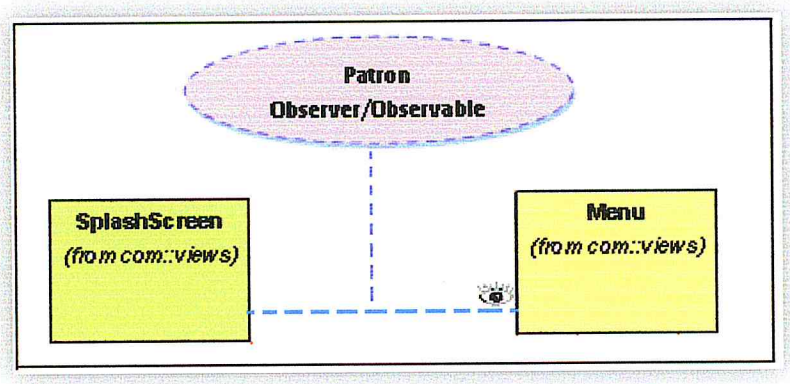


Figure3.15 : SplashScreen et Menu sont reliés par patron Observer/Observable.

La classe *About*:

Il joue le rôle d'*Observer* de *Menu*, dès quand clique sur l'élément ***about*** de la liste *Menu*, on récupère une vue, nous donnant des information sur le framework et son concepteur..

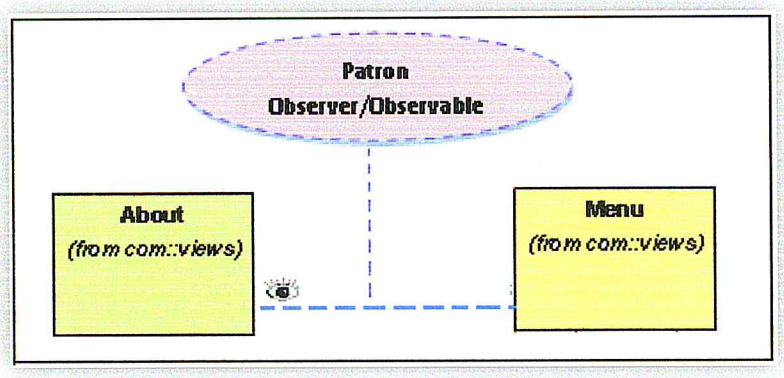


Figure3.16 : About et Menu sont reliés par patron Observer/Observable.

La classe *Options*:

C'est une *Form*, qui est notifiée par *Menu*. Il est conçu, mais pas encore utilisé, pour choisir certaines options du logiciel.

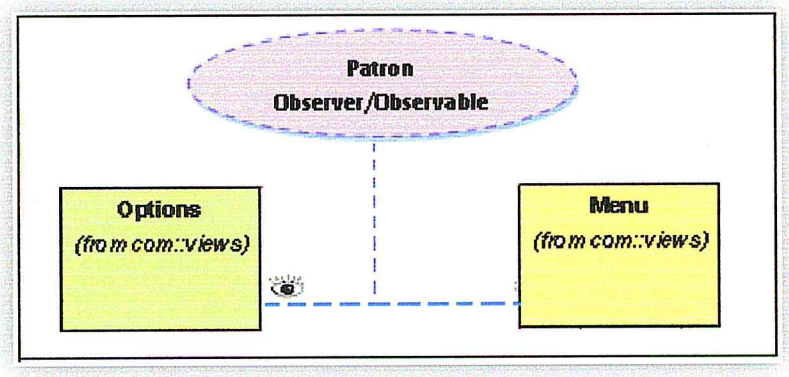


Figure3.17: Options et Menu sont reliés par patron Observer/Observable.

La classe *EnvironmentView* :

C'est une sorte de *Form*, qui nous affiche si un agent est ajouté à l'environnement. La création (ou destruction) d'un agent To notifie l'*Environment* pour qu'il mette à jour sa liste. Ce dernier va à son tour notifier la *Form EnvironmentView* pour qu'elle opère un certain affichage dans la fenêtre.

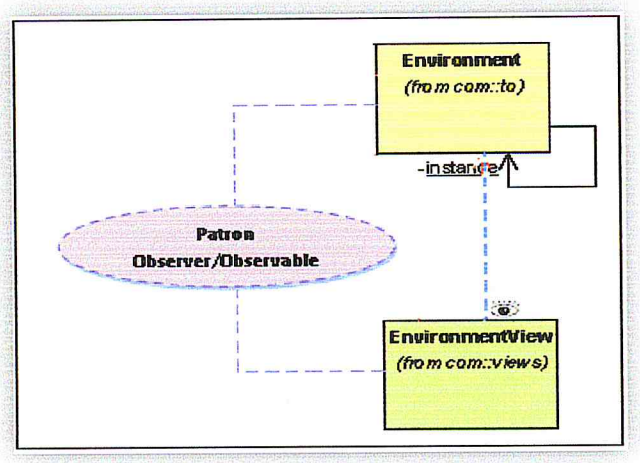


Figure3.18 : Environment et EnvironmentView sont reliés par patron Observer/Observable.

La classe *KMessageView* :

C'est une sorte de *Form* (fenêtre), qui va nous informer qu'un événement ayant un lien avec un KMessage vient de se réaliser, comme par exemple l'envoi ou l'arrivée d'un KMessage sur l'écran du téléphone portable.

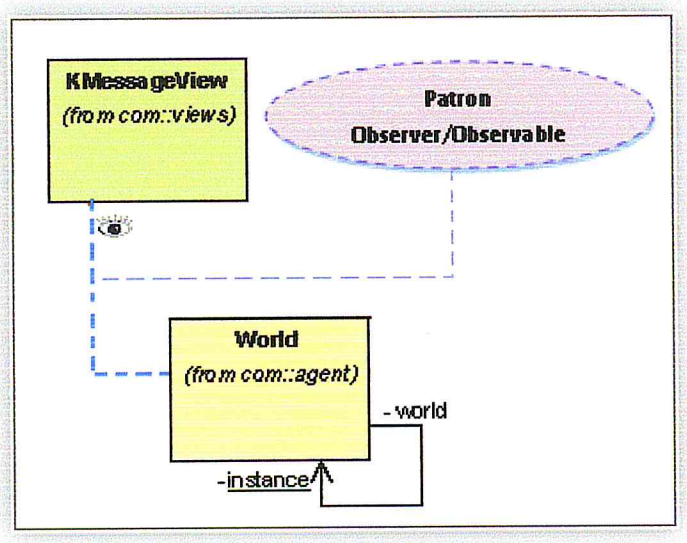


Figure3.19 : la relation entre le World et le KMessageView.

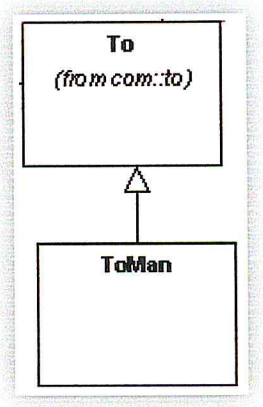


Figure3.21 : ToMan « est un » To.

L'interface *Maneable* :

C'est l'interface du comportement des *Man*, comme méthode *eat()*, et *isIll()*.

La classe *Food* :

Classe créée dans le besoin de faire appliquer la méthode *eat()* de *Maneable*.

La classe *Banana* :

C'est une classe hérité de la classe Food, elle désigne le type de nourriture qu'un *To* peut manger.

La classe *ToException* :

une classe d'exceptions particulière.

Remarque :

On remarque que toutes les classes sont conçues au moyen de patrons, comme :

- l'*Observer/Observable,*
- le *Command,*
- le *Singleton,*
- le *Decorator,*
- le *Proxy.*

Cela prouve la force du framework réalisé.

3.7. Le package *test*

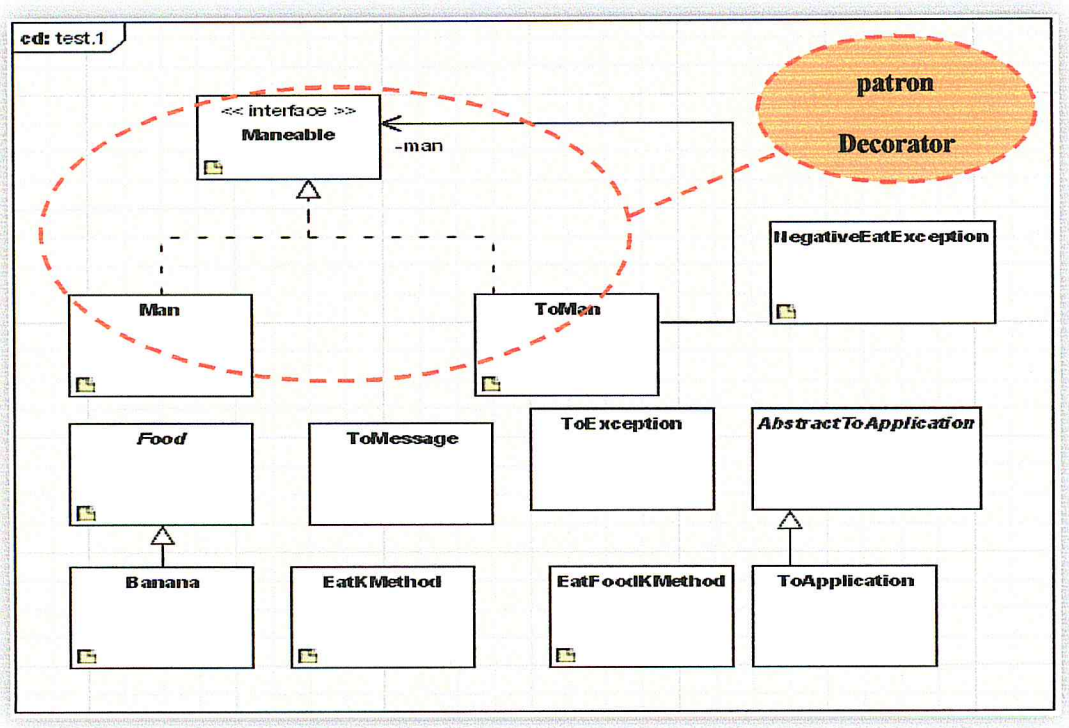


Figure3.20: diagramme de classe présentant le package test.

La classe *ToApplication*:

C'est la classe principale de l'application, c'est le HotSpot qui sera implémenté par l'application du Framework

La classe *ToMan*:

Cette classe est un ***To***, qui décore un *Maneable*; c.-à-d. c'est un agent, il hérite toute les caractéristiques d'un agent; mais il implémente la l'interface *Maneable*, pour que cet agent puisse avoir un comportement de base d'un être humain.

C'est conçu au moyen du *patron Decorator,* qui va décorer un Man, et dons son comportement de *Maneable*, pour lui attacher le côté « exécution de comportement raisonné ou intelligent »

III.2 REFABRICATION (REFACTORING) DE LA MOBILITE de TO

1. INTRODUCTION

Dans cette partie du mémoire, on va décrire le processus que nous avons proposé, pour la re-conception du pseudo mobilité par la re-fabrication de la sérialisation.

Pour arriver analyser le *Framework To*, et arriver à faire ressortir les champs qui vont être sujet à notre optimisation, c'est-à-dire soit à une refabrication, soit à une re-conception, nous avons, principalement, adopté trois méthodes :

La première méthode c'était la consultation du mémoire des mémoires des ex-chercheurs de l'équipe GLODOO (LRDSI), et principalement celui du développeur de To, la deuxième c'était à travers la collaboration avec les chercheurs de l'équipe GLODOO, et en troisième lieu à travers la méthode de la conception inverse (*reverse engineering).*

2. Notre solution (re-fabrication)

La sérialisation comme elle est définie dans les sections précédentes est le processus de conversion de l'objet qu'on voudrait envoyer sur réseau en objets élémentaires correspondants à ceux demandés par les flux (streams) de base de la communication réseau proposés par java.

La solution proposée par R. Mazari [Maz09], nous semble être re-fabricable, en vue de deux buts distincts .

Le premier consiste, en son application à des objets plus complexes, et le deuxième est un but de maintenabilité, c'est-à-dire qu'on vise, comme cela avait été initié, par R. Mazari [Maz09], à la réutilisation à travers des patrons de conception ou si possible, un framework à base de patrons.

L'idée de base de notre travail est basée sur la remarque que le comportement de sérialisation n'est pas un comportements des objets à sérialiser, mais plutôt, des flux relatifs au objets à sérialiser.

Cette remarque, nous a principalement menée, à ce qu'on devrait concevoir comme dans le cas du java standard un flux, dont le comportement est base des deux méthodes :

void writeObject (KSerializable object), dont le paramètre est le *KSerializable* objet à sérialiser sur le flux que nous envoyons sur le réseau, et

void readObject(), qui récupère du stream spécial de l'*Object* qui a été sérialisé et qui dérive de *ObjectInputStream* l'*Object* construit à partir de son flux de désérialisation que nous récupérions du réseau.

La solution qui a été proposée dans ***To***, c'est d'attacher à chaque classe d'objets, une méthode spéciale dont le rôle est de décomposer l'objet en objets primitifs pouvant être envoyés sur les flux réseau en java.

Ces méthodes consistaient principalement à décomposer l'objet dans un tableau de *byte*, que nous envoyons à travers un *OutputStream*.

Les faiblesses de cette méthode, c'est qu'elle n'est pas directement applicable aux objets complexes, c'est-à-dire des objets, dont la structure comporte des éléments qui peuvent être eux-mêmes des objets complexes, et ainsi de suite.

Et, d'ailleurs, dans *To*, il n'y a pas eu de cas d'application de la sérialisation à ce genre d'objets.

Donc, la base de cette opération de re-fabrication est d'augmenter les *OutputStream*s fournis primitivement par java j2me, et qui ne permettent d'écrire que des bytes, par un flux général, un *ObjectOutputStream*, qui nous permettra d'écrire des *Object*s, comme le montre la figure suivante :

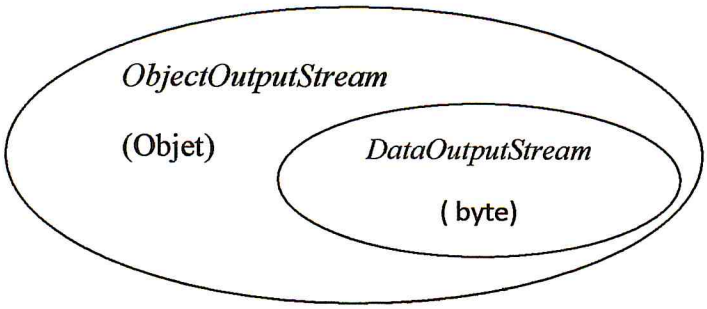


Figure 3.22 : la couche ObjectOutputStream englobe le DataOutputStream

La sérialisation qu'on voudrait développer en j2me, devrait permettre une utilisation identique de celle de java standard, en fournissant un squelette de base d'un que les développeurs pourrait utiliser, par la suite, pour sérialiser des objets sous j2me, comme cela se fait sous le java Standard.

Dans le processus de développement, qu'on a suivi, nous avons tout d'abord mis notre idée en pratique en confectionnant, toutes les classes réseau et autres pour tester la sérialisation d'une classe même simple. **Voir sous-chapitre III.6 Test**

3. Explication de notre solution :

L'objet qu'on veut envoyer dans le réseau il doit implémente l'interface *KSerializable* ; vide, qui au fait, n'est un englobant général de tous les objets variés qu'on voudrait sérialiser ;

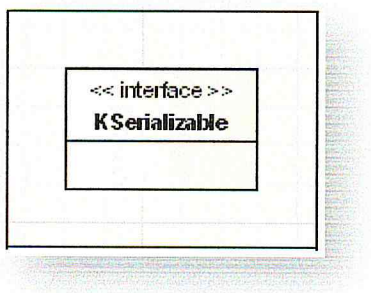


Figure3.23 : l'interface KSerializable.

Ensuite l'objet à sérialiser devrait implémenter cette interface ; comme le montre la figure suivante.

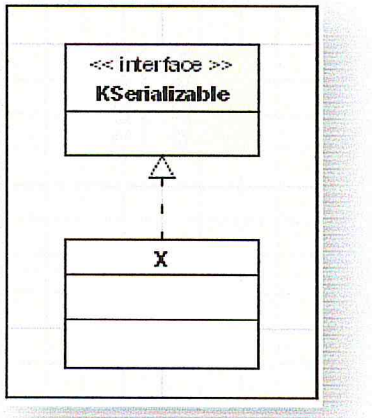


Figure 3.24 : exemple d'implémentation.

On a créé ensuite, les classes *ObjectOutputStream* et *ObjectInputStream ;* se sont les flux d'entrer et de sortie, pour la sérialisation des objets.

Les classes *ObjectOutputStream* et *ObjectInputStream comportent les méthodes :*

WriteObject(KSerializable), et

ReadObject()

Ensuite, il suffisait de créer des classes d' *ObjectOutputStream* et d' *ObjectInputStream*, particulières à chaque classe d'*Object*, qui dans la première solution, en vue de la confection d'un framework blanc, vont hériter de ces dernières classes.

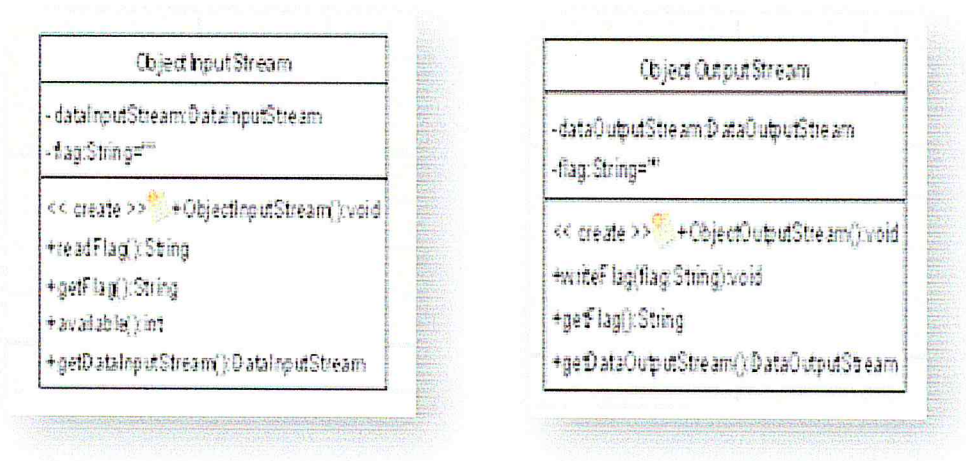


Figure3.25 : les Stream d'écriture, et de lecture.

les méthodes *writeFlag()*, et *readFlag()* permet d'écrire ou bien de lire un *Flag* d'un objet dans le *DataOutputStream*, et *DataInputStream*; ce qu'il faut pour chaque objet, on leur donne un flag.

Pour chaque objet qu'on veut l'envoyer, on doit leur créer des *Streams* de sorti et d'entré, qui ont les méthodes *writeObject()*, et *readObject()*, ces dernier doivent hériter les méthodes *writeFlag()*, et *readFlag()* du *ObjectOutputStream* et *ObjectInputStream*.

En premier lieu, on a testé cette technique à travers un simple objet de classe **X**, pour que nous puissions s'assurer se sa réalisabilité, c'est-à-dire pour nous rendre concrètement compte de la faisabilité de notre solution. **Voir sous-chapitre III.6 Test.**

Pratiquement, on voulait voir, lorsqu'on applique la structuration proposée, qu'est que qu'on allait récupérer de l'autre côté, après qu'on ait sérialisé et envoyé notre objet sur le réseau.

Le problème qui se posait à nous, c'était de pouvoir reconnaître les types particuliers des objets *KSerializable* qu'on reçoit à travers le flux de sortie.

Nous avons résolu, ce problème en adjoignant deux méthodes, *writeFlag()* et *readFlag()*, redéfinissables, dans les classes des flux particuliers aux vrais objets à sérialiser,.

Le rôle de ces méthodes est de positionner un indicateur (drapeau ou mouchard), qui est un indicateur de la classe de l'objet qu'on reçoit sur le flux.

Le deuxième problème qui s'était posé à nous, est un problème de réutilisabilité ou de confectionnabilité du framework.

Ce problème consiste en comment, incorporer dans le code le code de lecture particulier à chaque type d'objets, sans passer pour des buts de réutilisabilité, par des structures de *case*.

La solution que nous avons retenue, est d'utiliser un patron qui nous assure le moins de couplage possible, et nous avons opter pour des raisons techniques (c'est-à-dire de maîtrise et de réalisabilé), et d'élégance du code pour le patron *Observer*.

L'idée est la suivante, il s'agit d'adjoindre un *Reader Observable* du *Receiver*.

Le rôle du *Reader* est, qu'une fois qu'il est alerté par le bon flag, de récupérer l'objet dont il est prédestiné à le faire.

Le *Receiver* n'est en fait qu'une l'abstraction du flux de sortie de la connexion réseau.

4. Implémentation (intégration) de la pseudo- sérialisation dans le logiciel To :

4.1. Introduction :

Auparavant, To a été construit autour de la pseudo mobilité des agents eux-mêmes.

Nous avons trouvé que cette idée ne correspond pas à la réalité de ce qui se faisant réellement.

Etant donné, que nous sommes contraint à n'opérer que par pseudo mobilité dans j2me, nous avons préféré ne faire migrer que les messages eux-mêmes, c'est-à-dire que dans les nouvelles versions, les To ne sont plus *Migratable*, par contre tous les messages le sont devenus.

Dans **To**, y'a trois sortes de messages. Puisque ces messages héritent tous, de la classe ***GMessage*** ; alors il faudrait maintenant que cette classe implémente l'interface ***KSerializable.***

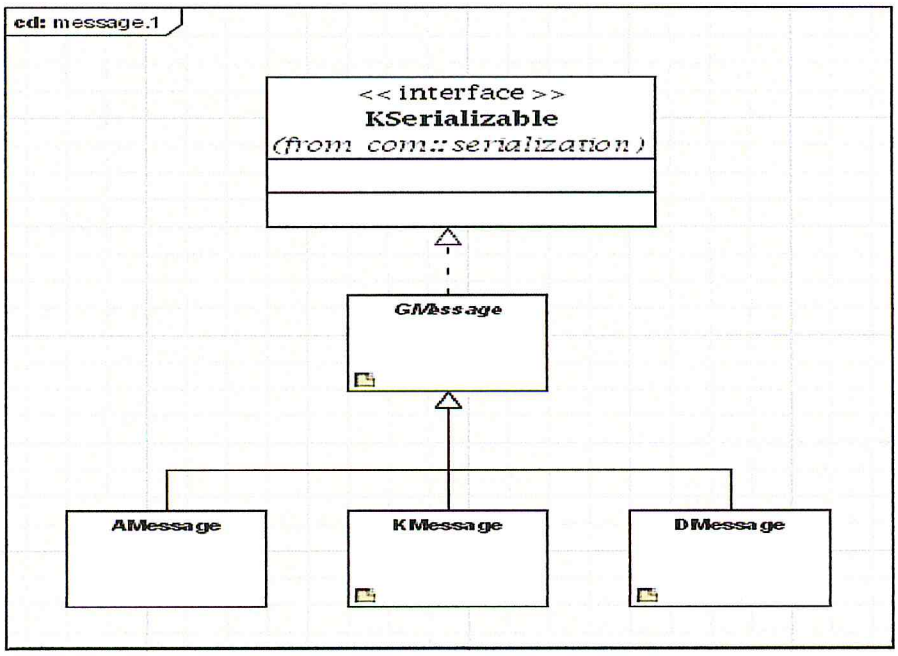


Figure3.26 : GMessage implémente KSerializable

4.2. Explication de la mobilité de messages :

Donc maintenant, comme cela a été expliqué dans le cas de la classe X[1], pour chacun de ces messages, on doit construire un flux spécial, dérivé de *OutputStream* de j2me pour sa sérialisation, et un deuxième, dérivé de *InputStream* de j2me, pour sa désérialisation, comme suit :

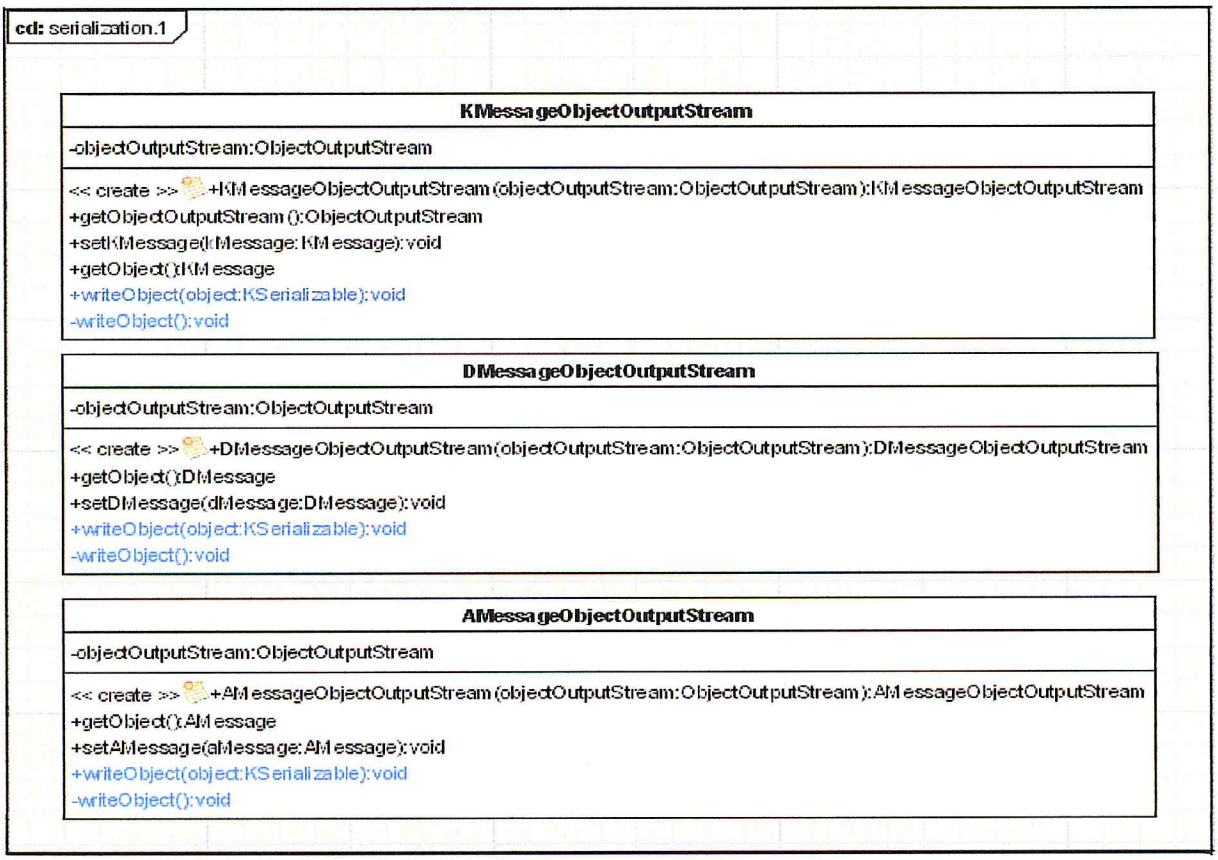


Figure3.27: les flux de sérialisation des messages de To.

[1] Voir sous-chapitre III.6

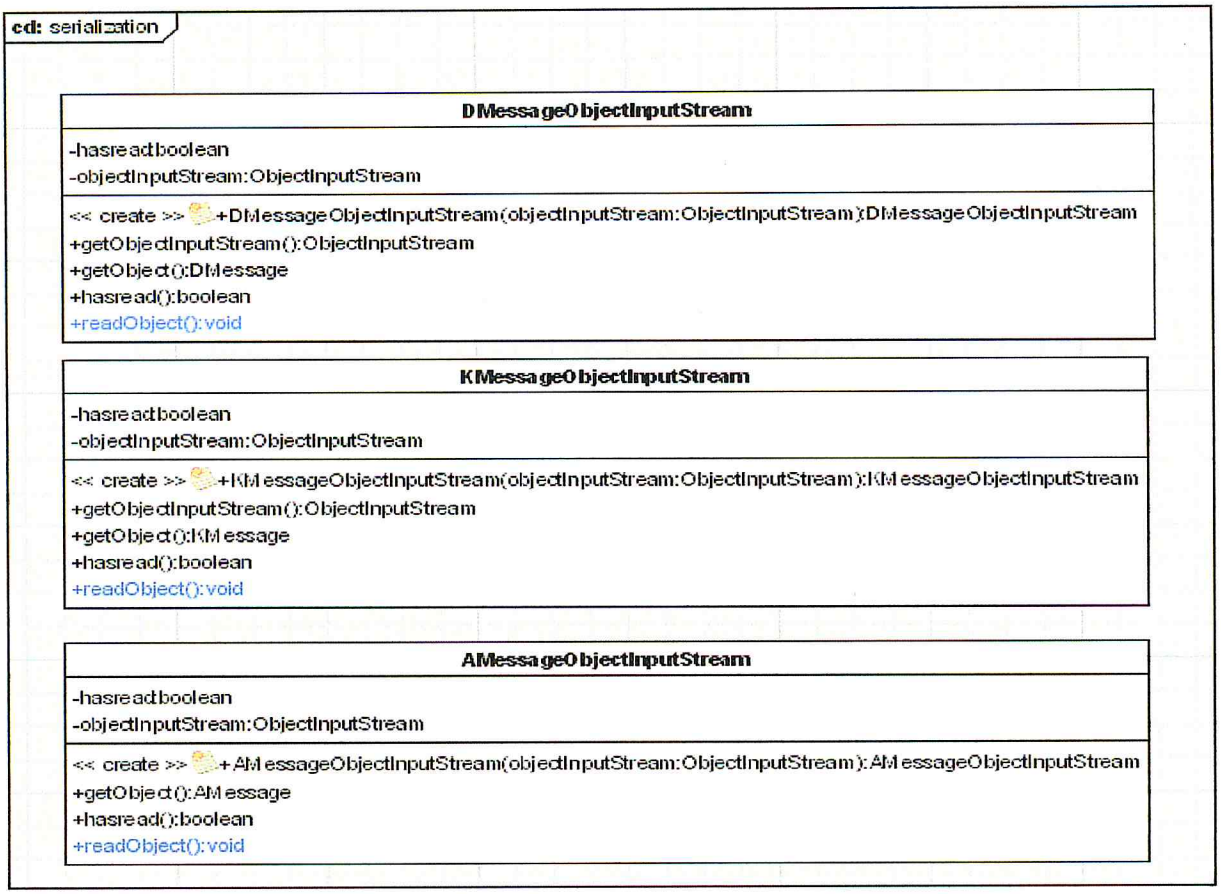


Figure3.28 : les flux de désérialisation des messages de To.

Le problème que nous avions eu à résoudre, c'était d'intégrer toutes ces classes, avec le serveur par *Socket*, initialement construit, et qu'on ne voudrait plus re-fabriquer étant donné qu'il donne toujours satisfaction, en raison de son architecture à classes déléguées et son patron *Proxy* qui est à la base du proxy du serveur.

L'intégration de cette partie dans le Framework ***To***, nous a amené donc à continuer à utiliser l'ancien server[2] de *To*, mais le problème qui en découlait, du fait de cette intégration, c'était comment garder cette architecture de framework, sans tomber dans l'engrenage des cases à incorporer pour le traitement de chacun des messages à travers ses flux particuliers, et éventuellement d'autres éléments à faire migrer que pourraient concevoir d'autres développeurs.

Ce problème, comme cela a été expliqué auparavant, a été résolu au moyen du patron *Observer/Observable*.

[2] Voir sous-chapitre précédant ; package mobility.

Nous avons scindé l'abstraction *Client*, en deux abstractions, l'une modélisée par un objet délégué spécial appelé *Sender* et l'autre par un objet appelé *Receiver* qui vont être des *Observer*s du *Client* et du *Server*, et qui auront pour tâche de s'occuper chacun d'un seul sens de la double liaison réseau avec le *Client*.

Et, nous avons délégué, pour chaque genre de message son objet *Writer* (*AMessageObjectWriter*, *DMessageObjectWriter*, et *KMessagaObjectWriter*), qui aura pour rôle d'être notifié par le *Sender*, pour démarrer la sérialisation du message particulier, dont il est le *Writer* particulier.

Quand l'information transporté par le flux arrive au *ServerConnexion* (un délégué du *Server*, qui s'occupe d'une connexion particulière avec le *Server*), celui-ci va notifier le *Receiver*, pour que ce dernier, notifie à son tour tous ses *Observer*s (*AMessageObjectReader*, *DMessageObjectReader*, et *KMessagaObjectReader*).

Le rôle d'un *Reader* est donc de s'occuper de la désérialisation d'un objet particulier, et cela après qu'il soit notifié, lui et les autres *Reader*s , au moyen du *flag* construit à partir de la classe du message particulier, par le *Sender*.

Nous allons détailler notre conception de la pseudo mobilité des agents To, en commençant de la plus simple jusqu'à la moins aisée,

4.2.1. Le Message *DMessage* :

1. La sérialisation :

La sérialisation de *DMessage*, est la moins compliquée, car pour faire cela, il suffit juste d'écrire, en premier lieu, dans son flux spécial le nom de l'agent à détruire dans l'autre *host* (environnement)[3].

[3]Voir sous-chapitre précédant ; package mobility.

C'est-à-dire, que dans ce cas, il ne s'agit plus que d'écrire un *UTF* dans le *DataOutputStream* du *DMessage,* tout en le devançant par l'écriture du *Flag* particulier au *DMessage.*

Ce qui est beaucoup moins aisé, c'est comment sera opérée la destruction de l'objet au niveau de l'environnement récepteur.

Nous avons résolu ce problème, toujours par le biais du patron *Observable/Observer*, en attachant l'*Environment* receveur comme *Observer* du *DmessageObjectReader*, et qui va avoir comme tache de chercher dans sa *Hashtable*, l'agent correspondant au nom reçu dans le *DMessage*, et de le supprimer.

1.1. Vue statique :

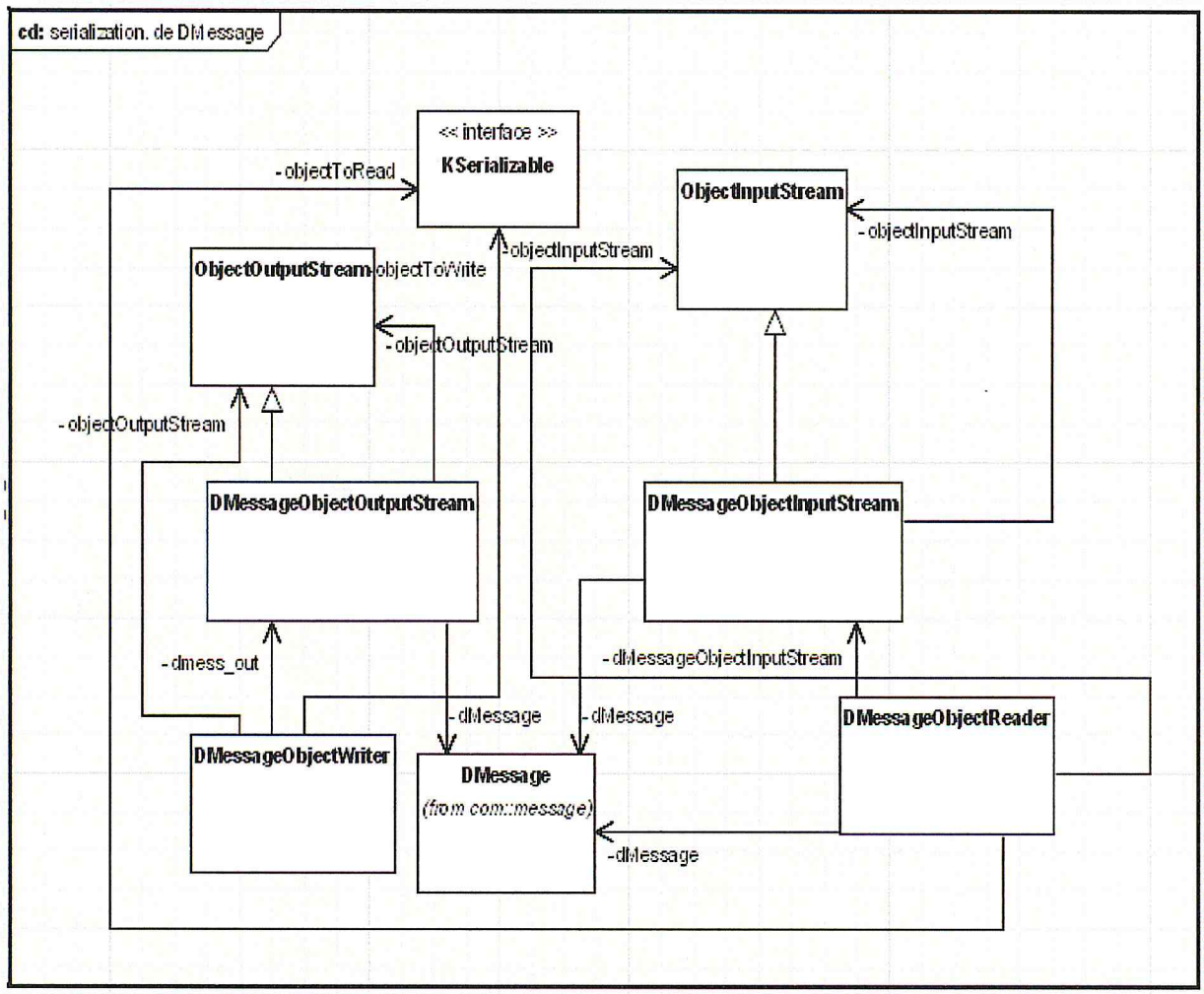


Figure3.29 : diagramme de classe montrant la sérialisation de DMessage.

1.2. Vue dynamique :

Le diagramme suivant décrit la dynamique de l'interaction des objets dans le cas de la sérialisation d'un *DMessage* :

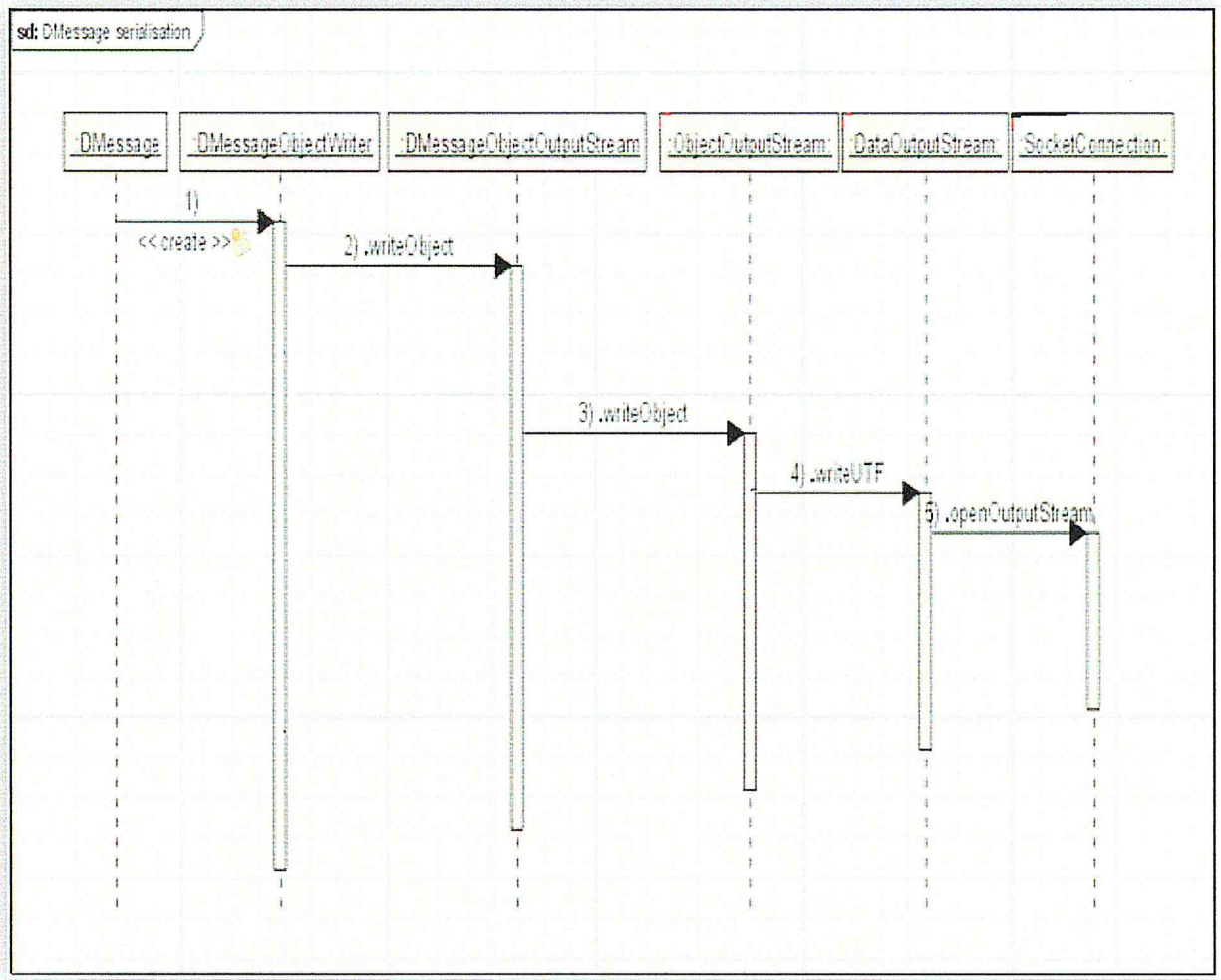


Figure3.30 : diagramme de séquence montrant la sérialisation du DMessage.

2. La désérialisation de DMessage :

Le diagramme de séquence suivant montrant la désérialisation de DMessage ; c'est-à-dire la lecture de nom de l'agent à détruire.

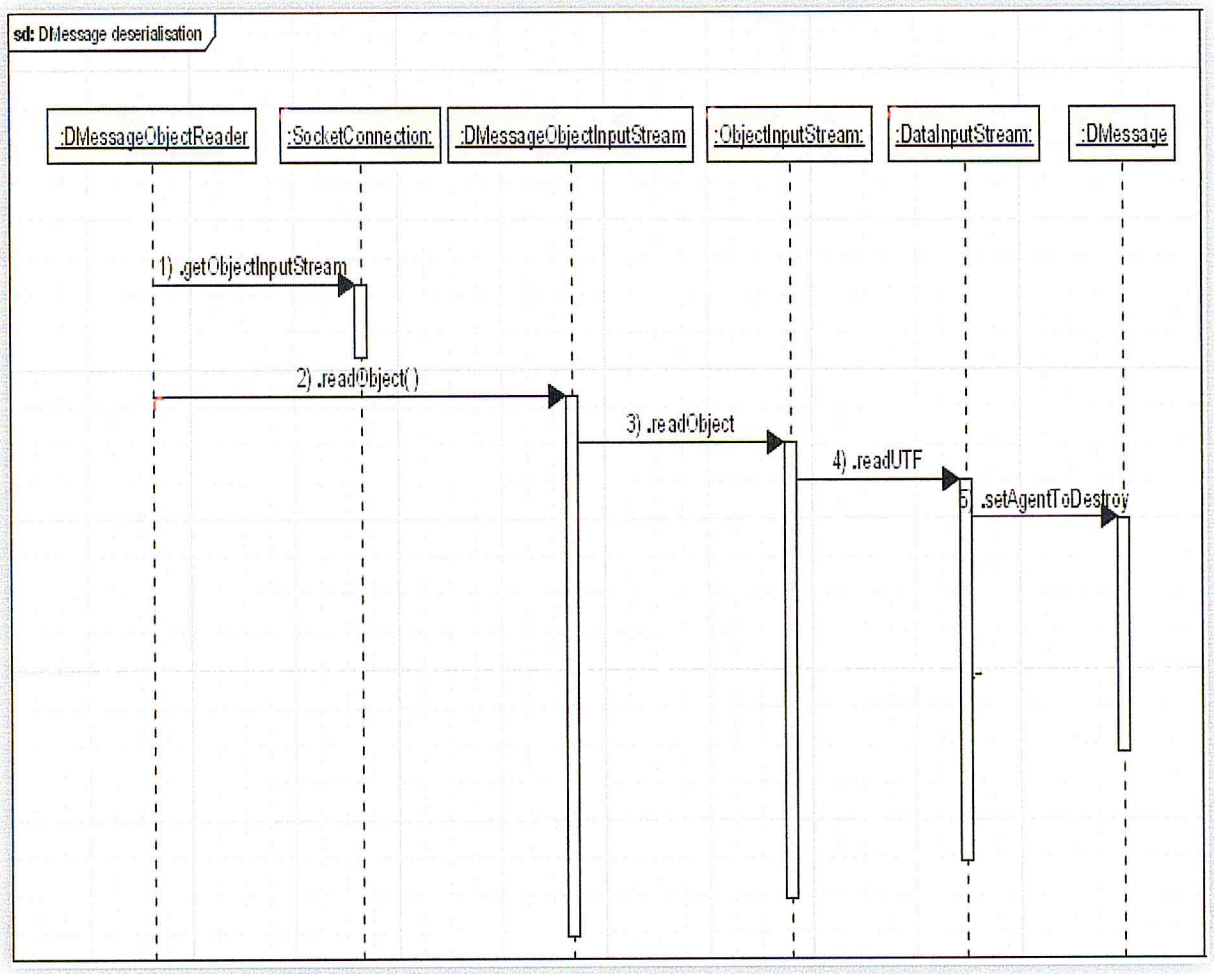


Figure3.31 : diagramme de séquence de la désérialisation du DMessage.

3. la migration du DMessage :

La migration d'un DMessage est effectuée grâce à la méthode *migrate()*, cette dernière, lance un thread, le *DMessageMigrateRunner*, qui lui à son tour s'occuper de lancer la méthode *migrateFct()* du *DMessage* : elle crée un *Client* qui va s'occuper de récupérer le flux de sortie depuis la socket, pour qu'il le passe ensuite au Sender, qui va l'utiliser, pour envoyer l'objet *DMessage*.

Le Sender va ensuite notifier tous les *Writer*, dont parmi eux le *DMessageObjectWriter*, qui lui, par rapport aux autres va reconnaître que c'est lui l'intéréssé étant donné qu'il reconnaîtra cela grâce au flag correspondant.

Le *DMessageObjectWriter* , va récupérer le restant du flux, qu'il sait maintenant qu'il s'agit bien d'un *DMessageObjectOutputStream* , il va donc en désérialiser un objet *DMessage*.

Du côté serveur, Le *ServerConnexion*, lorsqu'il reçoit dans le flux d'entrée de la socket l'objet, il notifie le *Receiver* pour qu'il lise ce message. Ce dernier après la lecture du *Flag* , il exécute *notifyObservers()* qui notifie ses *Observer*s par ce dernier.

Dans ce cas, c'est le *DMessageObjectReader*, qui va se reconnaître sujet de la notification, « tant donné le flag, va récupérer, de la même façon, comme s'est expliqué plus haut, le flux de lecture, il qu'il s'agit bien du *DMessageObjetcInputStream*, pour le lire.

Ensuite, après sa lecture le *Reade*r va notifier l'*Environnement*, qu'il s'agit d'un *To* à détruire par la connaissance de son nom. Ce dernier le supprime de la liste des To de l'*Environment* par l'exécution de la methode *removeToIfExistByName()* de cette liste.

C'est ce qui présenter dans le diagramme de séquence suivant :

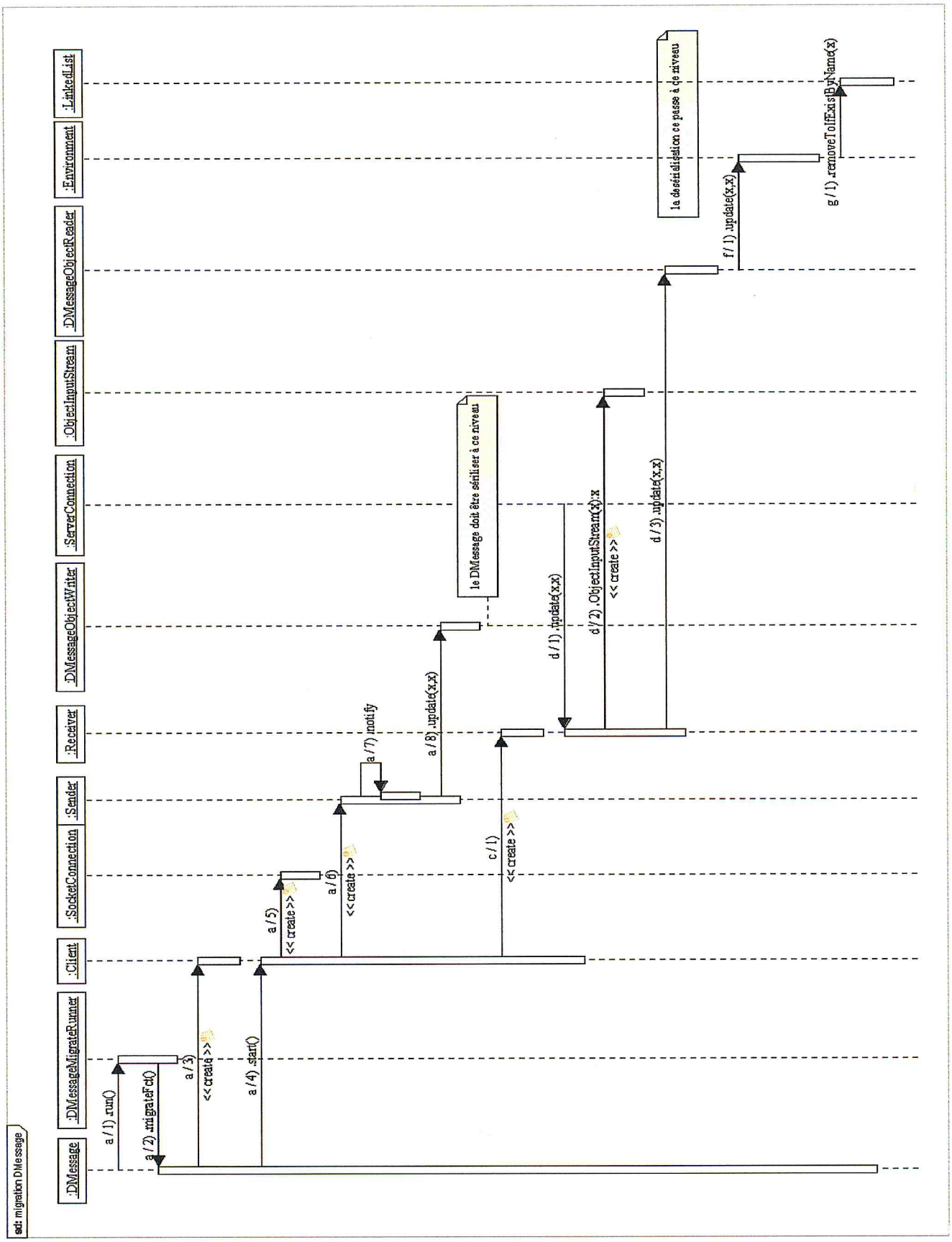


Figure3.32 : diagramme de séquence montrant la migration de DMessage.

4.2.2. le Message AMessage:

La création d'un agent va dépendre :

- du nom de l'agent à créer,
- du nom de la classe de l'objet décoré(*Man*),
- du nom de la classe du *To* (*ToMan*),
- des arguments pour la construction de l'objet décoré, en l'occurrence *Man* ici, c'est à dire son nom, son âge, et son état.

1. Description du processus de création d'un To à distance:

Dans le processus de création d'un To à distance, il va d'abord s'agir de sérialiser un AMessage.

Cette sérialisation, va s'occuper de mettre le *AMessage*, ou précisément les informations données plus haut dans le flux *AMessageObjectOutputStream* qui va les transporter sur le réseau. Le *AMessageObjectOutputStream*, va faire cette sérialisation, par l'intermédiaire de *EncryptedInputOutputStrem*[4], *et ObjectArrayOutputStream.*

EncryptedInputOutputStrem s'occupe de sérialiser les trois premières informations données ci dessus, et *ObjectArrayOutputStream* de sérialiser un tableau d'*Object*s (*ObjectArray*), qui ne sont au fait que les arguments de l'objet décoré, en l'occurrence *Man*, dans le cas de notre application.

Nous allons nous servir de *ObjectArrayOutputStream* pour y mettre les *Object*s du tableau.

Nous pourrions pour le faire, comme auparavant, choisir un *Writer* pour chacun des éléments de ce tableau. Il faudrait donc prévoir, pour chaque indice, le *Writer* qui lui

[4] La conception de la Sérialisation de AMessage été fait après la réalisation de sécurité : sous-chapitre III.3.

correspond. Mais le problème qui se pose ici, c'est que nous n'avons aucune idée sur les classes des objets des éléments du tableau.

Nous avons résolu ce problème par l'utilisation de pattern *Observer*, de la façon suivante:

L' *ObjectArrayOutputStream* parcoure le tableau, et pour chaque indice il écrit son *Flag*, avec quoi il va notifier tous les Writer d'arguments possibles.

Les *Writer*s comparent ce *Flag*, par rapport au nom de la classe dont il est spécialisé à écrire, et s'il trouve qu'il détient le flag approprié, c'est-à-dire celui qui correspond à la classe qu'il sait écrire, il spécialise le Stream en Stream de la classe à écrire, et il lance le processus de sérialisation de l'*Object* de la classe à écrire.

Ce qu'il faudrait donc prévoir c'est des classes *Writer*s et des classes de flux spécialisées, pour tous les types de classes qu'on pourrait avoir comme arguments.

Nous finissons par avoir des classes de la sorte *KSerialsableClassWriter* et *KSerialisableClassOutputStream*, avec Class étant le nom de la classe *String*, *Integer*, *Boolean*, Etc.

1.1.Vue statique: Le diagramme de classe:

Le diagramme de classe suivant, donne une vue statique sur les classes qui entrent dans la sérialisation de AMessage;

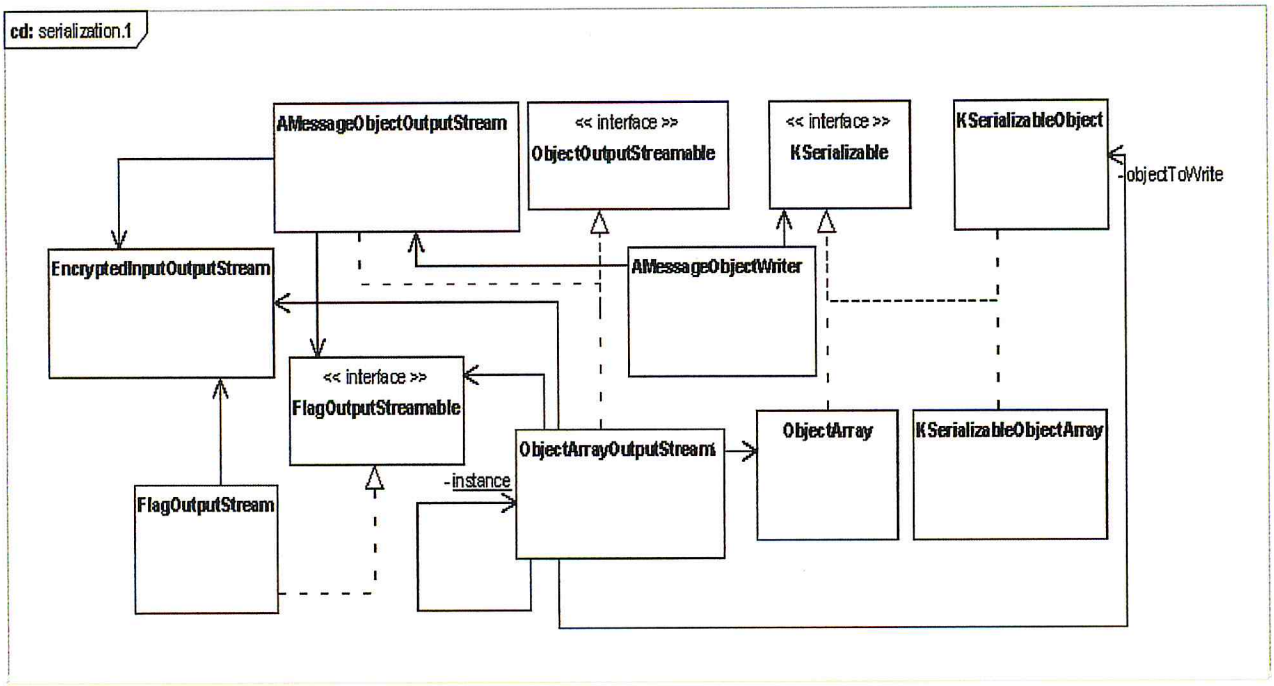


Figure 3.33 : diagramme de classe montrant la sérialisation de AMessage.

1.2. **Vue dynamique:** diagramme de séquence:

On va expliquer dans le diagramme de séquence suivant l'enchainement des phases de la sérialisation, pour montrer du point de vue dynamique comment s'effectue la migration d'un *AMessage*.

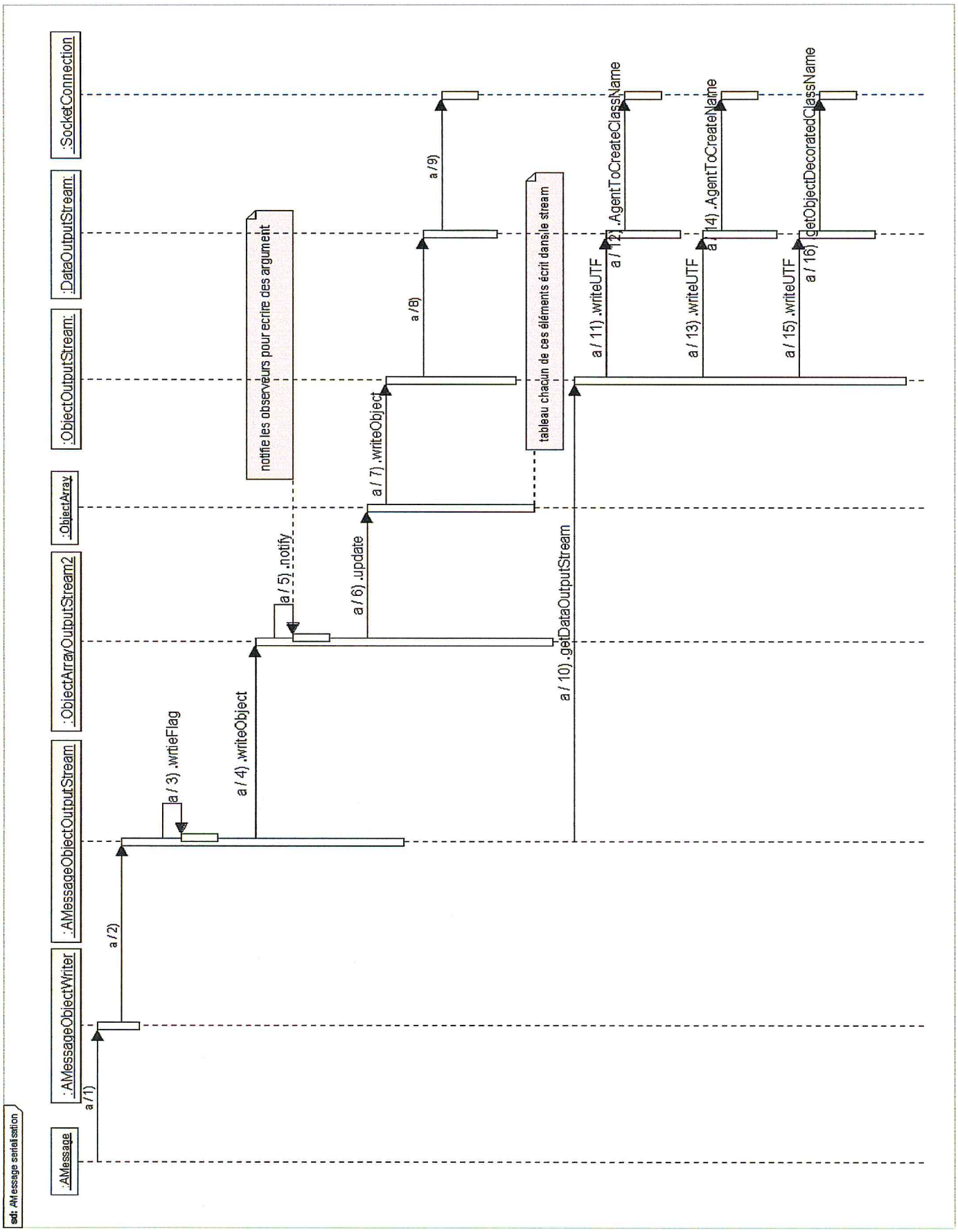


Figure 3.34 : diagramme de séquence montrant la sérialisation de AMessage.

2. **La désérialisation :** La désérialisation, ou bien la création de l'agent, est présenté dans le diagramme de séquence suivant

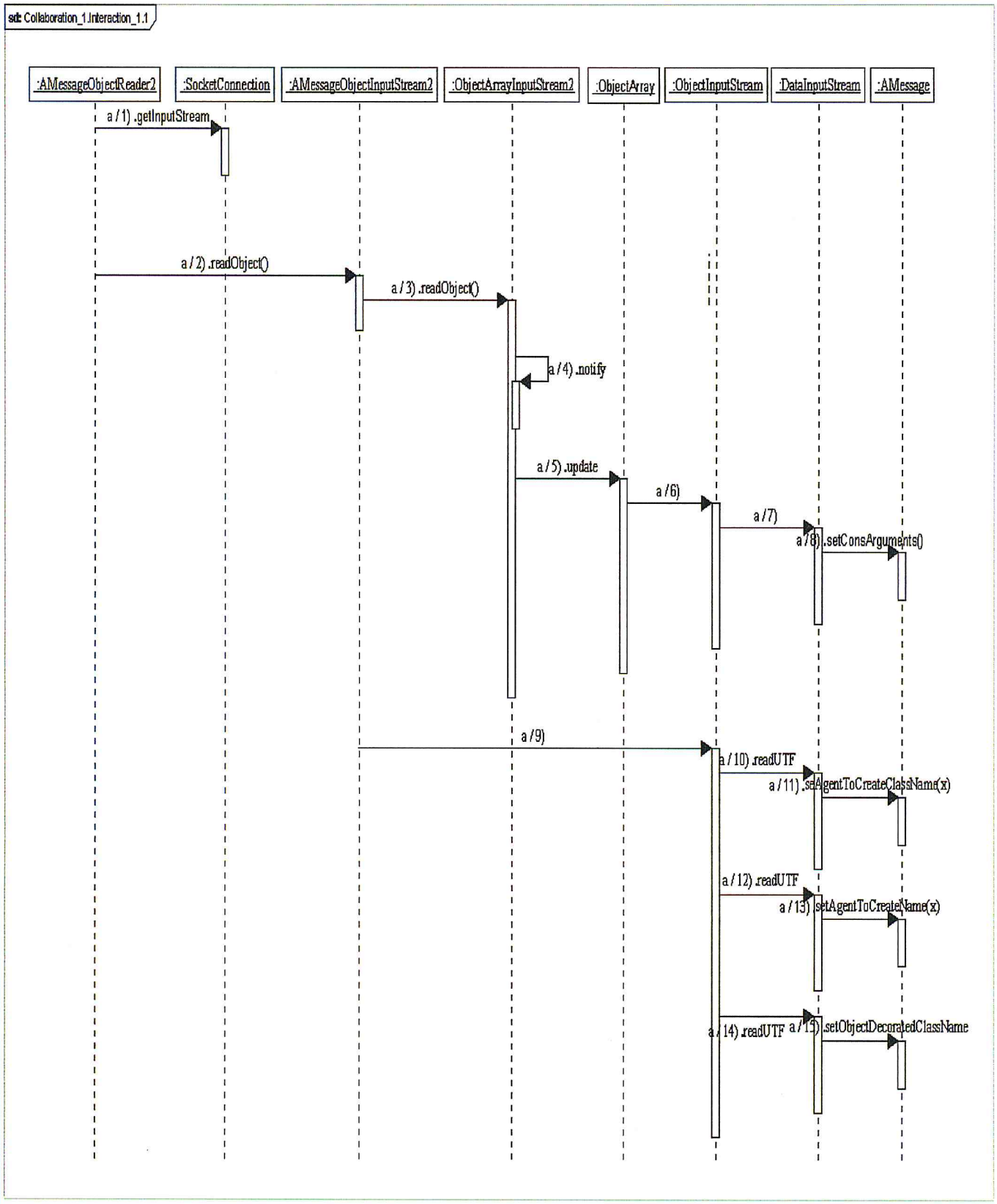


Figure 3.35 : diagramme de séquence montrant la désérialisation de AMessage.

3. la migration du *AMessage* :

La migration d'*AMessage*, elle sera donnée par le diagramme de séquence qui suit. Pareillement au *DMessage*, elle s'effectue toujours grâce à la fonction *migrateFct()*.

A la réponse de ce dernier, les étapes sont analogues à celles du *DMessage*, sauf que dans la sérialisation, et c'est ce qu'on avait déjà expliqué plus haut, au niveau du *AMessageObjectReader*, après la récupération de l'objet, par *getObject()*, il exécute les méthodes *createTo()*, et *getAgentToCreate()*, qui permettent de créer un nouveau *To*. Ce dernier doit notifier l'*Environment*, et *World*, pour les rajouter dans la liste des *To*s.

3.1.vue dynamique:

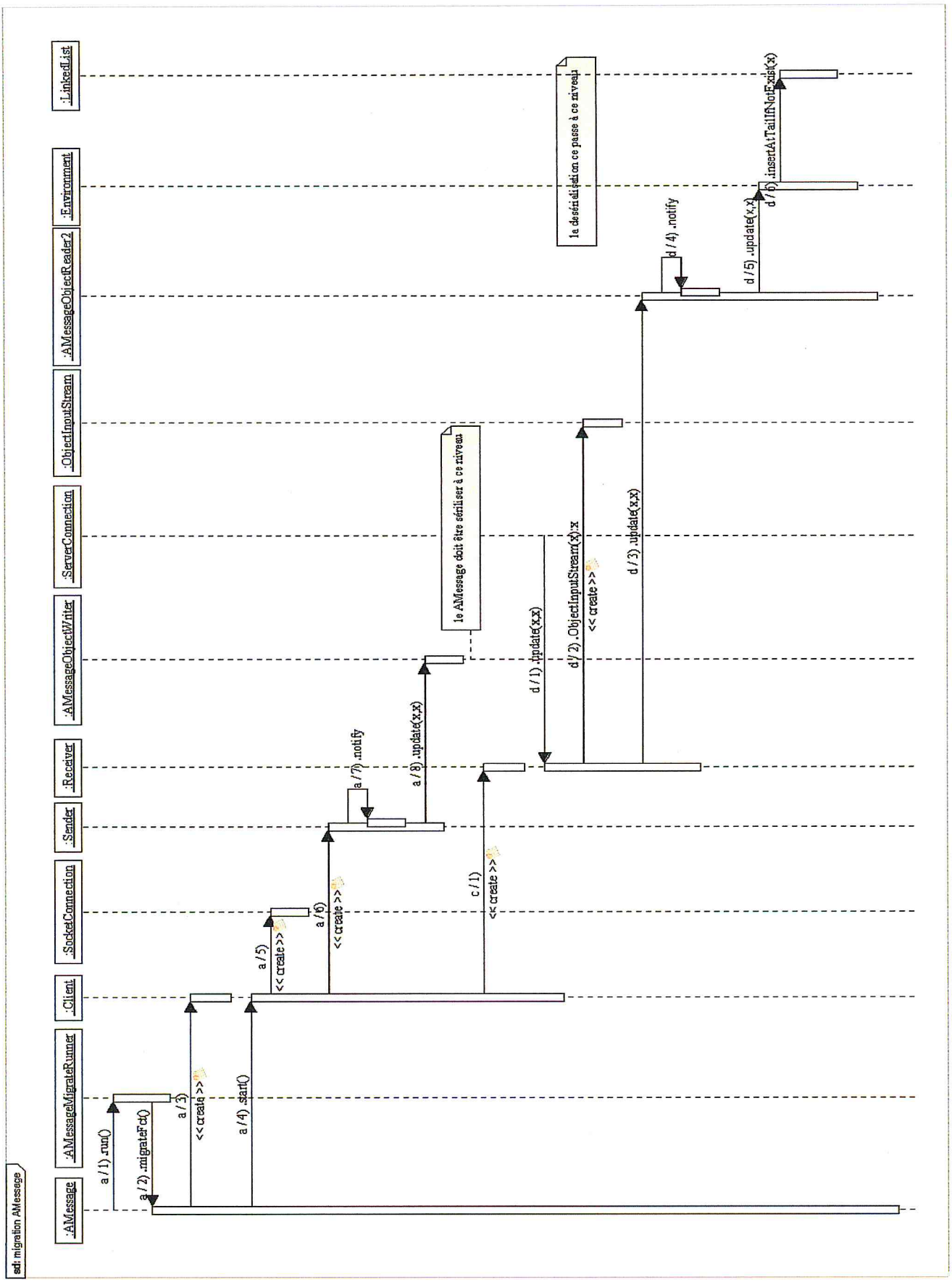


Figure 3.36 : diagramme de séquence montrant la migration d'AMessage.

5.Package de sérialisation :

Pour pouvoir simplifier l'accès à la partie sérialisation, nous avons préféré créer un package que nous avons nommé *serialization.* Ce package nous permet surtout, d'accélérer la recherche de classes, dans les parties développement et tests.

Le diagramme suivant, représente sous forme d'un diagramme de classes, les relations entre les classes du package, sauf celles rentrant dans l'écriture dans des flux.

Cette représentation d'un package, nous est plus utile que celle plate, représentant un conteneur de classes, du fait, bien sûr, de l'incorporation des relations entre les classes.

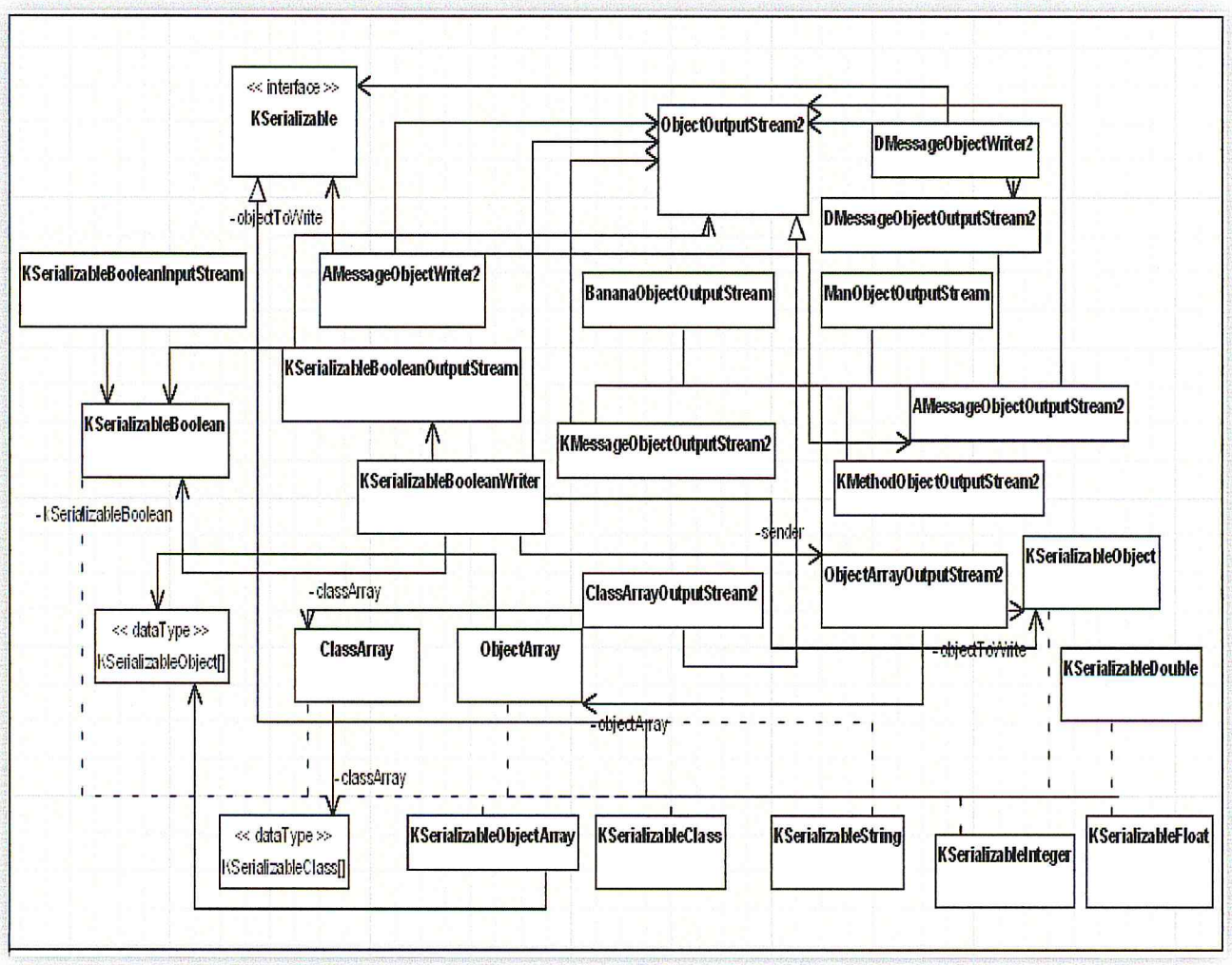


Figure3.37: diagramme de classe de package de sérialisation(écriture).

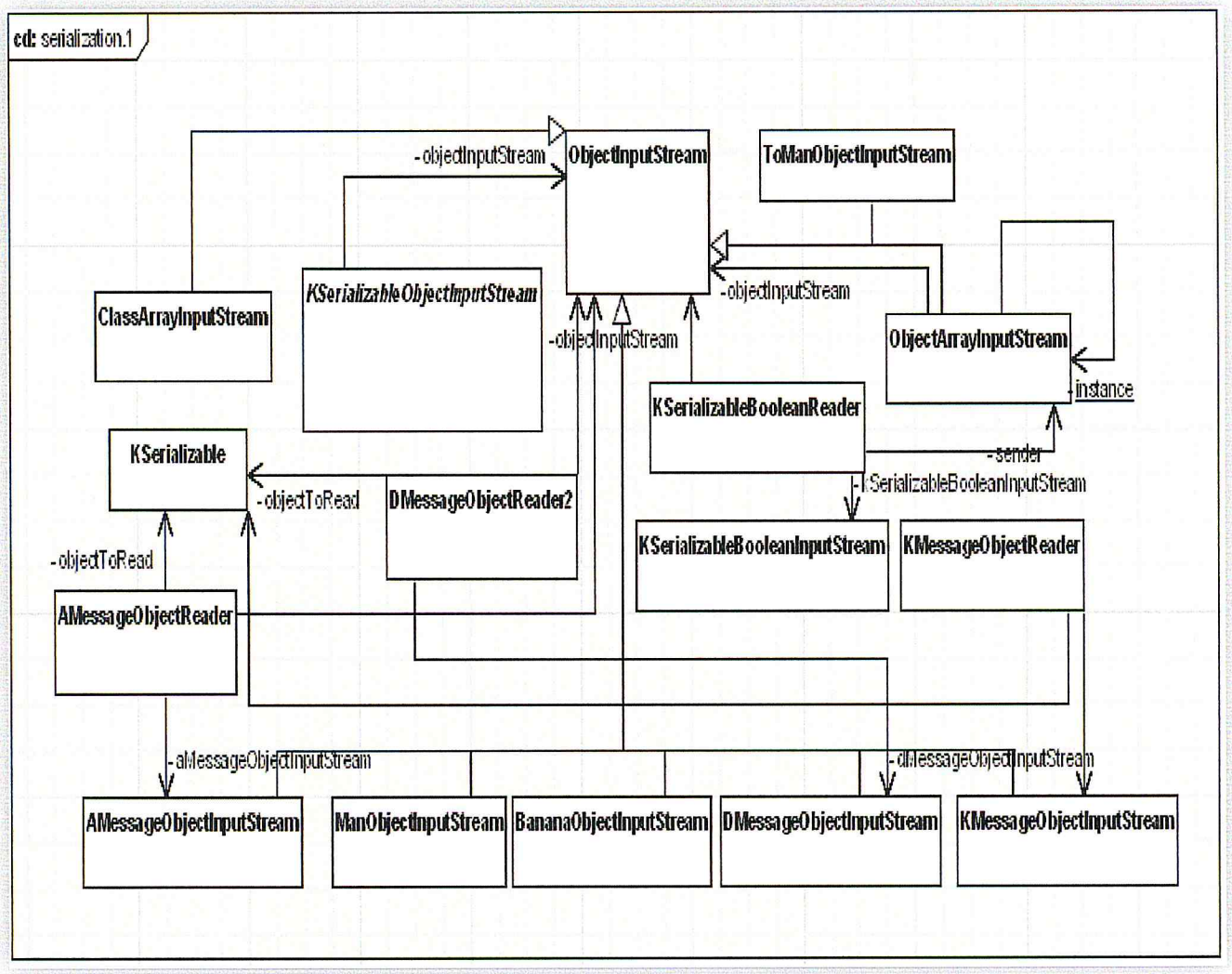


Figure3.38: diagramme de classe pour le package de sérialisation (lecture).

6. Conclusion :

Nous avons présenté dans ce chapitre les étapes de réalisation du pseudo de sérialisation, et l'ensemble de re-fabriacation (refactoring) fait, pour satisfaire la mobilité de messages.

III.3 SECURITE EN (To)

1. INTRODUCTION

Dans l'étape d'étude et d'analyse du Framework *To* par les techniques de reverse-engineering, nous avons remarqué l'absence de l'aspect sécurité dans le *To* initial, dans la transfert réseau. Cet aspect va donc être beaucoup plus un aspect de re-conception. les plus important pour les logiciel.

Dans cette partie de mémoire nous allons montrer comment nous avons, à travers une re-fabrication progressive, nous sommes parvenu à adjoindre l'aspect sécurité au Framework *To*.

Pour la conception de la sécurité en TO, nous avons adopté deux approches, la première consistant à ne crypter qu'un élément choisi du flux à envoyer sur réseau, alors que dans la deuxième, on crypter tout ce qui s'écrit sur les flux réseau.

2. CONCEPTION ET REALISATION DE L'ASPECT SERCURITE EN *TO*

Afin de la réalisation la couche de sérialisation, par l'utilisation des techniques de développement de logiciel orientés objets; nous voudrions incorporer l'aspect sécurité pour augmenter la valeur du logiciel.

En commençant à réfléchir au coté conception et implémentation, les question qui se sont posées à nous sont :

- A quel niveau du Framework va t'on incorporer le cryptage de l'information,
- Comment va-t-on procéder en vue d'une bonne maintenabilité,

- Quelle information va t'on crypter,
- Quels sont les algorithmes de cryptage à utiliser.

Puisque nos objets que nous envoyons à travers le réseau sont tous affectés d'un flag, permettant de les reconnaître:

Flag ⟹ le nom de la classe (un UTF à écrire dans le flux(Stream))

2.1. Première approche:

La première solution qui nous a paru intéressante est de crypter le moins d'information possible. Notre choix a porté, à ce moment là, sur le flag caractéristique de l'objet à crypter.

Le flag nous semblait une bonne décision, parce-que, comme nous l'avions vu précédemment, sans la connaissance du flag, aucune structure d'information se sera reconnue, et donc aucune possibilité de bonne réception des flux ne pourrait avoir lieu.

Car chaque objet qu'on voudrait faire migrer est caractérisé par un flag. Donc crypter ce flag est suffisant pour rendre méconnaissable le contenu d'un flux, car le *Reader*, qui lit depuis le *Stream* est un *Observer*, et il ne va prendre en compte le flux, à travers lequel il est notifié, que sur la base de la connaissance de la valeur ce flag.

D'un autre côté, pour permettre l'utilisation et le choix des possibilités de l'algorithmique de cryptage par d'autres utilisateurs ou développeurs, et d'écrire un code réutilisable, on s'est rapidement penché sur le patron *Strategy*.

L'idée ici, comme cela a été expliqué auparavant, est de travailler avec des interfaces java, que le développeur ou l'utilisateur, implémentera à sa façon.

Les interfaces choisies, *Decryptable* et *Encryptable*, l'une pour crypter (*encrypt()*) et l'aute pour décrypter (*decrypt()*), sont décrites par les diagrammes suivants :

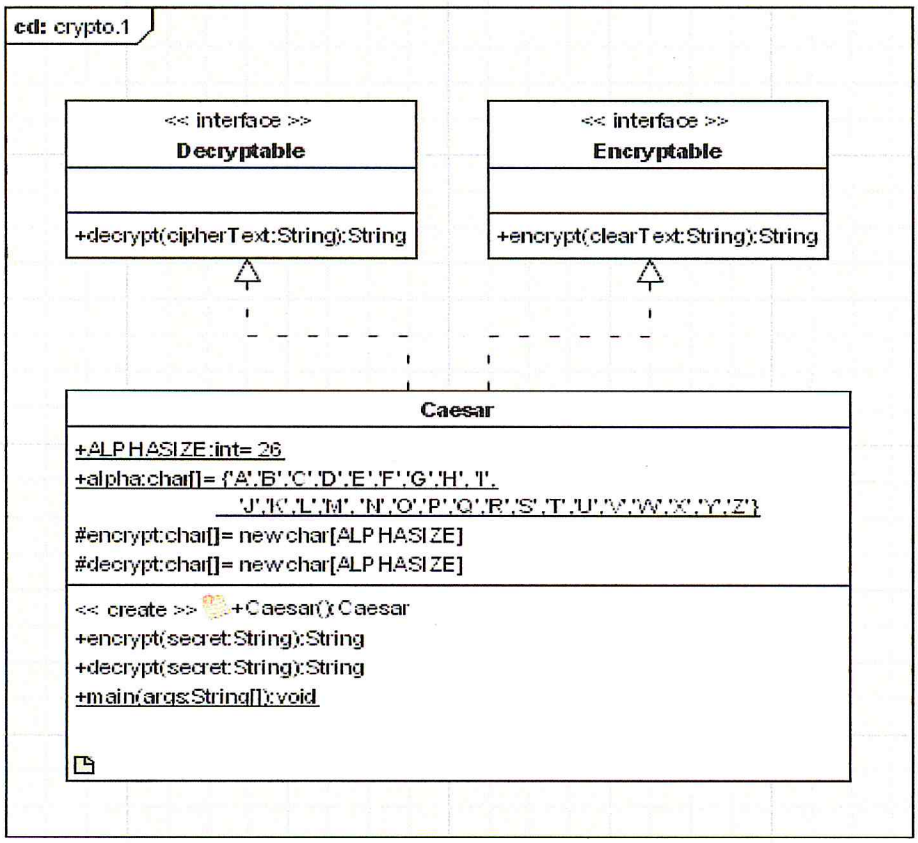


Figure3.39 : diagramme de classe montrant l'algorithme de Caeser.

Maintenant, en ce qui concerne, la conception de l'incorporation des éléments du code pour la réalisation de l'aspect sécurité, nous avons simplement choisi d'utiliser le *Decorator*, mais pour ne décorer dans notre cas ici, que les classes qui comportent les méthodes *writeFlag()* et *readFlag()*.

Du point de vue conception, nous avons été, jusqu'à la source des flux réseau, c'est à dire dans le *Client*, ou le *Server*, pour leur attacher, à travers le patron *Strategy*, un encrypteur et un décrypteur.

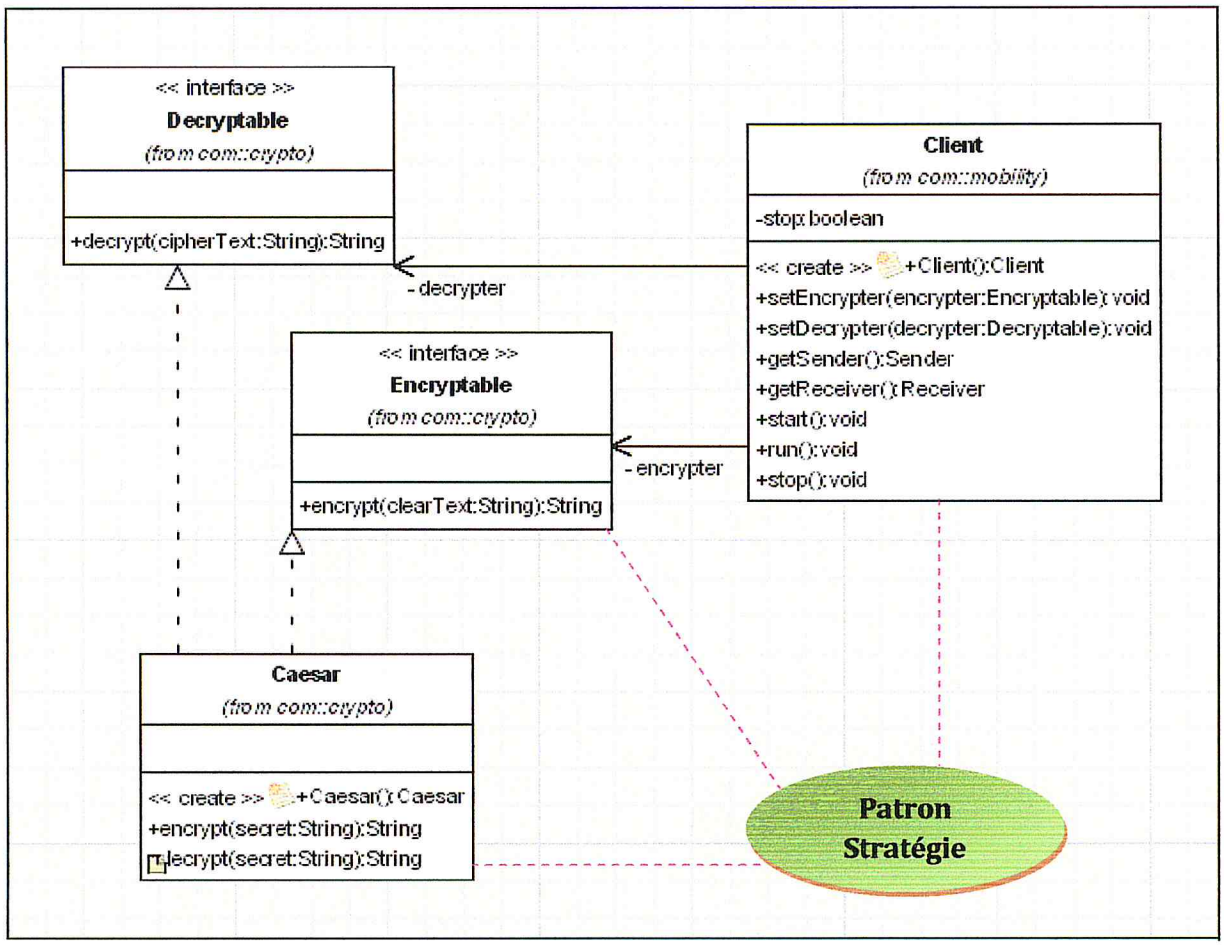


Figure3.40: diagramme de classe présentant le pattern Strategy.

L'implémentation de l'algorithme de *Caeser*, doit être comme suit :

```
Client client = new Client();
Caesar cipher=new Caesar(); //Vigenere cipher=new Vigenere("to");
client.setEncrypter(cipher);
client.setDecrypter(cipher);
```

Remarque :

Dans le bout de code précédemment, on a implémenté l'interface de *Strategy*, par un algorithme *Caeser*[1],

[1] Voir sous-chapitre II.6.

La classe *Caeser :*

Cette classe qui implémente, au moyen de l'algorithme de César, à travers les méthodes *crypter()* pour crypter, et *decrypter()* pour décrypter, les interfaces *Encryptable* et *Decryptable,* comme le montre la figure ci-dessus.

L'incorporation de *Strategy:*

L'idée, est de crypter ou de décrypter, à travers l'écriture ou la lecture, des flux.

C'est à dire, que nous allons crypter en écrivant dans les flux, et décrypter en lisant les flux.

L'incorporation du patron *Strategy,* dans le framework de sérialisation, s'est faite comme le montre le diagramme suivant :

L'idée clef est de faire hériter les flux existants à partir de nouveaux qui vont, à travers *Strategy,* permettre le cryptage et le décryptage.

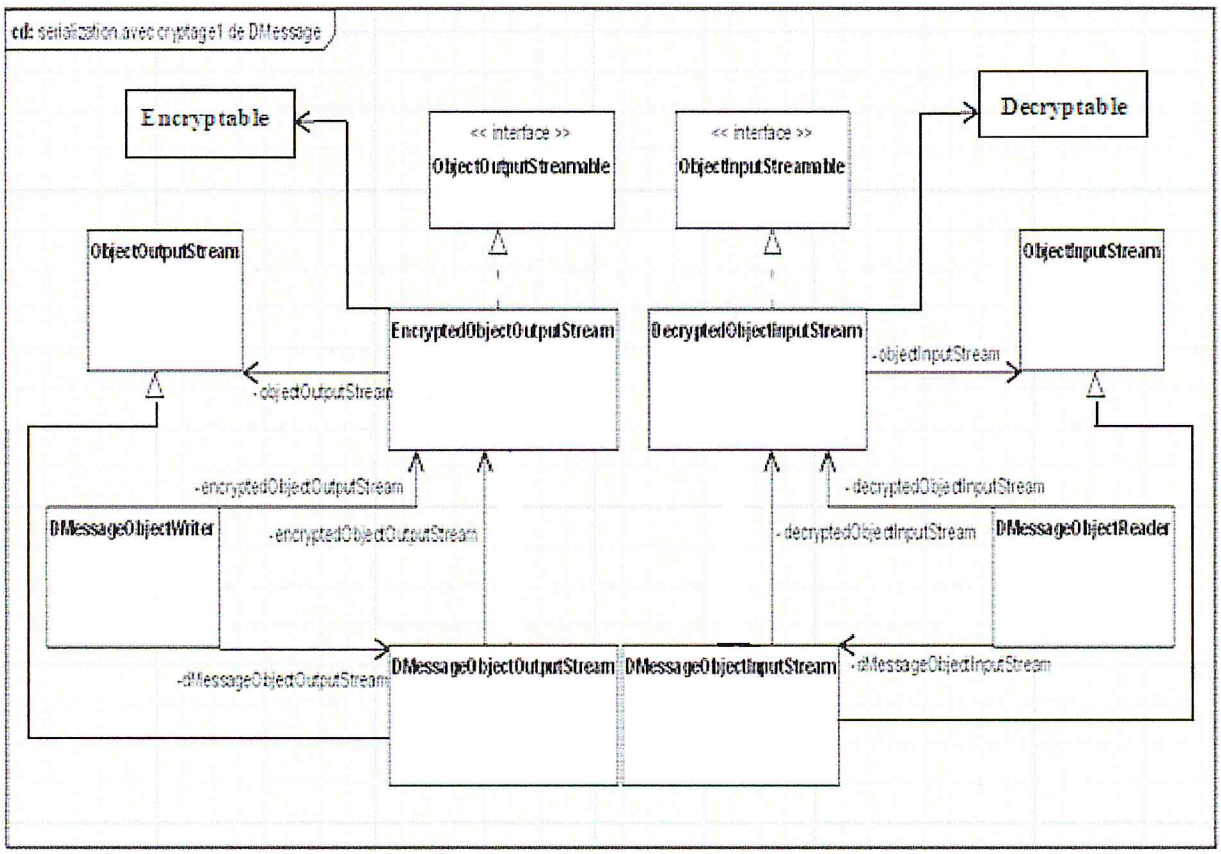


Figure 3.41: diagramme de classe présente la sérialisation de DMessage avec le cryptage.

2.1.1. Explication :

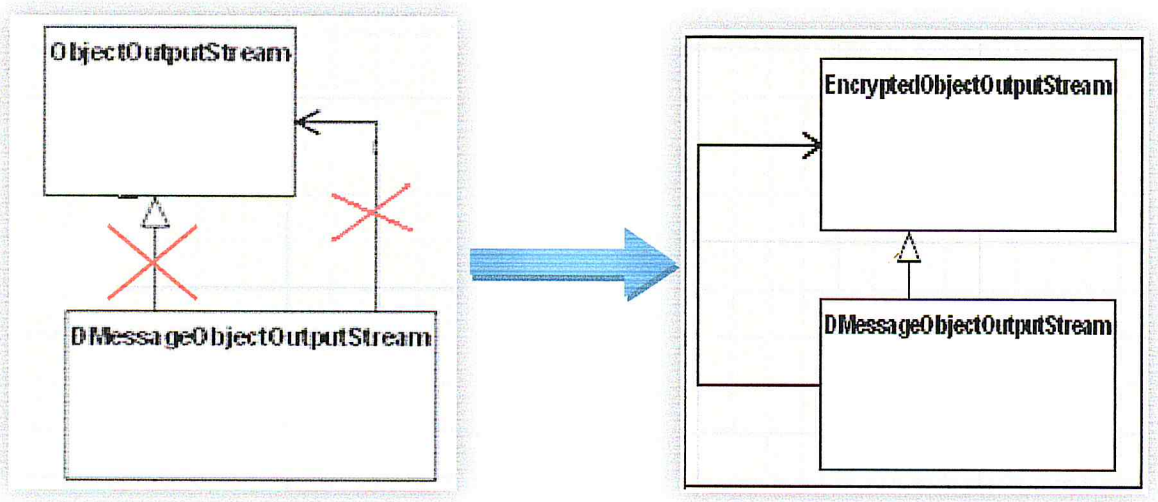
On va maintenant expliquer comment nous avons procédé pour incorporer la sécurité dans le framework de sérialisation.

Puisque c'est le *Sender* qui s'occupe de récupérer le *Stream d'ecriture* depuis la *Socket* du client, pour qu'il le transmette à celui qui va écrire l'objet , le *Writer*,.

Nous avons pensé, impléménter la méthode *writeFlag()*, dans un *EncryptedObjectOutputStream* qui utilise, par composition d'un *YObjectOutputStream* de telle façon qu'il puisse lui rajouter le coté cryptage.

On pourrait schématiser, ce processus, par le représentation suivante :

Avant: les flèches rouges reprrésentent comment nous avons opéré la déconnection, des relations entre les classes pour incorporer les nouvelles relations, incrporant de la sorte le patron *Strategy*.



Avant *Après*

Figure3.42 : refactoring au niveau de flux.

Ça veut dire pas que la classe *ObjectOutputStream* doit être supprimée ; c'est justement le rôle du *refactoring*[2] qu'on utilise ici. Alors, c'est à ce niveau qu'on pourrait poser la question, comment doit-on faire pour garder le comportement inchangé et ne modifierr que la structure ?

Pour résoudre ce problème, notre solution est de rajouter une nouvelle interface qui va s'appeler *ObjectOutputStreamable*, qui a les mêmes méthodes que le *ObjectOutputStream*, à la seule différence que l'*ObjectOutputStream* est une classe, et que l'autre est une interface, c'est à dire ne comportant donc que des méthodes abstraites.

Pourquoi la rajouter ainsi, est une autre question qui pourrait se poser. Alors la réponse est simple, et sera donnée, à travers la figure suivante :

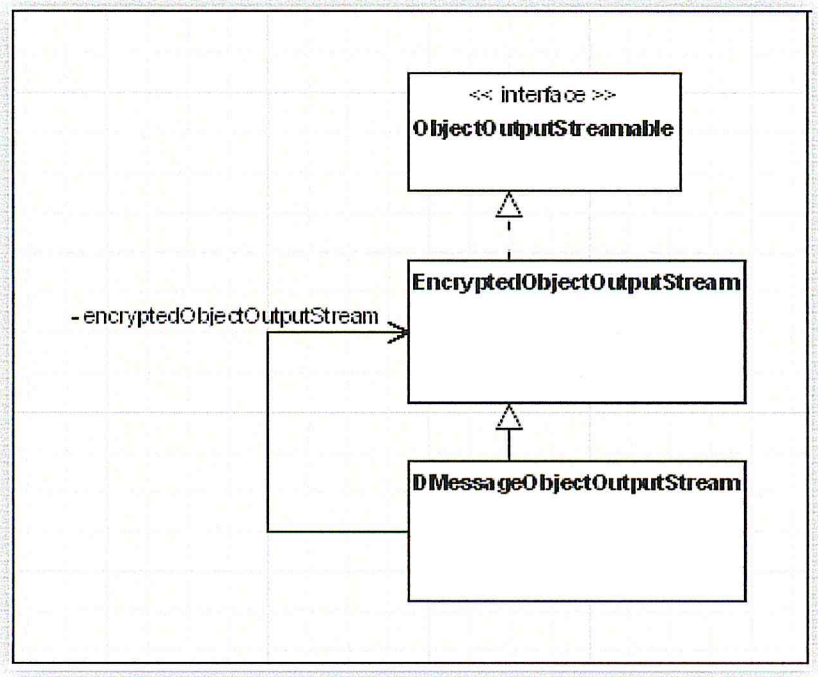


Figure3.43 : EncryptedObjectOutputStream implémente ***ObjectOutputStreamable.***

[2] Voir sous-chapitre II.10.

La classe *EncryptedObjectOutputStream* va implémenter cette interface, et nous lui gardons une référence à la classe *ObjectOutputStream*, pour qu'elle puisse l'utiliser comme son ascendant, c'est à dire comme si elle en hérite, comme présenté dans la figure en-dessous.

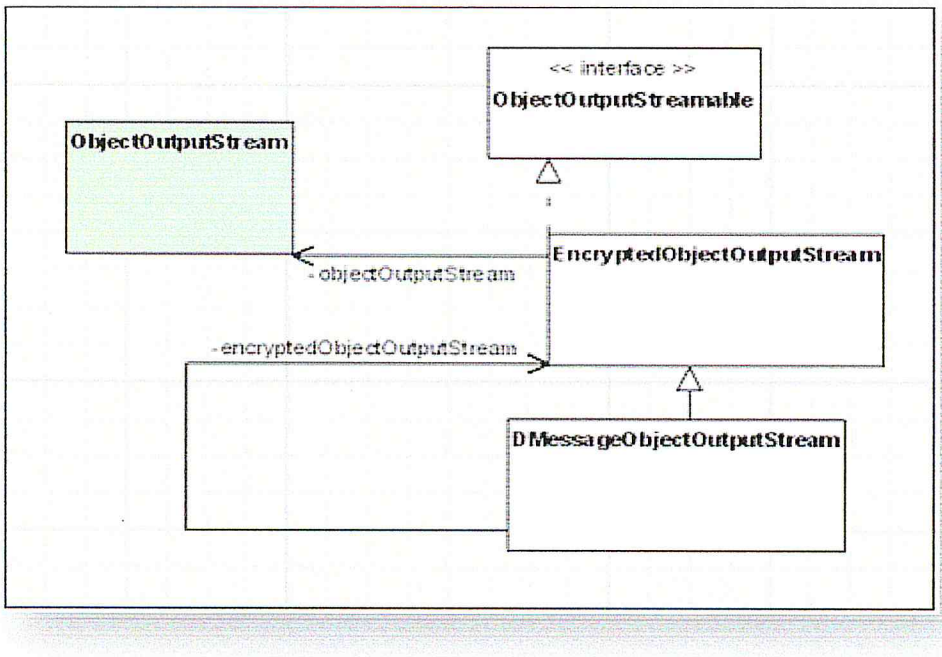


Figure3.44:EncryptedObjectOutputStream "à une référence sur" ***ObjectOutputStream.***

D'après ce qui précède, on peut dire que nous avons gardé un certain niveau d'abstraction, par le fait que nous avions fait remonter la classe *ObjectOutputStream* au niveau supérieur du Framework. Et c'est ce à quoi nous voulions arriver.

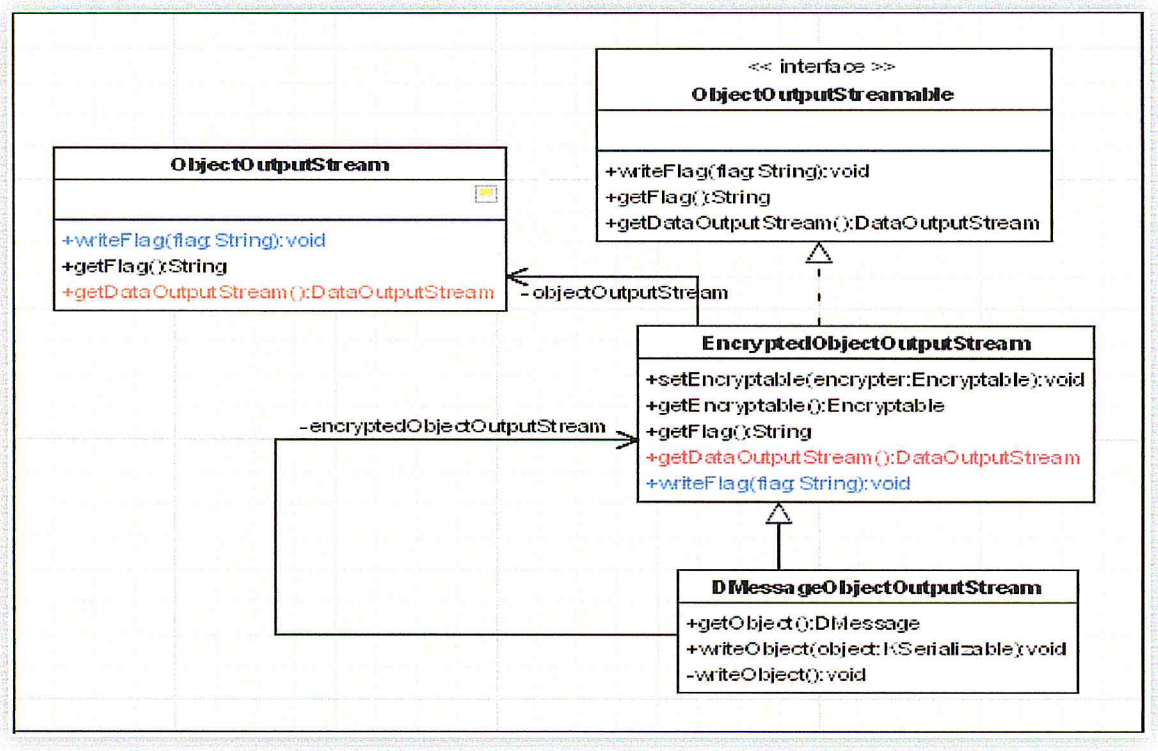


Figure3.45 :diagramme de classe montrant les méthodes.

On rattache le *DMessageObjectOutputStream* (*DMessageObjectInputStream*) avec l' *EncryptedObjectOutputStream*(*DecryptedObjectOutputStream*) par héritage, pour qu'il lui passe la méthode *writeFlag()* qui permet de crypter le flag, et la méthode *getDataOutputStream()* pour que ce dernier puisse l'implémenter depuis l'*ObjectOutputStream*, qui est aussi référencié par ces derniers.

2.1.2. Les avantages :

Les avantages de cette première approche de cryptage du *Flag* seulement, c'est surtout qu'elle permet de réduire le temps de cryptage, tout en ayant une conception par patrons permettant ainsi de grandement améliorer la maintenabilité et la réutilisabilité de notre Framework.

L'algorithme de *César* que nous avons utilisé pour un premier test, pourrait facilement être supporté par un téléphone portable et nous donne un maximum d'optimisation d'espace à cause de sa taille réduite, et de temps d'exécution, en raison de son application un nombre limité de fois.

La solution choisie est assez élégante, de par le choix de l'information à crypter, de l'algorithme de cryptage, et surtout du choix de l'architecture logicielle à base du patron *Strategy* pour la concevoir

et l'incorporer au framework de sérialisation, mais peut-être qu'il faudrait penser à une autre plus généralement applicable.

2.2. Deuxième approche

La deuxième possibilité serait, bien sûr, de crypter toute l'information à transférer sur réseau.

La question qui pourrait se poser de quelle manière il faut procéder pour faire crypter toute l'information de l'objet à envoyer ou recevoir sur réseau ?

Notre solution est représentée dans le diagramme ci-dessous. Grâce à l'utilisation du pattern *Decorator*.

Cette deuxième solution permet d'améliorer en généralisant le choix précédent pour qu'il puisse crypter tout l'objet, et pas seulement le *flag*. A cet effet, on va utiliser le *patron Decorator*, sans pour autant modifier le patron *Strategy* de cryptage décrit auparavant.

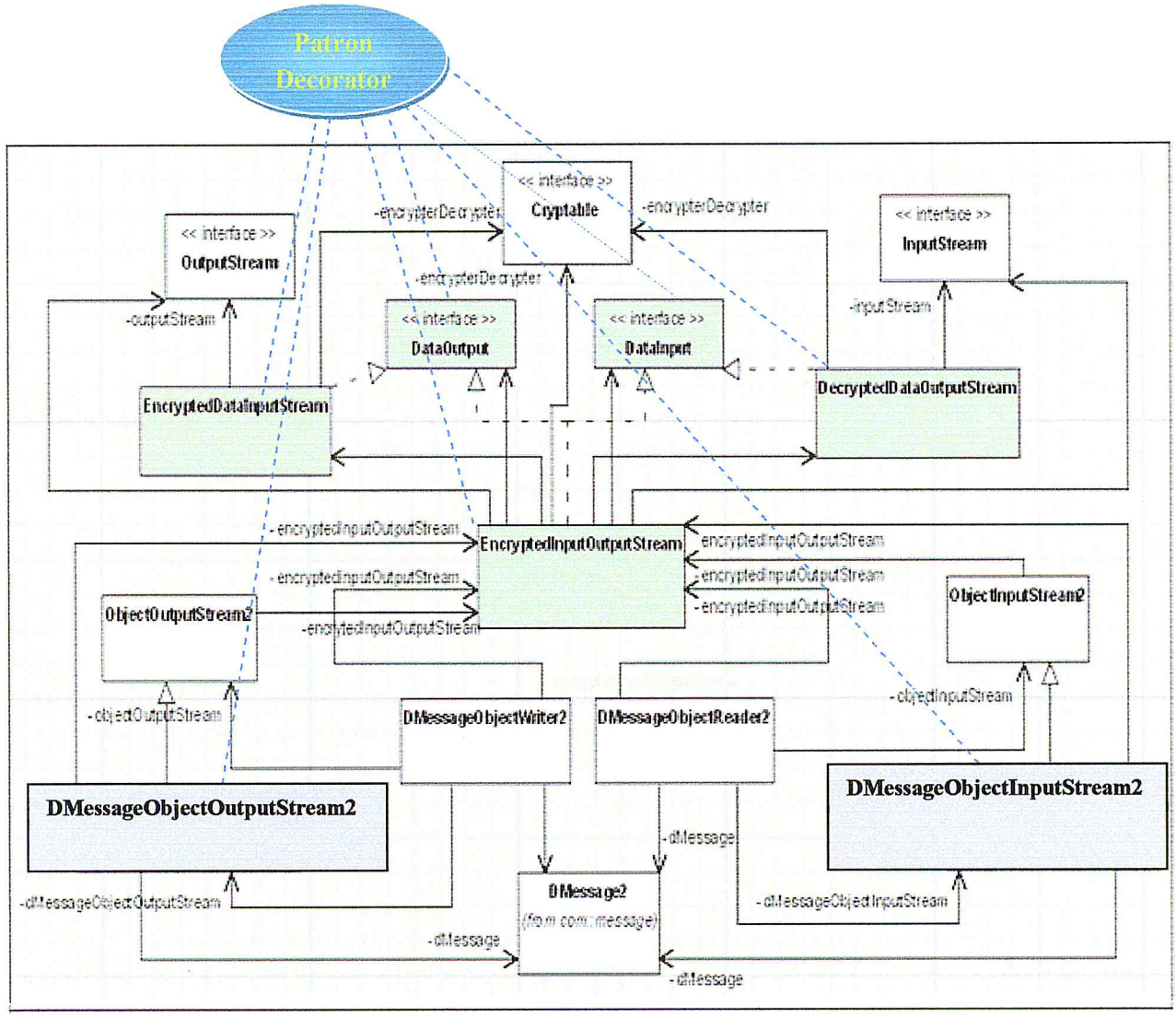


Figure3.46: diagramme de classe montrant le patron Decorator avec le DMessage.

Ce diagramme de classe montre d'une manière générale schématique, l'incorporation du patron *Decorator* dans le framework, dans le cas du cryptage du *DMessage*.

La chose la plus intéressante ici, c'est le fait d'introduire les interfaces *DataOutput* et *DataInput* de *java*, comme interfaces principales, dans la confection de notre pattern. Pour réaliser cette solution, on a préféré englober les deux interfaces de la solution précédente en une seule qui s'appelle *EncryptedInputOutputStrem* et qui permet de crypter et de decrypter, et qui doit implémente les interfaces qu'on a cité en plus haut.

C'est cette classe qui joue le rôle du *Decorator*[3]. Dans notre cas ici, le *Composant Concret* qui doit implémenter la même interface que le *Decorator* est *EncryptedDataOutputStream* et DecryptedDataInputStream. Ceux sont ces deux classes qui permettent d'écrire des bytes cryptés dans le flux, ou de les décrypter.

C'est la principale différence par rapport à la solution précédente, sinon au niveau le plus bas tous les choses restent les mêmes, sauf que notre *DecoratorConcret* est dans notre cas le *DMessageObjectOutputStream* ou le *DMessageObjectInputStream*.

A cause de l'absence de l'héritage multiple en *java*, le problème qui peut se poser ici ; c'est que le *DecoratorConcret* doit hériter du *Decorator*, et le notre hérite déjà de l' *ObjectOutputStream*. Puisque on continue d'optimiser en gardant toujours à l'idée le principe de la réutilisation, et le maintien du comportement, pour remédier à ce problème, en agissant par composition, notre *DecoratorConcret* doit avoir une référence sur ce dernier.

Du point de vue de la *refactoring*, cette dernière solution semble assez intéressante.

L'inconvénient qu'on peut citer avec cette deuxième solution de cryptage : c'est le temps d'exécution ; et pour le résoudre ce problème on n'avait pas utilisé un algorithme lourd comme *RSA* ; mais plutôt on lui a préféré le léger algorithme *César*.

Remarque

Après la réalisation de ce module, et puisque ***To*** est organisé autour de packages, il serait intéressant d'ajouter un autre package à l'application qu'on nomme ***« crypto »***.

Ce package doit inclure toutes les classes qui sont utilisées en relation avec la réalisation du côté sécurité, dans *To*.

Le diagramme suivant permet de montrer les relations avec les différentes classes qui le composent.

[3] Voir sous-chapitre II.3

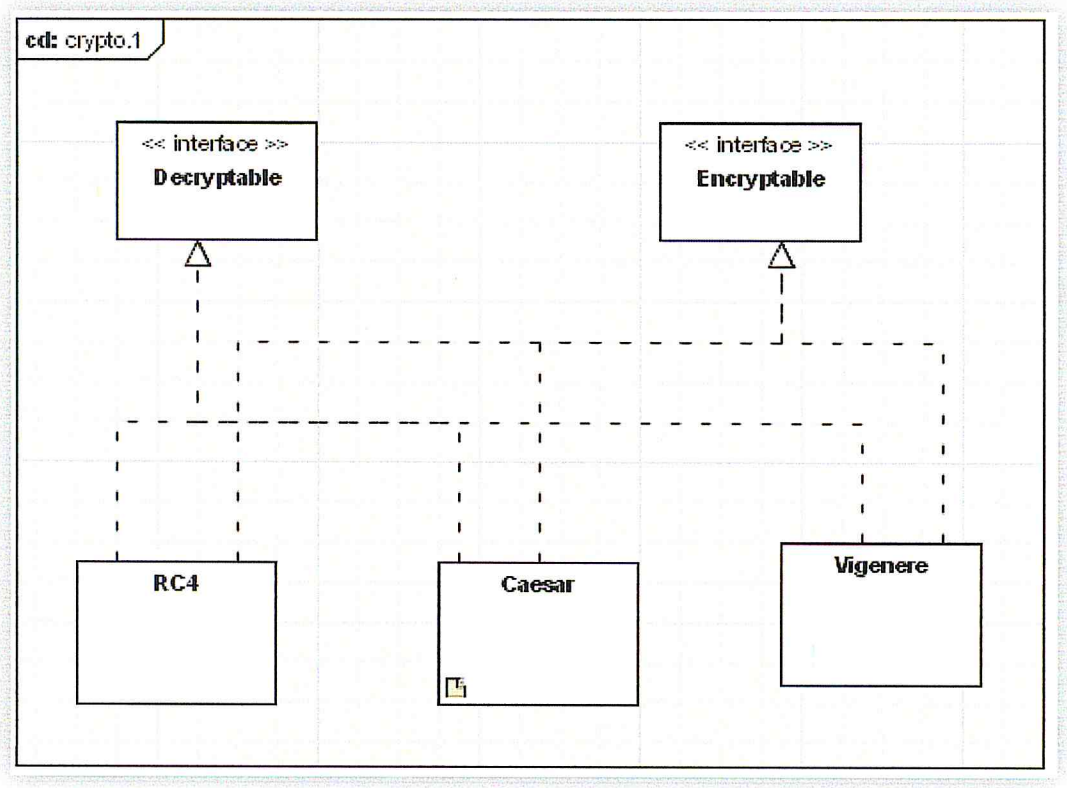


Figure3.47: diagramme de classe montre le package" crypto".

Ce qu'on a fait pour le moment c'est un *Framework* de type white-box *Framework* ; alors on n'a pas encore arrivé à un bon *Framework*, ce qui fait de penser comment le rendre vers un Framework de type *Black-Box Framework*, et comment

On savait bien que la différence entre ces deux types de *Framework*, c'est que le premier est basé sur l'héritage, et le deuxième sur la composition. Alors c'est le deuxième caractère qui le rendre beau. Mais comment ?

3.White-Box Framework *vs* Black-Box Framework :

Les bons frameworks devraient toujours « tourner » (évoluer) du framework blanc au noir, avec le temps. Notre amélioration

Le notre, sans prendre de l'âge, nous avons déjà commencé à le re-fabriquer pour le transformer, en un framework blanc.

Dans notre *Framework*, est tel qu'il est présenté dans le diagramme suivant :

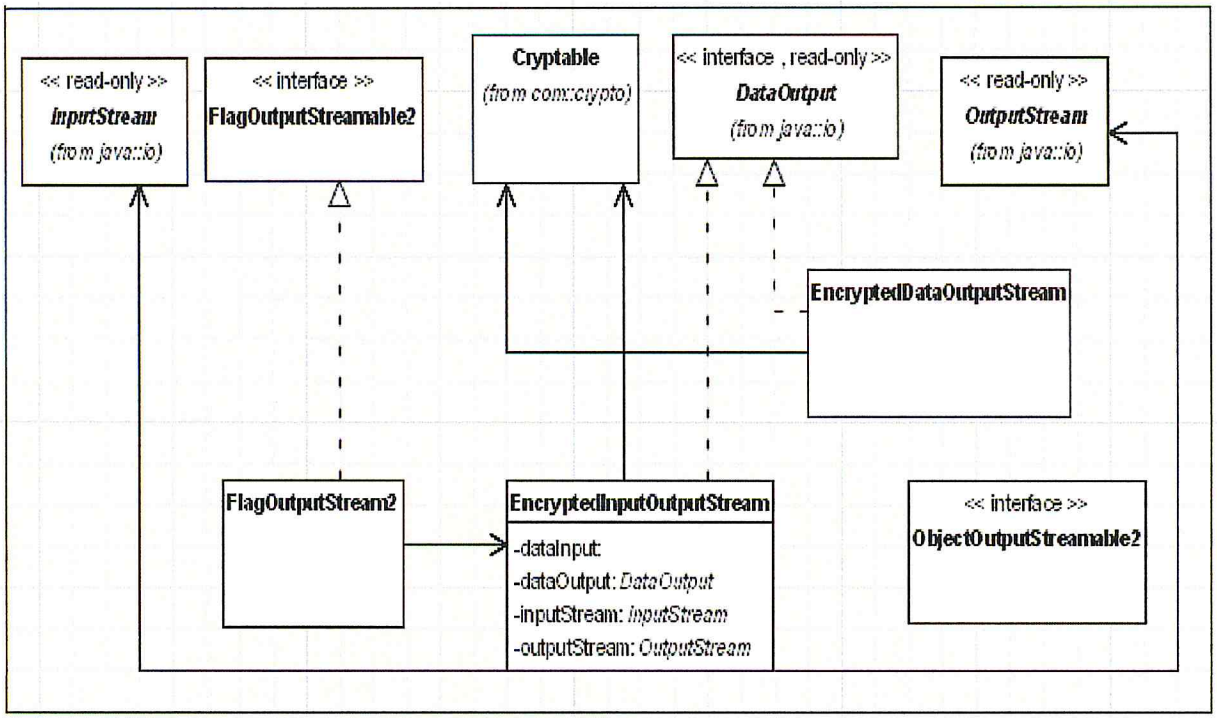


Figure3.48 : diagramme de classe de Framework(pseudo) de sérialisation.

Il y'aurait encore des choses à re-fabriquer et à re-concevoir, et à réfléchir comment y incorporer d'autres interfaces possibles, à la place de classes :

La classe *ObjectOutputStream* elle a comme attribut le *Flag*, et que un *Stream* il permet d'écrire des objets, alors ce n'est pas une caractéristique d'un Stream, mais on est besoin d'elle,

En réalité un flux s'occupe d'écrire un objet, pas un flag. Ce qui nous amène à modifier dans notre . . . La même remarque pourrait être faite pour le cas du flux de lecture.

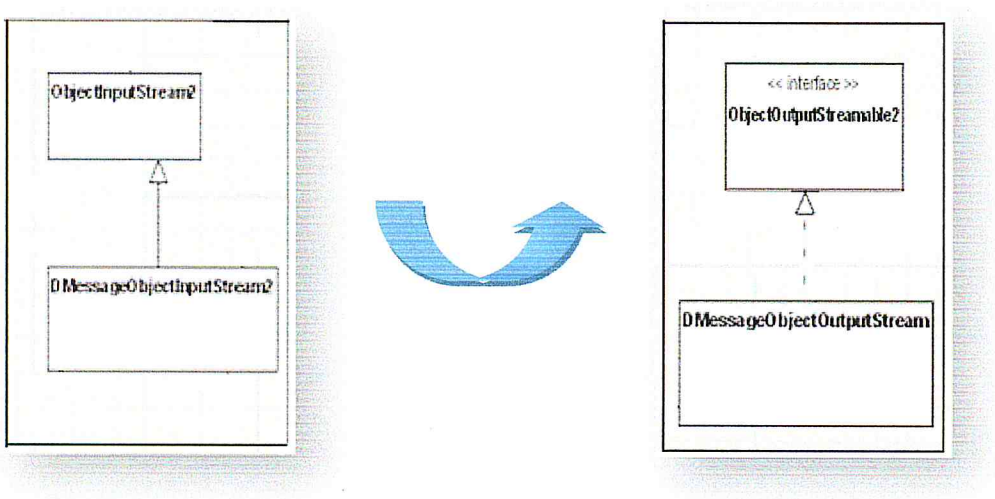
On remplace la classe *ObjectOutputStream*, par une interface *ObjectOutputStreamable* ; qui doit contenir les méthodes abstrais *writeObject(KSerializable objet)*, et *readObject()*. Les *Stream* spéciale pour chaque objet doivent implémenter cette interface, pour qu'ils puissent redéfinir ses méthodes.

Pour les méthodes *writeFlag(String flag)*, et *readFlag()*, on les déplace à l'interface *FlagOutputStreamable*, et *FlagInputStreamable*, et qui doivent être abstrait.

On peut poser la question, pour quoi on changer tous ça ?

On a à transformer l'héritage par la composition.

Par ce qui suit, nous schématisons, la transformation héritage en composition. Par exemple, dans le cas de la classe *DMessageObjectOutputStream* par exemple, cette transformation se fait comme suit:



Avant **Après**

Figure3.49 : déplacement de méthode de flag vers l'interface.

Mais, après le déplacement de la méthode concernant le flag vers l'interface, nous nous retrouvons face au problème de comment on va faire pour le crypter.

Pour résoudre ce problème, on a prévu une classe *FlagOutputStream* ; qui implémente cette interface, et qui a une référence sur le *EncryptedInputOutputStream*.

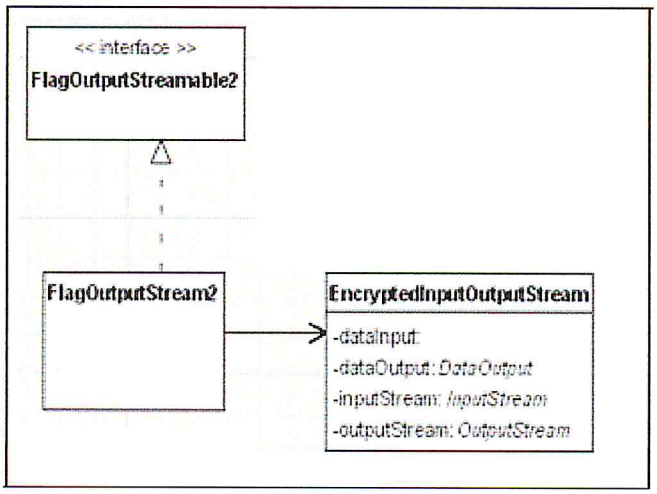


Figure3.50 : ajout de l'interface de flag.

4. Conclusion

Le *Framework* de sérialisation commence à devenir complet et générale, il a toutes les caractéristiques d'un *Framework* à base de patrons, mais on pense toujours à l'améliorer à chaque fois qu'on lui trouve une faille, et comme il a été conçu dans les règles de l'art de la maintenabilité, sa re-fabrication ou sa re-conception vont être assez aisées.

III.4 XML, MOBILITE ET PERSISTANCE

INTRODUCTION

On va présenter dans ce chapitre la re-conception de la mobilité de la base de faits de système expert, où on va utiliser ces notions : XML, Servlet, JDBC, et base de données.

1. La Migration de la base de faits du système expert

1.1. La sérialisation:

1.1.1 La vue statique:

Dans le cas de sérialisation de la base de faits, les choses diffèrent un petit peu, par rapport au cas de la migration des messages, par:

- ⇨ l'utilisation de la structure sous la forme d'un schéma XML,
- ⇨ de l'utilisation d'un serveur http,
- ⇨ de la migration vers un ordinateur,
- ⇨ de l'utilisation de la persistance de la base de faits, sous la forme d'un schéma relationnel,
- ⇨ de la communication *MIDlet*, *Servlet* et base de données relationnelle sous *postgresql*

Nous avons préféré, utilisé le schéma XML suivant, pour transformer une règle d'un objet de la classe *Rule* :

1) Le nom de la règle, par exemple :

   <Name>

   A

   </Name>

2) Le numéro du fait, par exemple :

<FactNumber>

1

</FactNumber>

3) Le nom du fait, par exemple :

<FactName>

</FactName>

4) Si le fait est vrai, par exemple :

<FactTruth>

true

</FactTruth>

Entre deux balises conteneur, on incorpore une sémantique concernant ce qu'on a voulu mettre entre les balises, et que nous voulons envoyer dans le flux qui lui correspond.

Cette structure est définit comme suit :

<Rule>

<Name>

</Name>

<Premisse>

<Fact>

<FactNumber>

</FactNumber>

<FactName>

</FactName>

<FactTruth>

</FactTruth>

</Fact>

</Premisse>

<Conclusion>

<Fact>

<FactNumber>

```
                    </FactNumber>

                    <FactName>
                    </FactName>

                    <FactTruth>
                    </FactTruth>
                </Fact>
            </Conclusion>
        </Rule>
```

C'est au niveau de l'écriture d'une règle dans le flux XMLRuleObjectOutputStream, que nous créons le String de représentation dans un schéma XML.

1.1.2 La vue dynamique:

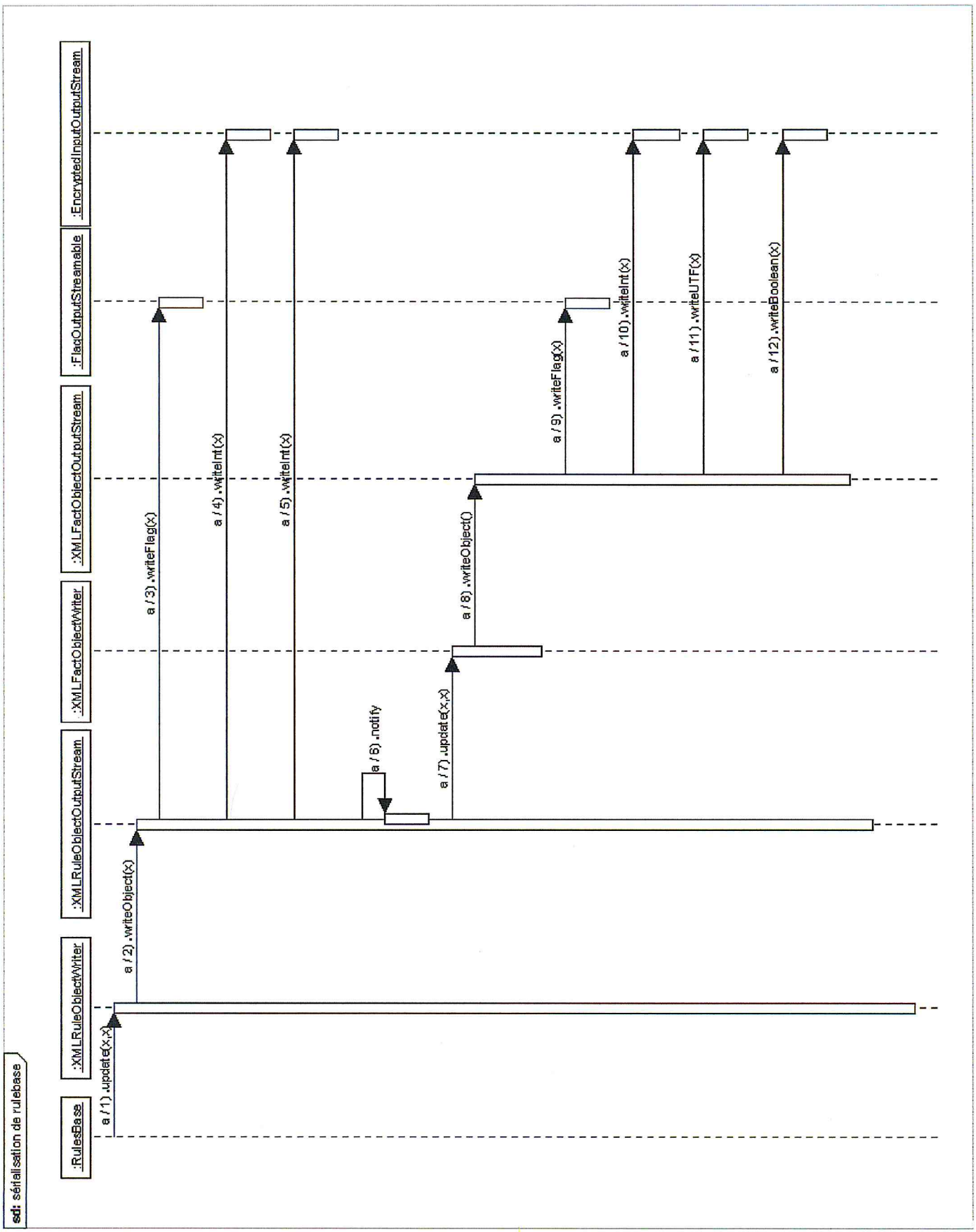


Figure 3.51 : le diagramme de séquence montranont la sérialisation de la base de faits et de règles

2. Le package xml

L'utilisation de XML dans la migration de règles nous a amener à rajouter un nouveau package à notre application, qui est le *package xml*. Ce dernier va donc contenir toutes les classes qui rentrent dans la sérialisation de la base de règles du système expert.

L'idée qu'il y'a derrière tout ça, c'est que lorsque un *To* migre, il devrait prendre avec lui son système de raisonnement, et qu'un téléphone portable n'est pas fait pour emmagasiner une quantité importante de données.

Parmi les solutions possibles pour ce genre de problème, c'est de permettre au téléphone portable de travailler en collaboration avec un ordinateur, qui va servir d'éléments de stockage partagée entre tous les environnements vers lesquels un *To* pourrait migrer.

Dans la mise en application de cette solution, nous avons utilisé le langage XML, pour structurer les Facts de la base de faits et les Rules de la base de règles, pour pouvoir les faire migrer à travers un Serveur http et les récupérer par des Servlets ou du Jsp de java, qui vont les parser sous forme de tables relationnels et les stocker dans une base de données.

Pour, mettre en application, cette solution, nous avons dû :

Confectionner plusieurs, comprenant entre autres :

- Toutes celle du package *xml*,
- celle qui sont dans le package *to* de *webapps* de *Tomcat*, le conteneur de servlets.

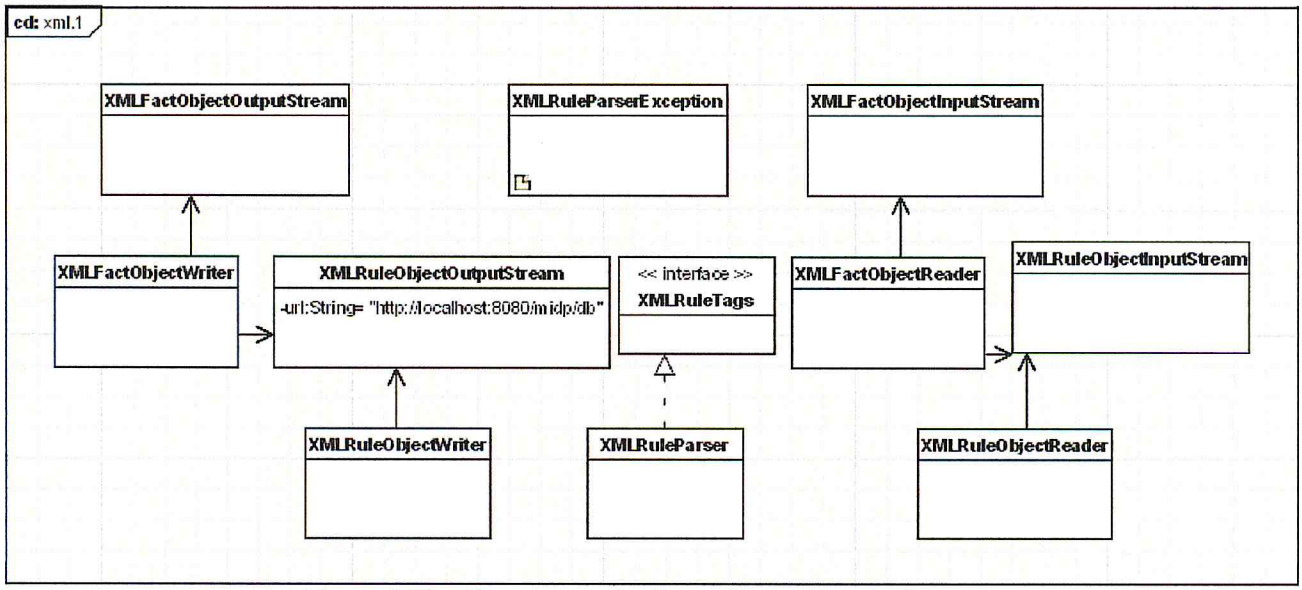


Figure3.52 : diagramme de classe du package xml.

2.1.Les classes coté *Sender* :

La classe *XMLRuleObjectWriter* :

C'est une classe *Observer* de *Sender*, si le flag qui été lu au niveau du *Sender* est la chaîne *Rule* , cette dernière va prendre en charge ce flux spéciale d'écriture d'une *Rule*, à qui il va demander d'écrire l'objet *Rule*.

La classe *XMLRuleObjectOutputStream* :

C'est le flux spécialisé pour la sérialisation (écriture) des règles *Rule* de la base de règles du système expert.

Comme une règle est composée d'une prémisse et d'une conclusion, et qui se sont en fait que des conteneurs de faits *Fact*.

Le traitement de sérialisation, consisterait donc à boucler une première fois, dans le cas des prémisses sur un *XMLFactObjectWriter* comme *Observer*, pour l'écriture des *Facts* de la prémisse, et une deuxième fois, dans le cas des conclusions sur le même *XMLFactObjectWriter* comme *Observer*, pour l'écriture des *Facts* de la conclusion de la règle. Ce traitement nous produit en plus un *StringBuffer*, dans lequel nous avons structuré la règle sous la forme d'un *String* image d'un document XML.

La classe *XMLFactObjectWriter*:

C'est un Observer de *XMLRuleObjectOutputStream*, il est notifié a chaque entrée dans la boucle de traitement de la sérialisation de la *Rule*. Il prend possession du flux pour l'englober dans *XMLFactObjectOutputStream*,

La classe XMLFactObjectOutputStream : c'est le flux spécial de la sérialisation d'un *Fact*, et la sérialisation ou l'écriture de ce dernier.

2.2.Les classes coté *Receiver* :

La classe *XMLRuleObjectReader*:

Si le flag lu à partir d'un *ObjectInputStream* est le String *Rule* la classe *XMLRuleObjectReader* va être le destinataire de la notification en vue de la lecture d'un *Object Rule*, à travers le flux *XMLRuleObjectInputStream*.

Ce dernier teste si le flux le est disponible à travers la méthode *available()*, il englobe ensuite le flux dans un *XMLRuleObjectInputStream* à qui il fait ensuite appel pour la désérialisation (ou lecture) des règles. L'opération, va ensuite notifier la base de règle *RuleBase* pour mettre à jour la base.

La classe *XMLFactObjectReader*:

Est un *Observer* de *Receiver*, il va notifier le *XMLRuleObjectInputStream,* à l'intérieur de sa boucle *for()*,le flux de lecture des *Facts*, le *XMLFactObjectInputStream*.

3. La persistance de la base de faits du système expert :

3.1.La base de données :

Le premier problème qui se pose, est bien sûr, de comment transformer et structurer un objet *Rule*, dans des tables relationnelles, c'est à dire, comment va-t-on s'y prendre.

Notre approche est un différente, dans le sens où, on a voulu conserver la structure d'arbre de du schéma XML de l'objet *Rule* dans les structres de tables du schéma relationnel, les relations père-fils de l'arbre vont être récupérées dans des colonnes prévues à cet effet, mais des relations de clef primaires et étrangères pour faire le lien de pointeurs dans les tables elles-mêmes.

D'après [Bou02], chaque balise conteneur ou composée sera transformé en une table, alors que dans la notre les choses se font un peu différemment : c'est seulement les balises composées qui vont être transformées en tables. Les balises conteneurs, ne seront, par contre elles, transformées qu'en colonnes dans des tables.

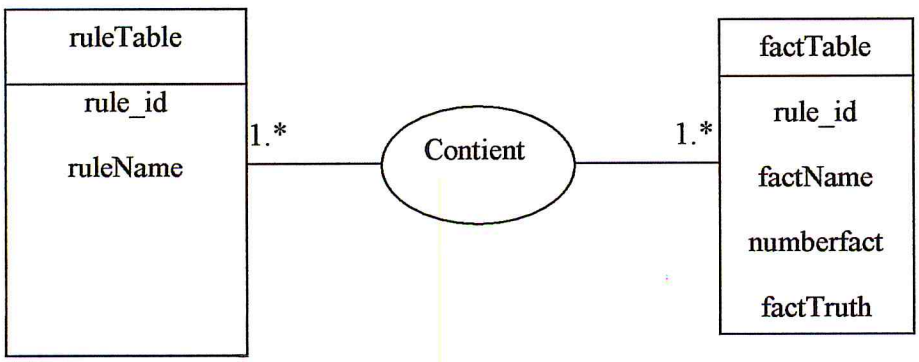


Figure3.53 : les tables de la base de données.

Les tables sont crées dans la base de données par le SQL, comme suit :

```
Create table ruleTable (rule_id int NOT NULL, rulename varchar(32), PRIMARY KEY(rule_id)) ;
```

Create table factTable(rule-id int NOT NULL, factName varchar(32), numberfact int, facttruth boolean, foriegn key(rule_id) refernces ruletable) ;

Le choix de JDBC est personnel, et c'est le PostgerSQL, qui nous a permet de faire des interrogations avec la base de données.

Le processus entier, va s'effectuer comme suit :

1) Le String XML transformé de la Rule en XML va s'effectuer lors de l'écriture dans le flux de la règle. Ensuite,

2) nous allons faire appel à un serveur http, le *HTTPServer* qui va créer une connexion http avec une *Servlet*, la *XMLServlet*,

3) cette dernière, à la réception de la requête, va demander un *parsing* de la *String* reçue en un objet *Rule*, ensuite

4) elle va se connecter avec la base de données, nommée *xmldb*, via le SGBD *PostgresSQL*, pour insérer les règles dans les tables prévues à cet effet dans cette dernière.

Un schéma de représentation général est définit comme suit :

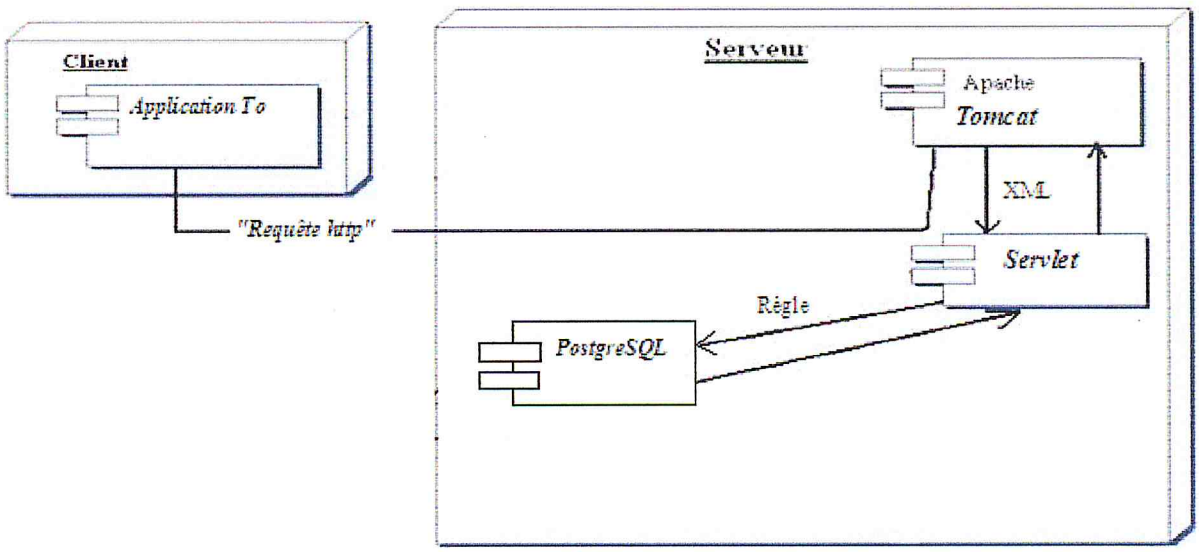


Figure3.54 : l'architecture 3 tiers

La Servlet va donc recevoir, à partir du serveur *HTTPServer* le String XML, elle va :

- le passer au parseur XML qui, en fait, n'est qu'un objet analyseur syntaxique xml spécial, qui permet de convertir le String XML du schéma xml d'une règle en un object *Rule*,. Ensuite, une fois l'Object *Rule* obtenu,
- la Servlet va demander de le persister dans les tables relationnelles prévues, à cet effet..

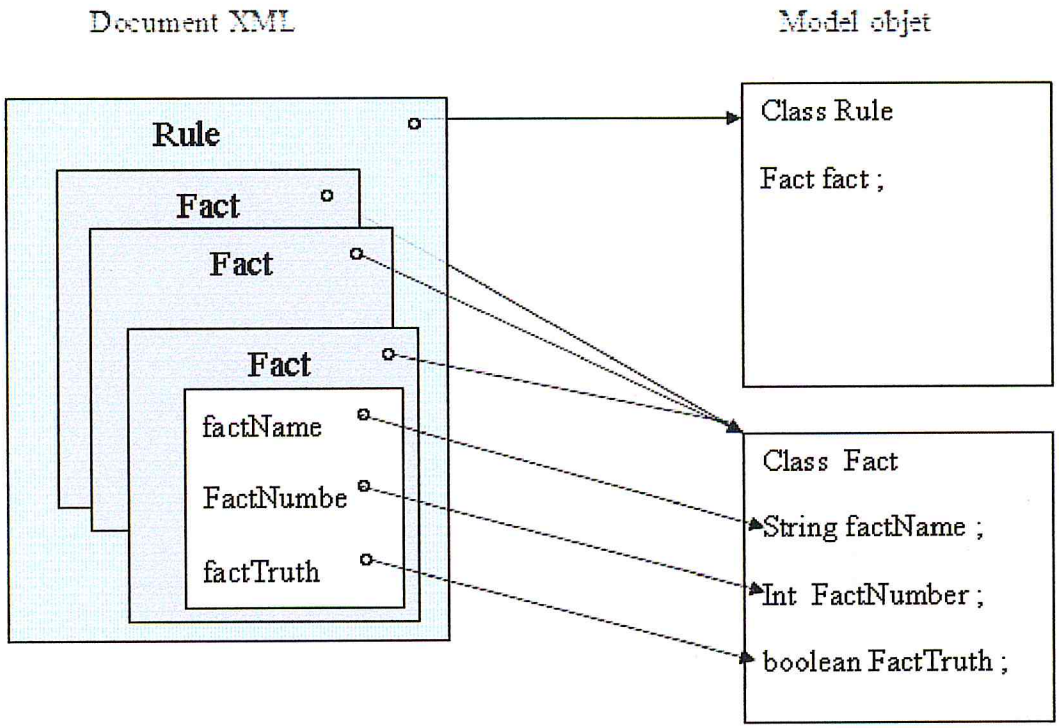


Figure3.55: la transformation de schéma XML au model objet.

Pour l'inserer dans la base de données par la méthode de la classe XMLDB :

```
public void insertRule(Rule rule) throws SQLException{
...
}
```

Cette méthode elle permet d'insérer une règle(Rule) dans la base de données après l'ouverture de la connexion

Pour notre cas on n'avait pas utilisé les Statement :

```
//Statement statement= connection.createStatement();
```

Mais plutôt des PreparedStatement

```
//PreparedStatement ps = null;

ps = connection.prepareStatement("INSERT INTO ruleTable VALUES (?, ?)");

ps.setInt(1,ruleID );

ps.setString(2, rule.getRuleName());
```

Le diagramme de classe suivant montre tous ça

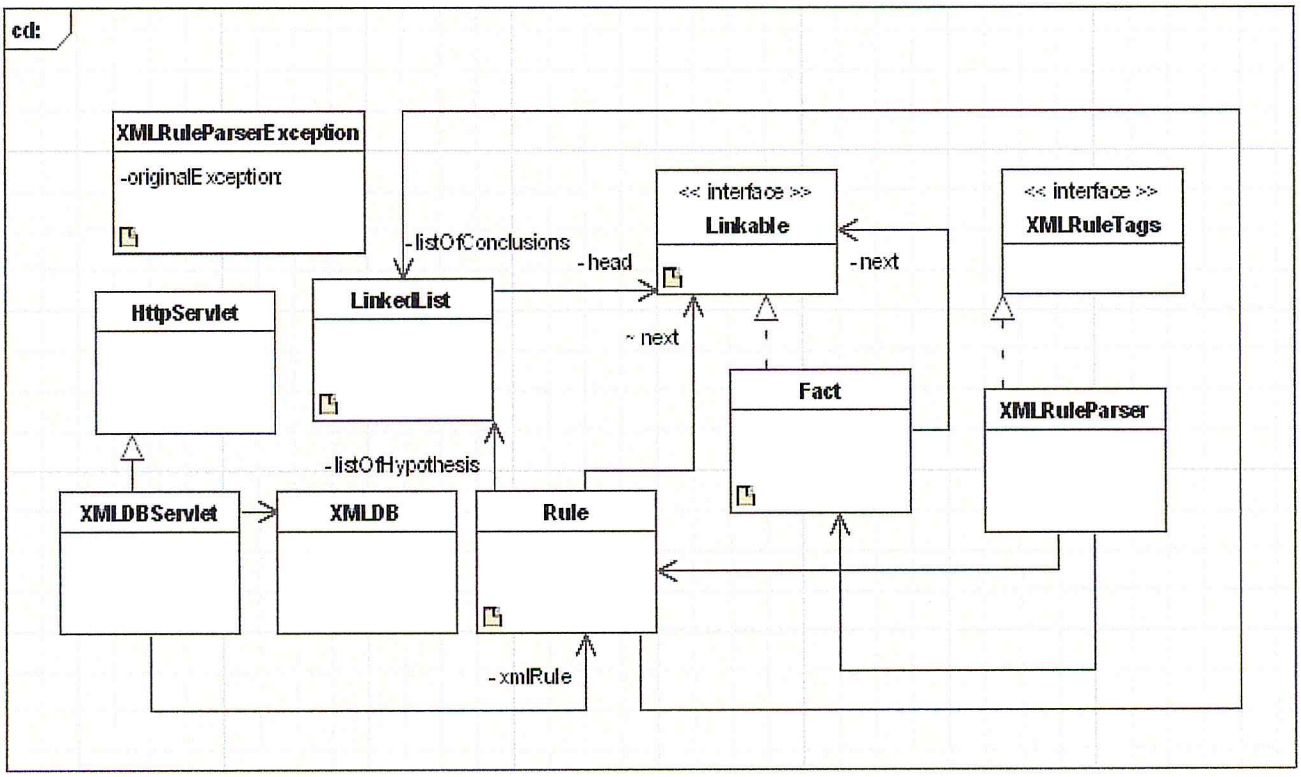


Figure3.56 : diagramme de classe de package to de webapps.

4. Conclusion :

Ce chapitre nous a permit :

- ⇨ d'employer le schéma XML, pour faire migrer des règles de production d'un système expert d'utiliser vers un serveur conteneur de Servlet.
- ⇨ D'écrire un parseur et l'utiliser pour parser le schéma XML de la règle en schéma objet,
- ⇨ De transformer le schéma objet au schéma relationnel et de stocker les règles préservé dans un format qui maintient la structure d'arbre de schéma XML et réduit les tables au niveau relationnel.
- ⇨ D'utiliser ce schéma relationnel pour faire des interrogations SQL et de les

III.5 INTERFACES GRAPHIQUES

Les différentes optimisations réalisées pour les affichages des vues :

1. Introduction

Y'avait certaines vues d'affichages, qu'on a modifié, et d'autres on les rajouter ; et qui ne sont pas existe déjà.

2. Présentation de vues

Le diagramme de classe suivant permet de montrer les classes de vues qu'on a rajoutées :

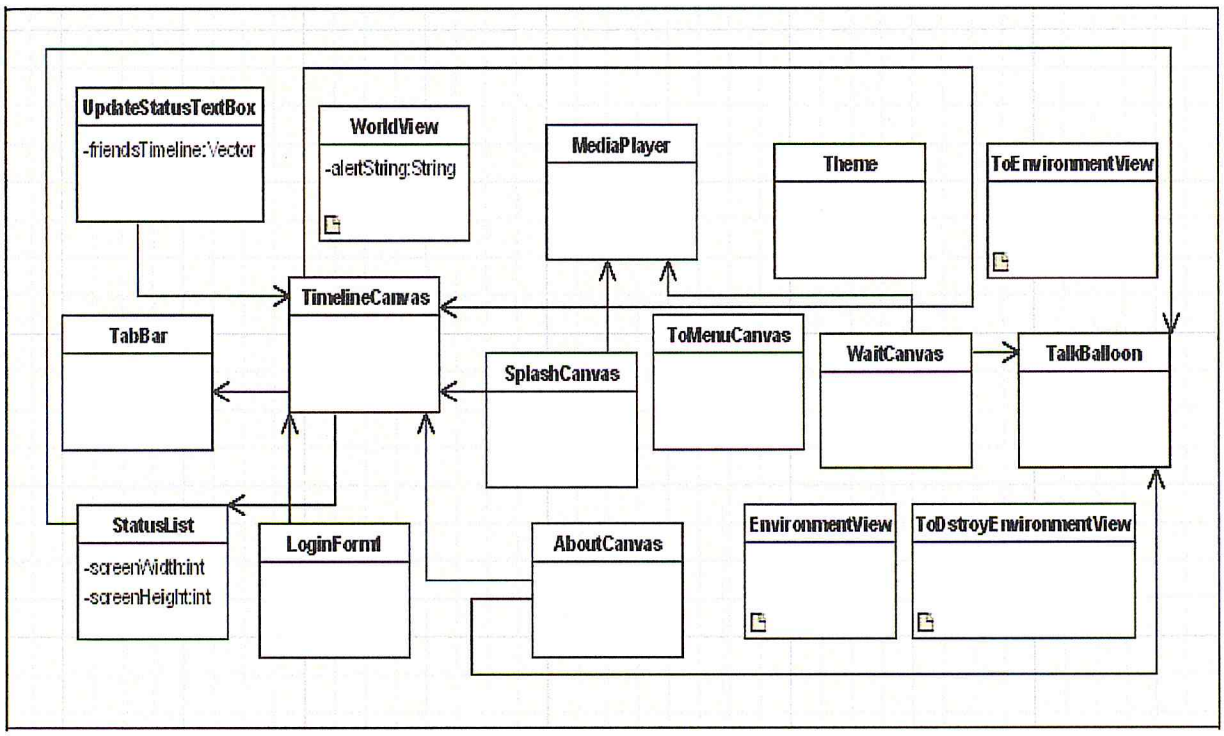


Figure3.57 : diagramme de classe de vue rajoutées.

La classe *AboutCanvas :* c'est un canvas, qui permet d'afficher sur le display les informations sur le développeur de ***To***.

La classe *SplashCanvas :* c'est la fenêtre principale de cette version ; il s'agit de mettre un logo pour To ; au lieu de mettre la classe ***SplashScreen***[1]

La classe *LoginForm :* permet d'écrire un username et un password

Qui permet de choisir l'un d'eux. Cette fenêtre de menu été existe avant dans To ; mais de faire des améliorations, c'est aussi mieux.

Ce qui nous donne la nouvelle classe qui s'appelle ***ToMenuCanvas*** au lieu de la classe ***Menu*** ;

La classe *ToMenuCanvas :* c'est une Canvas qui permet de mettre des retours à chaque fois au menu principale

La classe *ToEnvironmentView :* est Observer de l'environnement ; qui permet d'afficher l'agent (par nom) dès sa création.

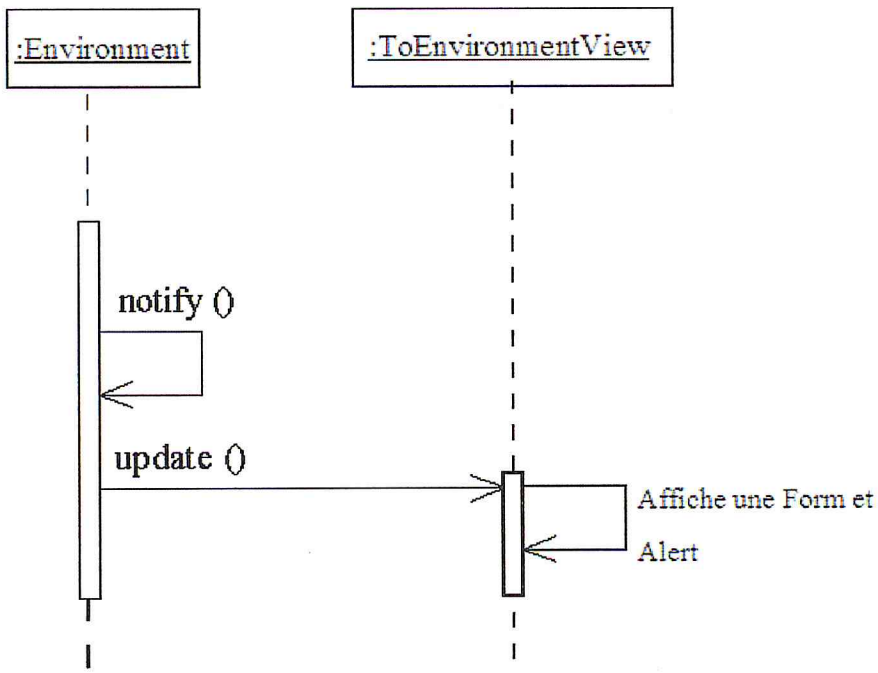


Figure3.58 : diagramme de séquence montrant la vue ToEnvironmentView.

La classe *EnvironmentView :* cette classe elle existe déjà, juste qu'on a l'utiliser dans le cas de migration d'un agent ; et de remplacer la vue de la création locale de l'agent par la classe *ToEnvironmentView*

[1] Voir sous-chapitre III.1

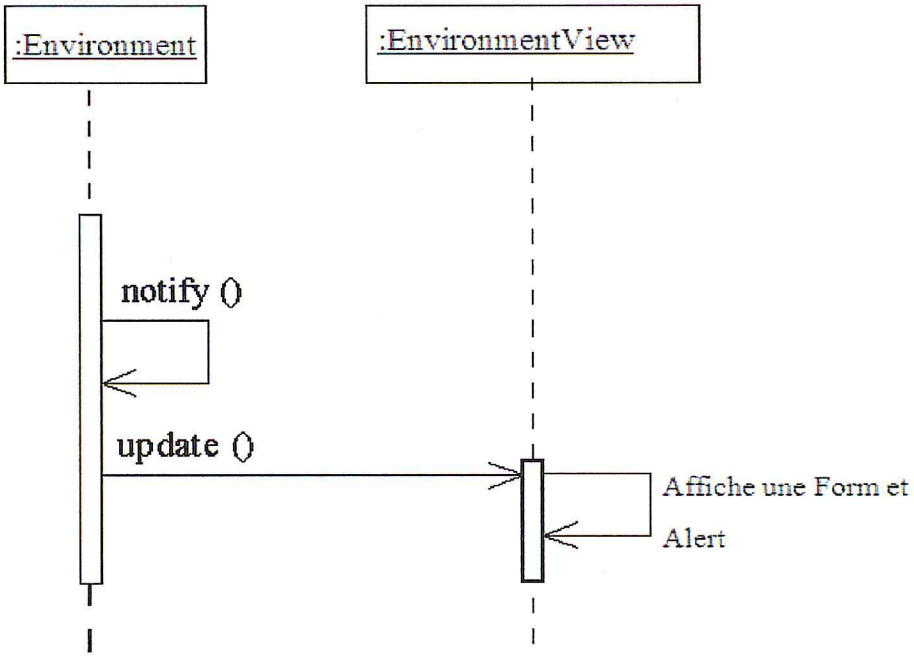


Figure3.59 : diagramme de séquence montrant la vue EnvironmentView.

La classe *ToDestroyEnvironmentView* : cette vue joue un rôle d'un *Observer* du *l'Environment* ; elle sera notifié par *l'Environment* lorsque il supprime l'agent, cette dernière elle nous affiche le nom de l'agent supprimé.

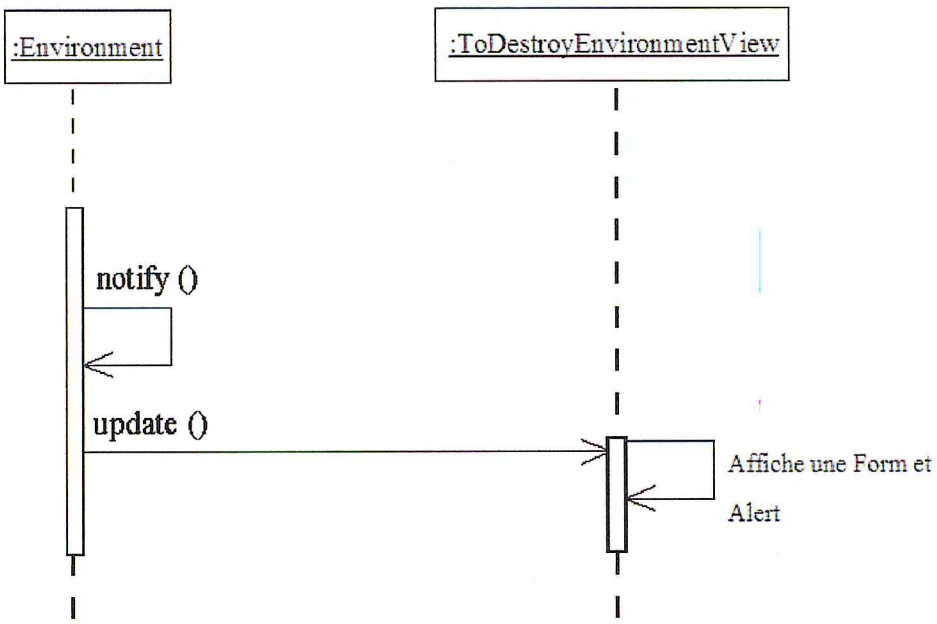


Figure3.60 : diagramme de séquence montrant la vue ToDestroyEnvironmentView.

La classe *WorldView* : cette vue joue un rôle d'un *Observer* du *World* ; qui nous donne des informations sur ce qui se passe dans le world, en cas de changement il le notifie, tous qui concernes les messages des agents. Avec une simple *Alert* qui apparaître.

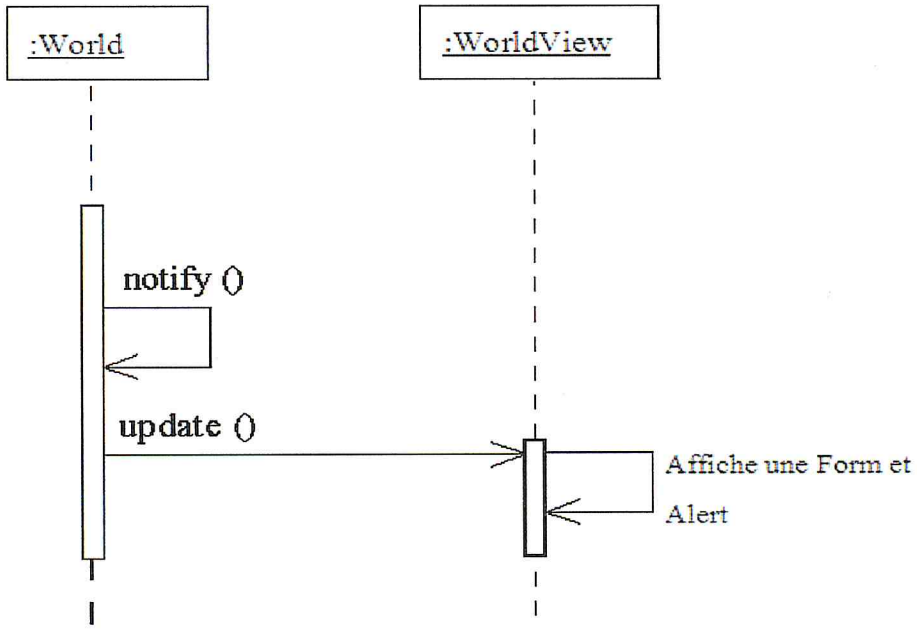


Figure3.61 : diagramme de séquence montrant la vue WorldView.

Au cours du développement, on a rencontré à chaque fois de la non existence des vues qu'on a déjà fait ; nous a fait un problème au niveau de performance ; et c'était à cause de l'utilisation des threads qui sont au même temps des observer/observable, est qui interrompe l'affichage de plusieurs veux.

La solution qu'on a adopté consiste à mettre une liste[2] , pour toutes les displays qui ont besoin de récupérer la MIDlet (par le patron Observer/Observable) ; ç.-à-d on l'insère dans la liste quand il récupère cette dernière (la MIDlet), comme suit :

[2] C'est la même List qui été utilisé dans le développement de To.

```
public Displayable getPreviousDisplayable(){
 return((LinkableDisplayable)(displayableLinkedList.getAt(getPrevious()))).getDisplayable();
}
public void saveDisplayable(Displayable displayable){
      LinkableDisplayable linkableDisplayable= new LinkableDisplayable(displayable);
      displayableLinkedList.insertAtTail(linkableDisplayable);
      }
```

3. Conclusion

dans ce chapitre nous avons présenté l'ensemble des vues qu'and rajouté dans l'application, en vue qui ont nous aidées surtout dans les tests. Pour afficher une vue, il faut passer par toutes les classes de Framework de mobilité.

III.6 TESTS

1. INTRODUCTION

Les tests dans la maintenance de logiciel sont effectués durant le développement, il va permettre la correction de défauts existants, et ceux découlant de l'ajout de fonctionnalités.

Lors du développement de notre système nous sommes passés par plusieurs étapes, et dans chacune d'elles, nous avons dépensé un temps plus important que celui du développement à faire des tests.

Nous avons fait des tests de :

- réalisabilité,
- validité de l'architecture,
- unitaires,
- d'intégration,
- de regression,
- performance,
- …

Nous avons vraiment compris que sans tests, il n'y a pas de re-conception, ni de re-fabrication, ni donc de maintenabilité.

2. COMMENT A ETE TESTE TO

Dans cette partie nous allons exposer un certain nombre de tests faits pendant le développement pour la maintenance du framework *To*.

Pour les tests unitaires nous avons choisi d'utiliser une simple classe ***Assert*** qui fait des cas de tests ; comme celle de *JUnit*. Si le test est échoué, une exception se lance ;

Pour l'affichage le traçage de l'exécution du programme, nous avons confectionné une classe ***Log*** ; pour qu'on suivre le déroulement de l'application sur l'émulateur.

Cette classe, avec ses méthodes temporelles, qui affichent l'heure, ou ses méthodes qui insèrent des lignes vides pour facilement localiser les erreurs, elle nous a était d'un grand secours.

Le diagramme de classe qui suit montre les classes utilisées pour faire les tests durant le développement

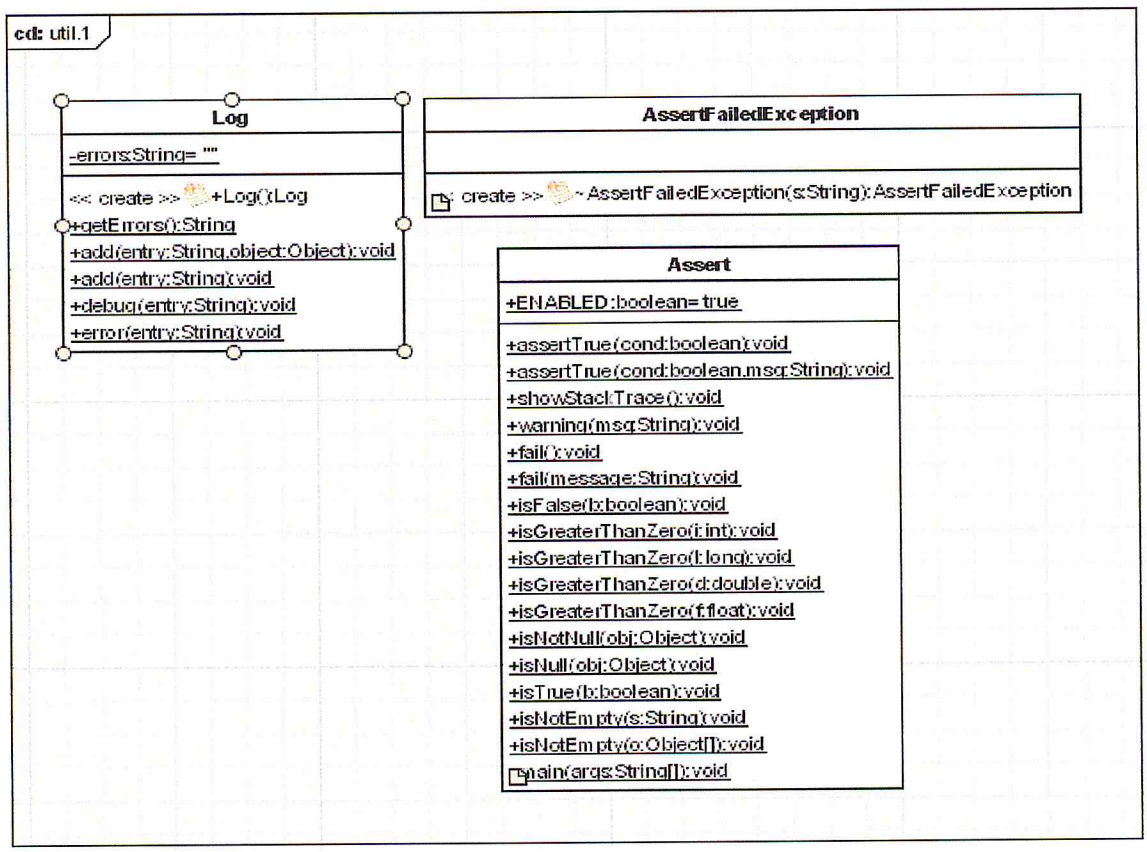


Figure3.62:diagramme de classe montrant l'ensemble de classe de tests.

Ce qui fait d'ajouter ces classes dans le package *util*

3. TESTS UNITAIRES

3.1. Test la serialization de l'objet X:

Le diagramme de classes général des classes de ce test est le suivant:

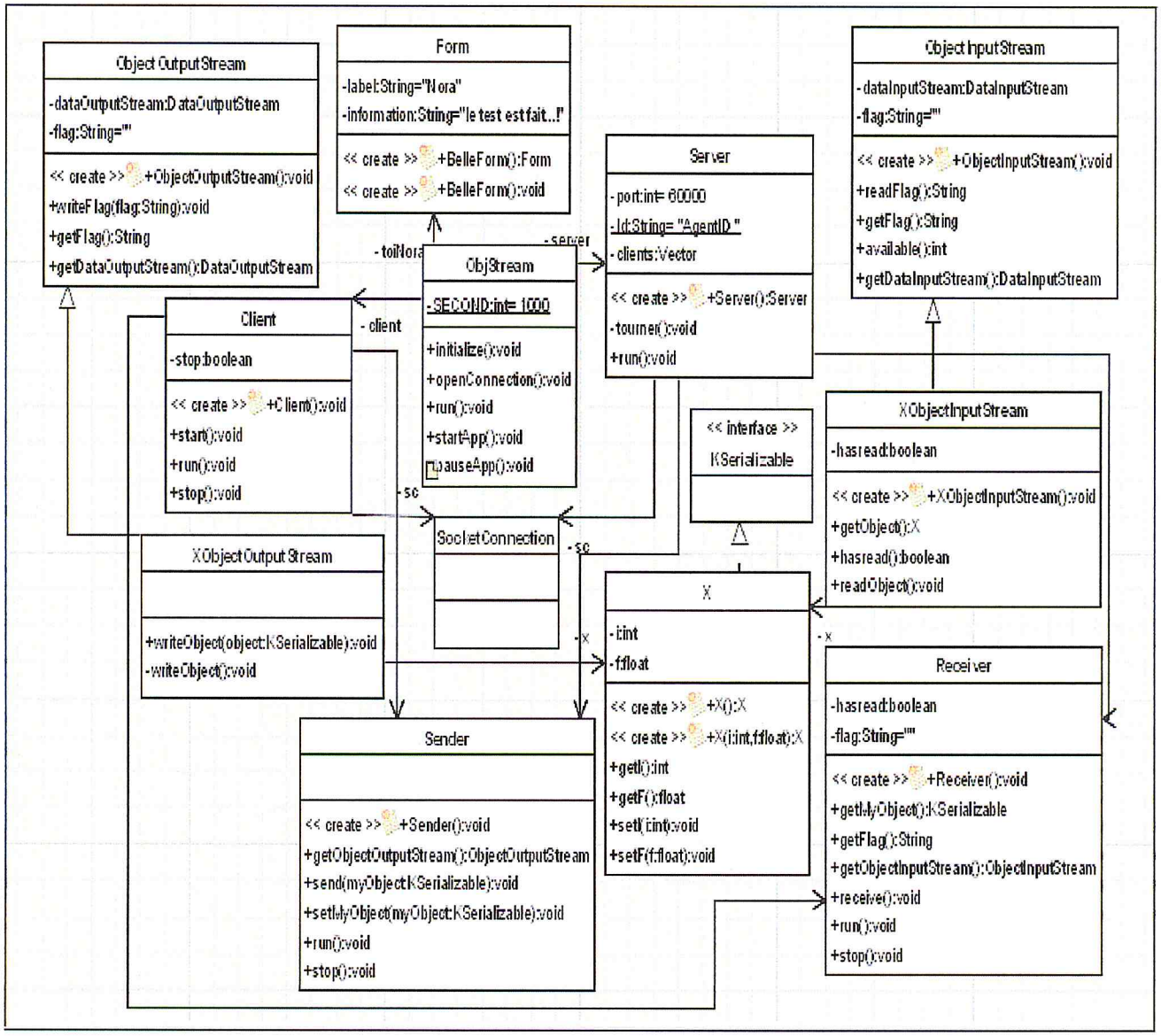


Figure3.63: diagramme de classe de test de la sérialisation.

L'objet qu'on veut envoyer dans le réseau il doit implémente l'interface *KSerializable* ; vide, qui au fait, n'est un englobant général de tous les objets variés qu'on voudrait sérialiser, comme le montre la figure suivante.

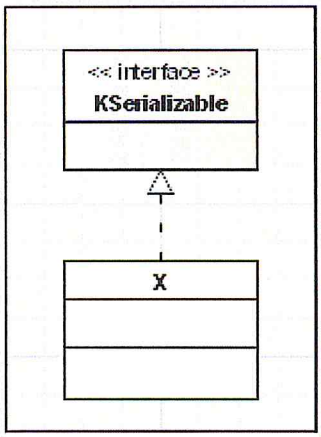


Figure3.64 : l'implémentation de l'objet X.

On va adjoindre à l'objet X ses flux, qui sont *XObjectOutputStream*, et *XObjectInputStream*, et qui doivent hériter des classes *ObjectOutputStream*, et *ObjectInputStream*, comme le montre la figure suivante.

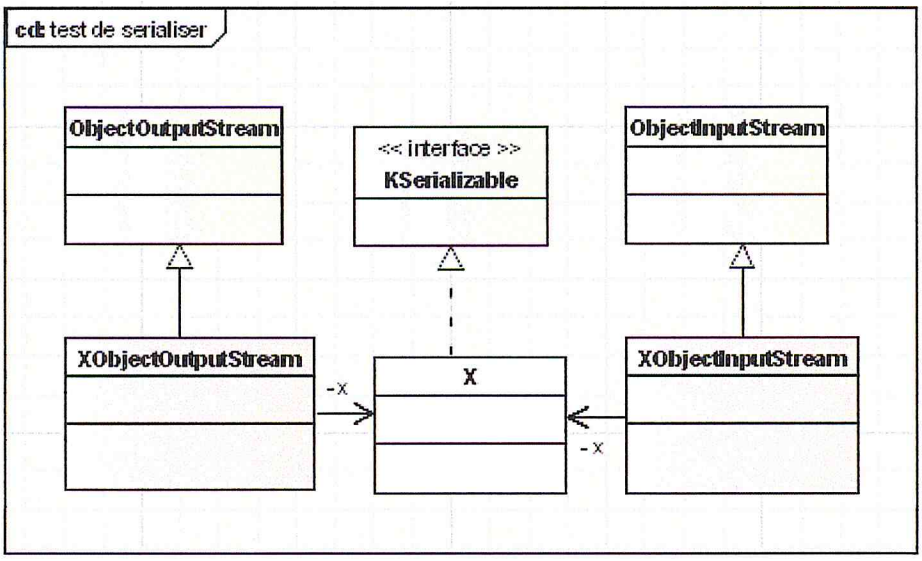


Figure3.65 : diagramme de classe montrant la sérialisation d'un objet X.

Cela reste insuffisant, tant que nous n'avons pas créé une connexion Client/serveur pour tester l'envoi de l'objet X sur le réseau après l'avoir sérialisé. Pour le faire, nous avons créé

un serveur particulier pour les besoins de la chose, c'est dire autre que celui que nous avons développé plus tard pour To, en utilisant une connexion par Sockets.

Nous allons, dans ce qui suit, présenter les classes de ce test unitaire :

La classe *Client* :

Il ouvre une connexion, avec le server par une même adresse de host et de port de ce dernier

La classe *Server* :

C'est un thread avec un numéro de port spécifique. Ce thread tourne sans arrêt à la recherche d'une connexion client,

Le diagramme de classes suivant montre, à travers une connexion client/Serveur, les relations entre ces classes:

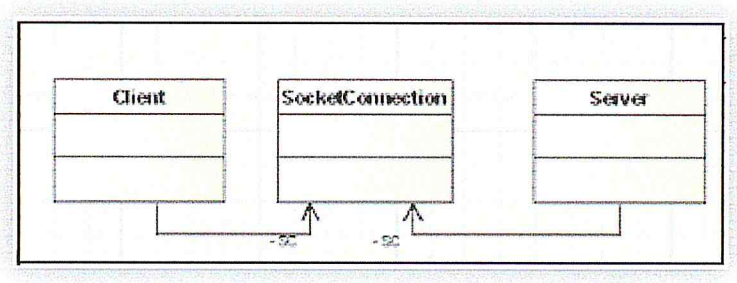


Figure3.66 : diagramme de classe montre la connexion Client/serveur.

Le diagramme suivant, à pour but de relier, toutes les classes précédemment citées de l'application:

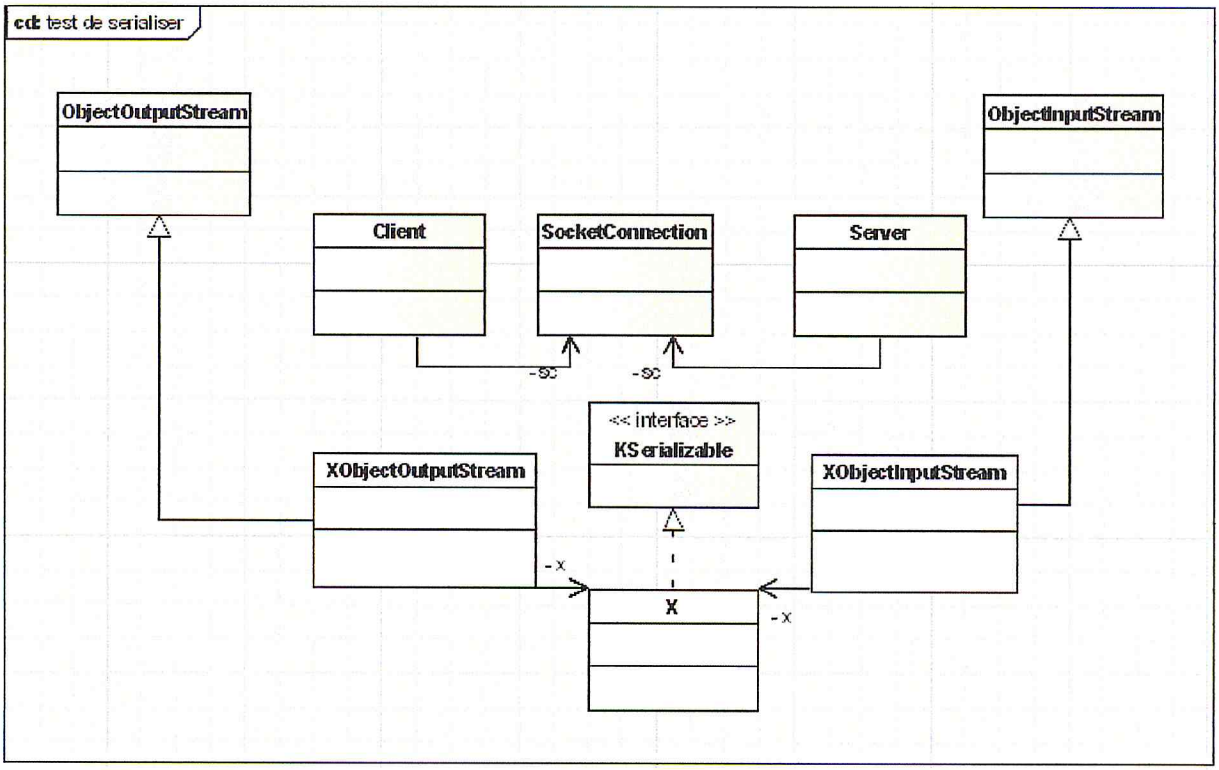


Figure3.67 : montre le diagramme de classe pour le test de sérialisation.

Il reste d'ajouter des délègue aux Client/serveur, c'est-à-dire l'envoyeur, et le receveur de l'information, et qui sont les classes :

La classe ***Sender*** et ***Receiver*** :

Sont les délègues de communication entre le Client et le Server,

Comme le montre le diagramme de classe générale de notre Framework de sérialisation suivant :

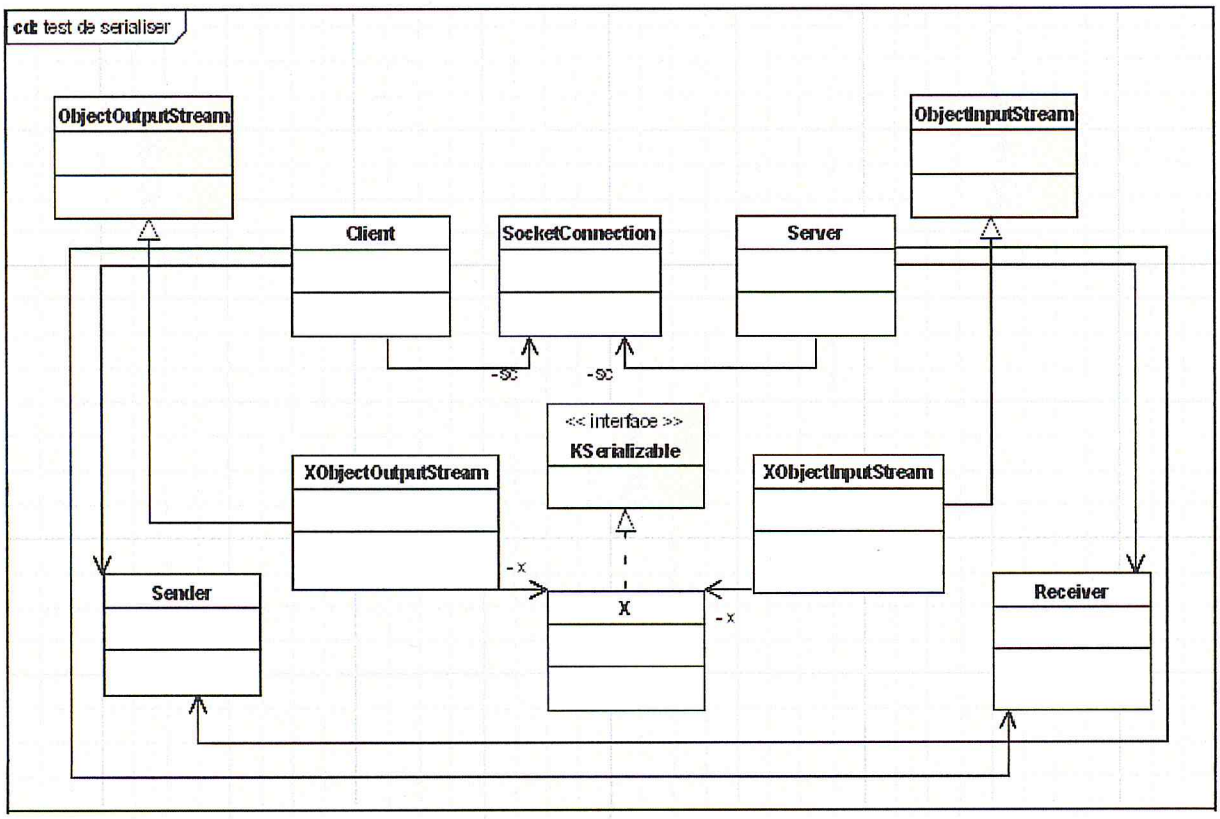


Figure3.68 : diagramme de classe de la sérialisation d'un objet de la classe X

Pour instancier ce *Framework*, on doit créer une *MIDlet*[1] nommée *ObjStream*, qui n'est qu'une simple *Form*, qui permet juste d'affiche l'objet qui a été envoyé sur réseau, après sa sérialisation.

Le diagramme, ci-dessous, donne une vue de relation statique entre les classes ayant servies dans ce test :

[1] Voir sous-chapitre II.1.

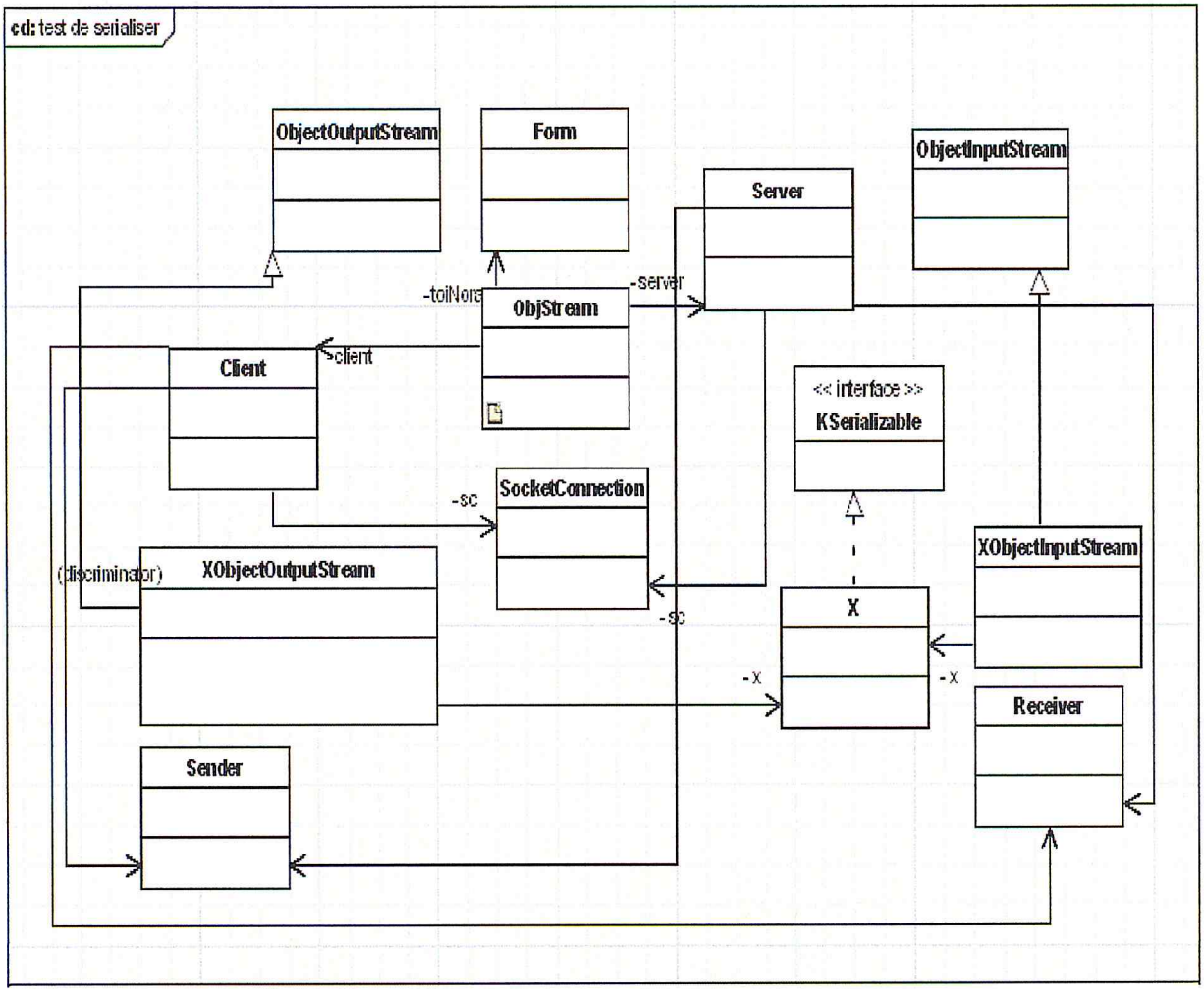


Figure3.69: diagramme de classe de l'implémentation de Framework de sérialisation.

Remarque :

La classe X, n'est qu'un objet simple, qui a des attributs de base, comme *int*, et *float* ; et dans ce cas la sérialisation n'était pas très simple, mais du moins faisable; mais il va y a voir d'autres problèmes à résoudre quand il va s'agir de sérialiser des objets plus complexes, et l'opération sera nécessairement beaucoup moins simple..

3.2. Test unitaire de l'algorithme de cryptage :

Dans certains cas, comme celui du test unitaire de l'algorithme de cryptage de César, on a tout d'abord réalisé le test sous java standard, malheureusement, dans le cas du test d'intégration, on a eu un problème concernant la méthode *replace(int, int, String)* de la classe

StringBuffer, qui n'existe pas sous j2me, et dont nous nous sommes inspiré pour écrire une méthode semblable, dans notre application.

4. TESTS DE VALIDITE DU LOGICIEL

Pour les tests de validité de logiciel, nous avons à chaque fois construit, un petit environnement, pour la réalisation de l'opération à tester, dont nous nous défierons juste après la réalisation du test, ce fut le cas du test de la sérialisation de la mobilité et de sérialisation d'un objet d'une classe générale, que nous avons appelé classe X.

5. TEST D'INTEGRATION

Ce type de test est réaliser, au début quand on a réalisé le pseudo sérialisation ; pour tester son intégration dans le système, et que l'intégration est correcte, stable, et se fait de manière cohérente avec le système existant.

6. TEST DE REGRESSION

On a choisi ce type de test, dans les cas suivants :

- le cas où on ajoute de nouvelles fonctionnalités (re-conception),
- ou bien de nouvelles classes,
- ou bien effectuer des re-fabrication.

Car à chaque fois que nous avions rajouté des nouvelles classes, on a rencontré de nouveaux problèmes dus à l'intégration du nouveau composant dans le logiciel existant, et souvent des centaines d'erreurs, qu'il faudrait corriger, vont apparaître.

On teste pour assurer que la partie existante reste valide, et continuera à fonctionner, avec les mêmes fonctionnalités, sinon on reprend le système pour localiser d'abord la source d'erreur, et d'essayer ensuite d'opérer des modifications, là où fût localisé le problème, en vue de la correction des erreurs.

À chaque fois que le logiciel est modifié, il faudrait s'assurer que "ce qui fonctionnait avant fonctionne toujours"

7. CONCLUSION

Les tests permanents nous ont permis de répondre aux besoins de manière dynamique et de consolider certains choix entrepris.

CHAPITRE IV

PRESENTATION DE L'APPLICATION.

IV. PRÉSENTATION DE L'APPLICATION

1. Introduction :

Nous avons choisi de donner une vue sur ce qu'il a été fut dans la version 1.0 de ***To*** comme tests

2. La version 1.0 de To

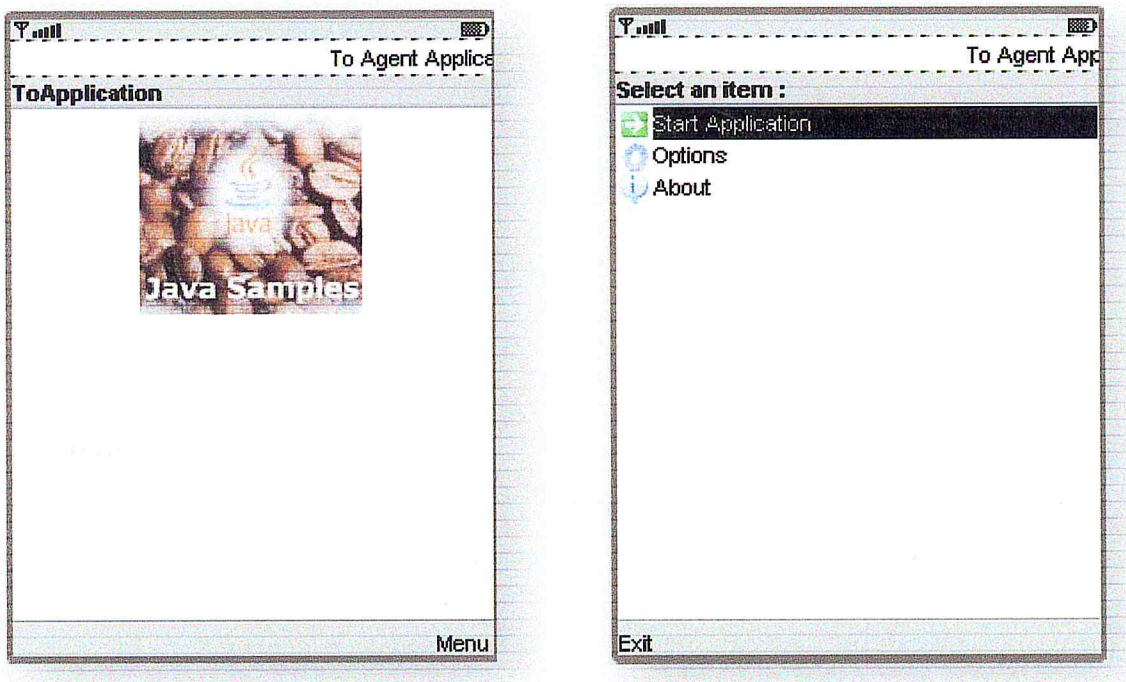


Figure4.1 : les vues d'affichage après le lancement de l'application.

L'application de Framework (instantiation) dans l'ancien version, est très simple test qui demande à un agent d'exécuter une méthode, par poste un KMessage à un agent spécifiant son nom.

On suit le traçage de ce test, dans l'émulateur, comme le montre la figure suivante

l'émulateur affiche le traçage de l'pplication de Framework, avec des appels à la méthode *System.out.println()*

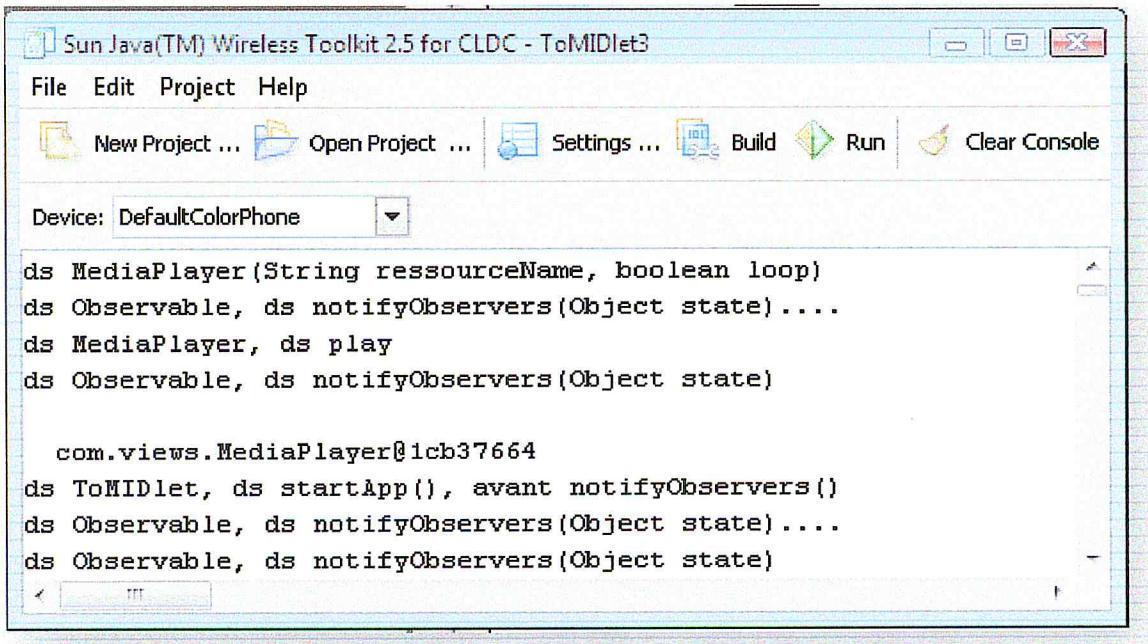


Figure4.2 : l'émulateur d'affichage le traçage de l'application.

Après le lancement de l'application le système expert est lancé automatiquement par le framework sur la base de la base de faits et la base de règles rattachées dans l »application, à un To, pour définir son système de raisonnement, après sa création, pour qu'il puisse faire du traitement « raisonnée » de messages, comme le montre la figure suivante :

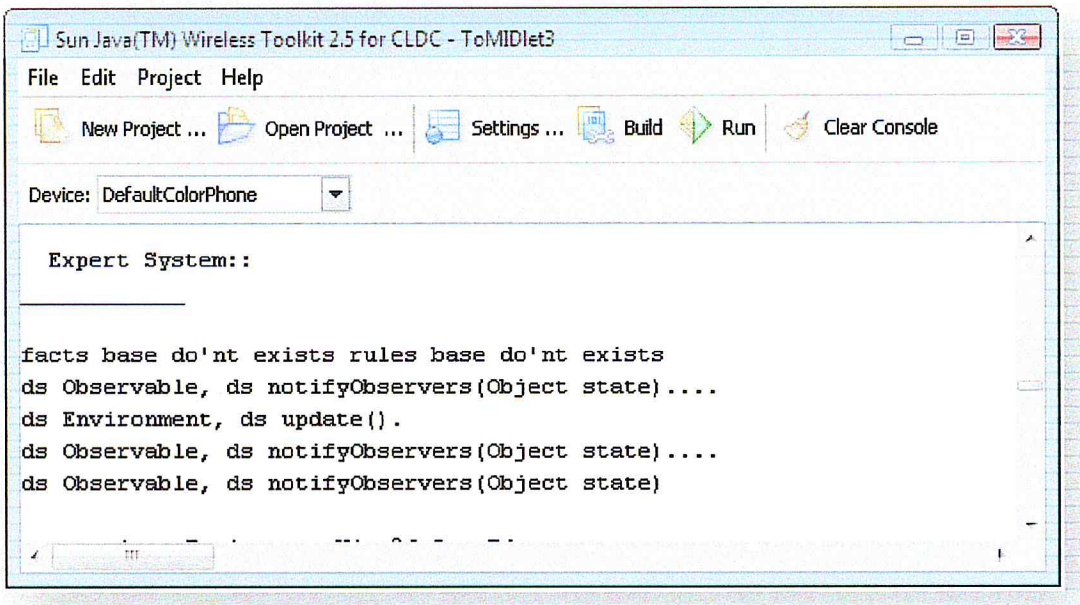


Figure4.3 : l'émulateur affiche le lancement de system expert

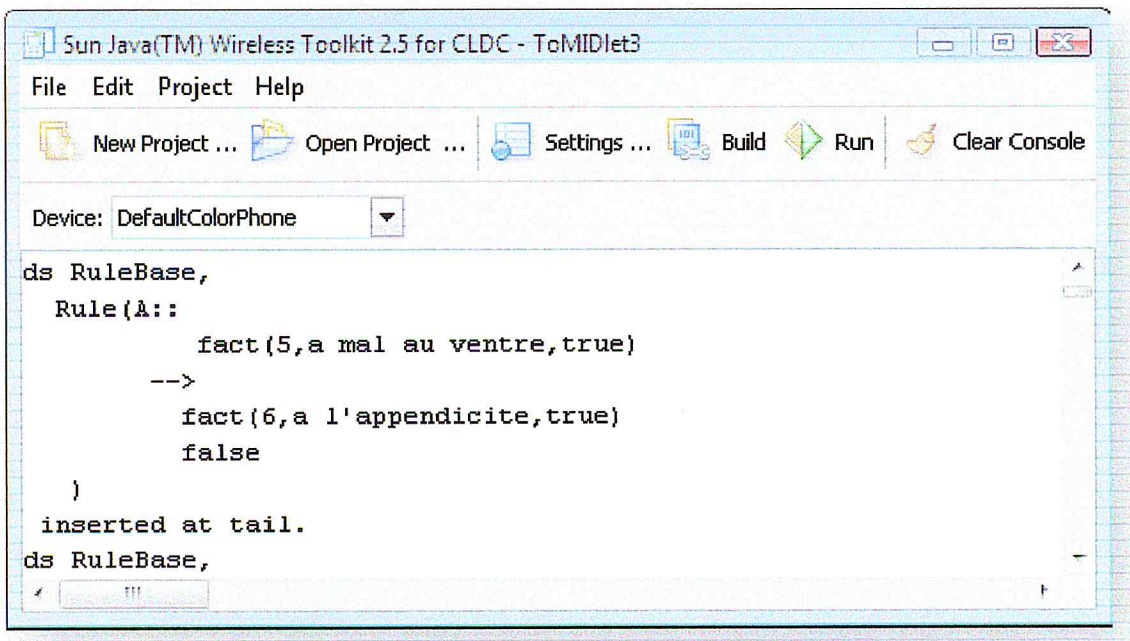
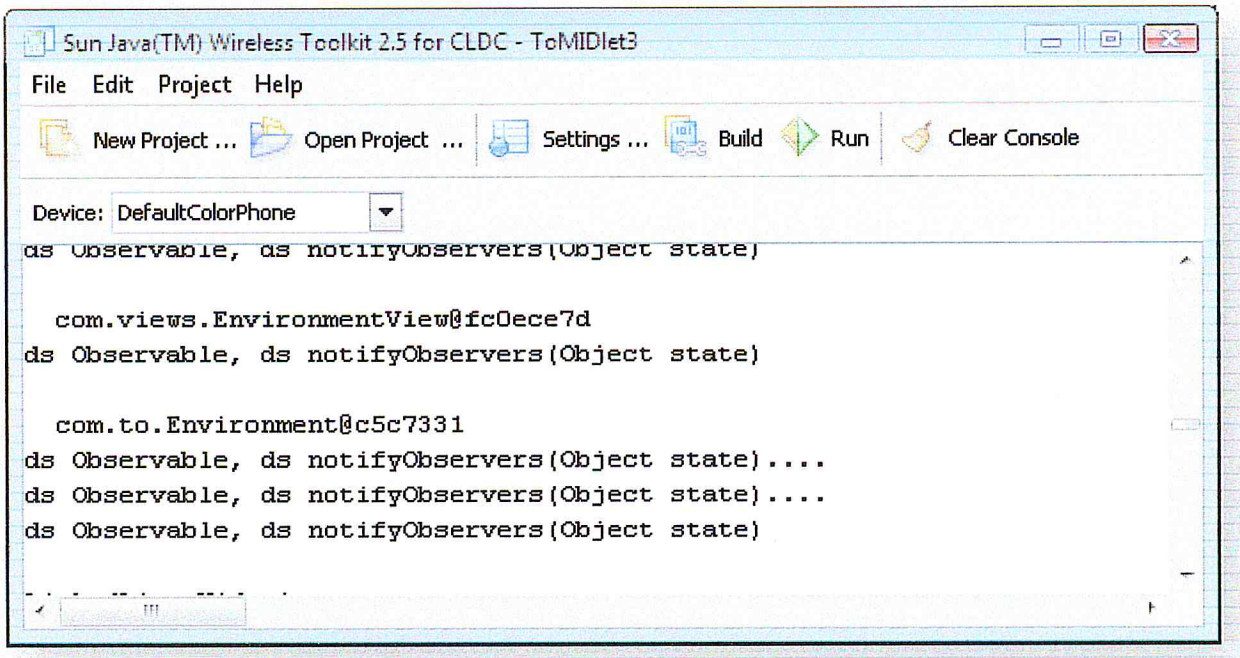


Figure4.4 : l'affichage de la base de règles

On suit toujours l'émulateur pour voir le traçage de l'application :



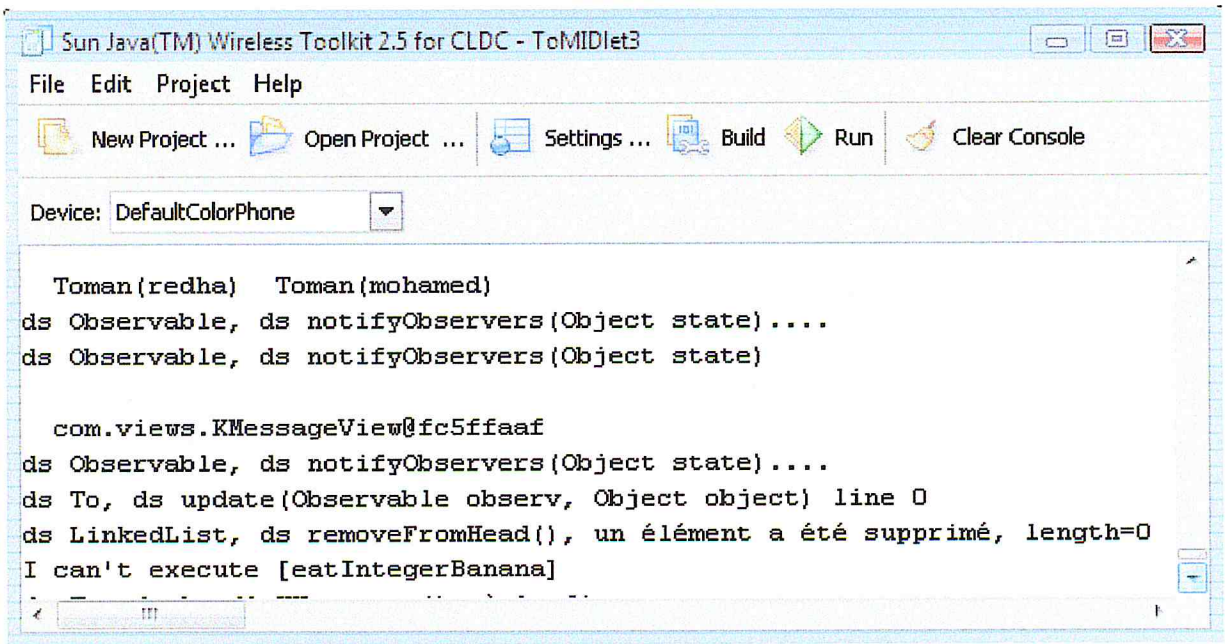


Figure4.5 : l'affichage des agents crées dans Environment.

Avec la notification de la vue que le message *KMessage* est exécuté, il y'aurait affichage de la vue KMessageView (les vues sont des Observers), donnant ce qui suit :

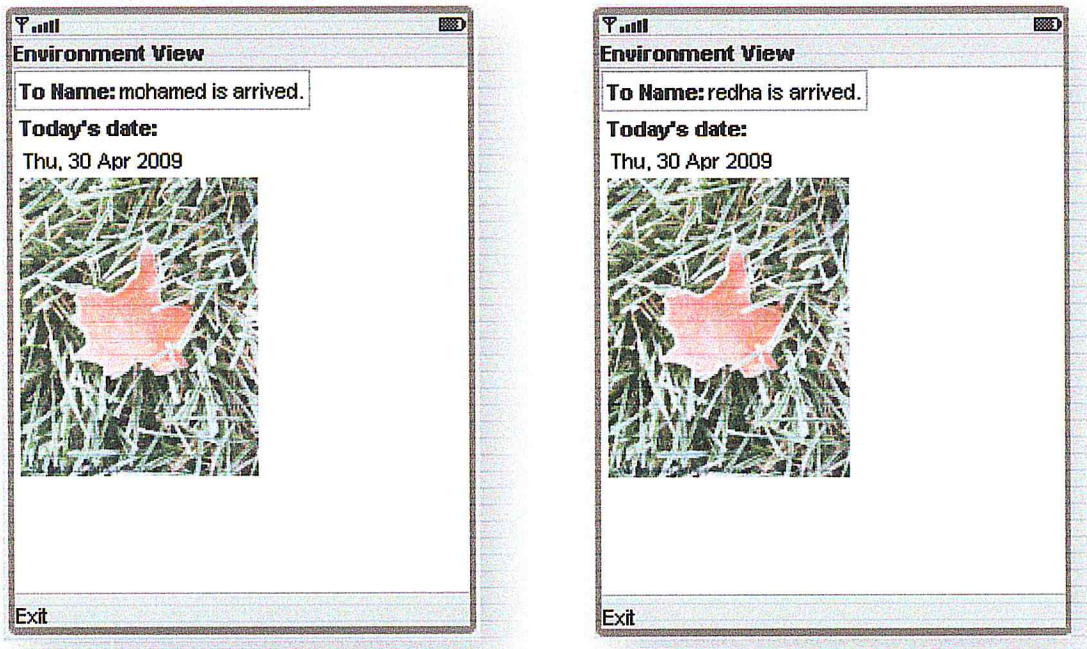


Figure4.6 : les vues d'affichages

On voie le résultat de la vue (KMessageView) dans l'émulateur tel qu'il apparaît dans cette impression d'écran.

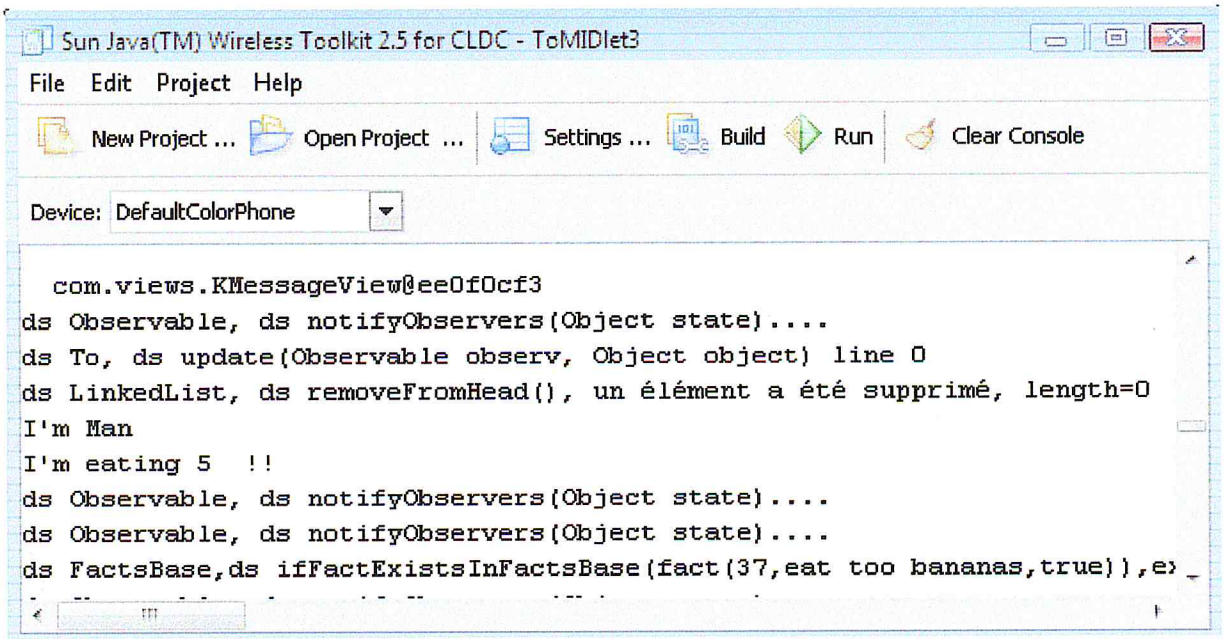


Figure4.7 : l'émulateur affiche l'exécution de message KMessage.

Au même temps on peut le voir, à tavers l'affichage dans la MIDlet

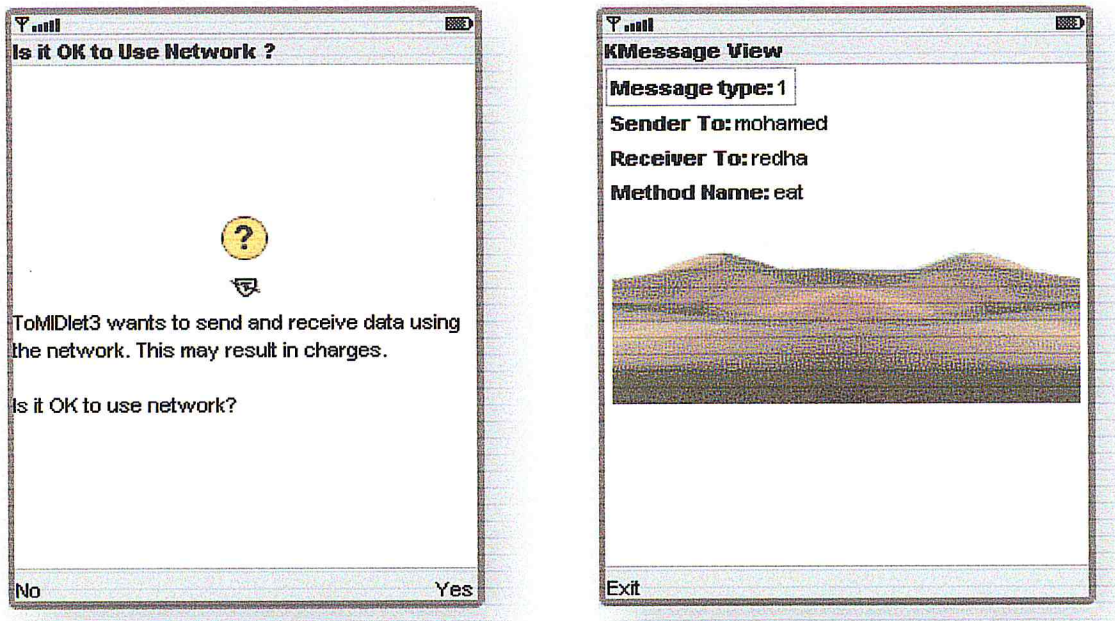


Figure4.8 :la MIDlet affiche le message KMessage.

En sélectionnant la Command *About*, de la fenêtre principale, il y'aura affichage des informations sur

- l'application, et sur
- son développeur.

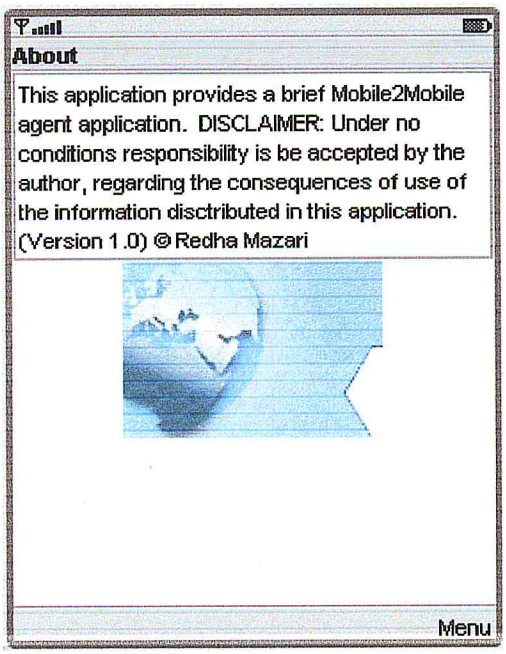


Figure4.9 : la vue About.

3. Réalisation et implémentaion *To* version 5.

Maintenant on va aborder notre réalisation.

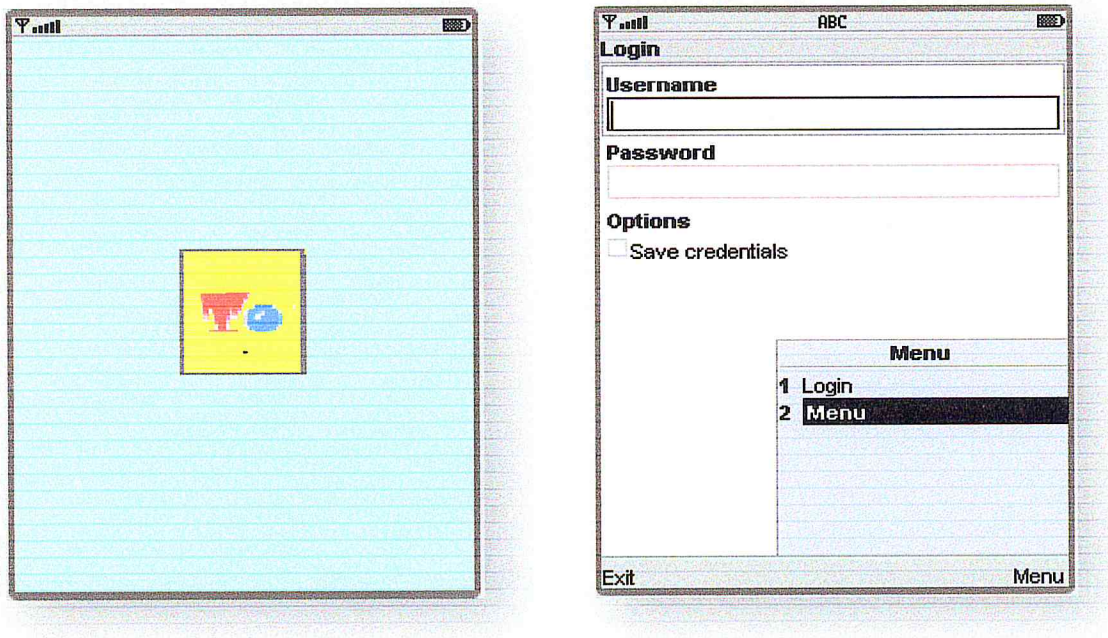


Figure4.10 : le logo de l'application et l'identification de l'application.

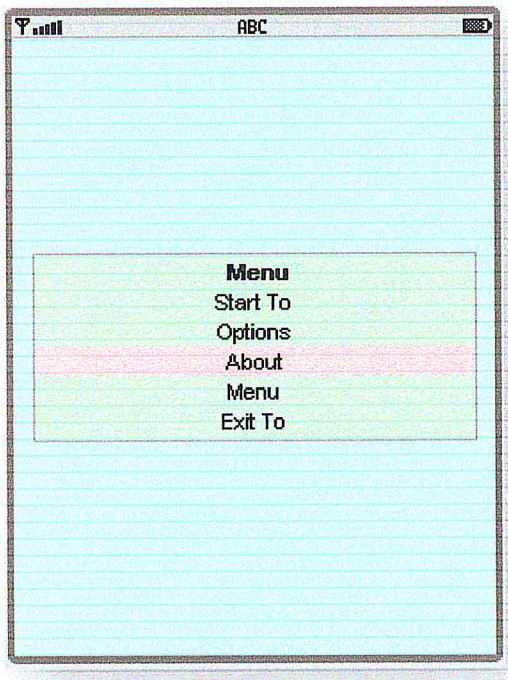


Figure4.11 :l'affichage de Menu.

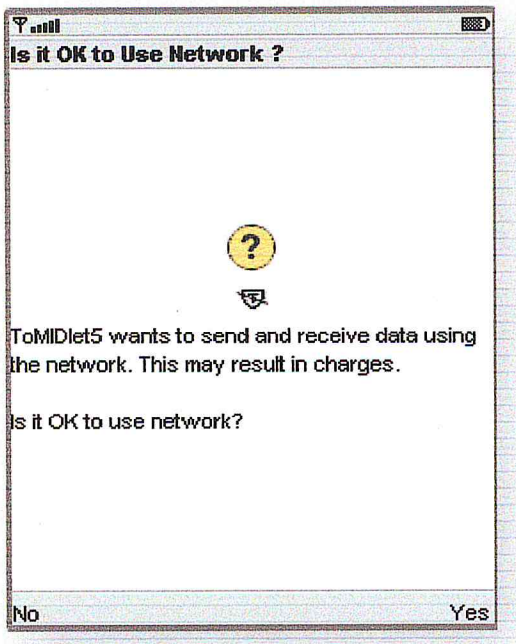


Figure4.12 : la demande de connexion

3.1. La création d'un agent:

Elle reste la même, sauf qu'il y'a ajout d'une vue qui s'affiche sur le *displa*y :

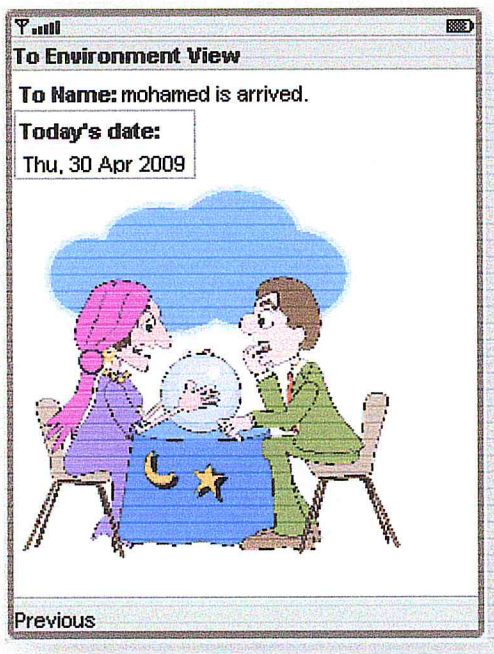


Figure4.12 : la vue de création d'un agent

3.2. Le test de l'exécution d'un DMessage:

Il est à rappeler, que la conception de ce genre de message n'été pas réalisée auparavant dans le To initial.

Dans notre re-conception, on a réussi à réaliser la destruction d'un agent, de l'environnement et du World, avec notre pseudo mobilité.

La **figure 4.13** est une capture d'écran, elle correspond à une vue du *World*, et à côté(figure 4.14) l'Alert correspond à l'état du *World*.

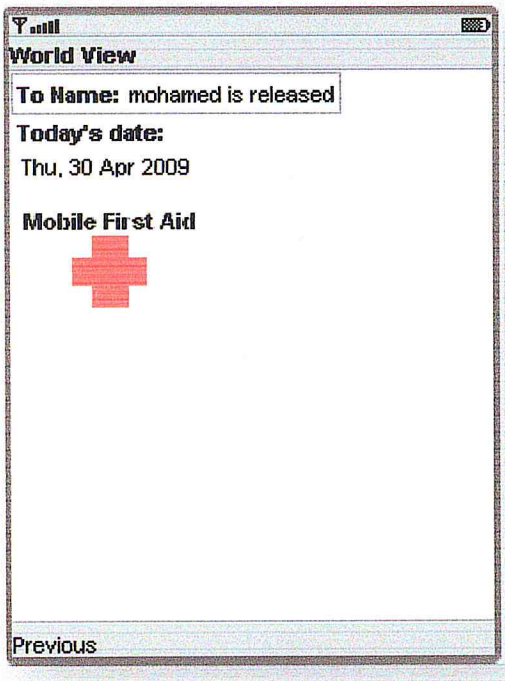


Figure4.13 : la vue de World.

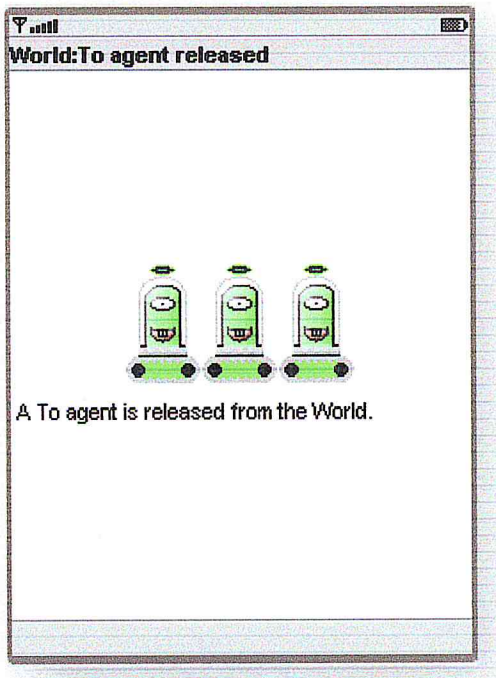


figure4.14 : l'Alert de World.

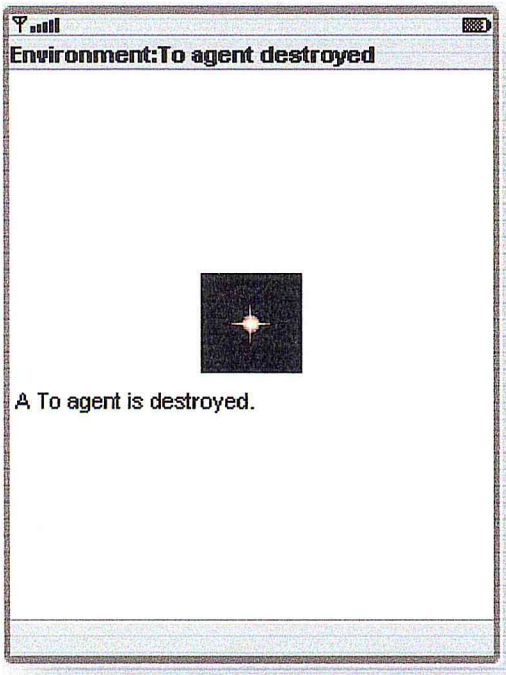


Figure4.15: l'Alert de Environment

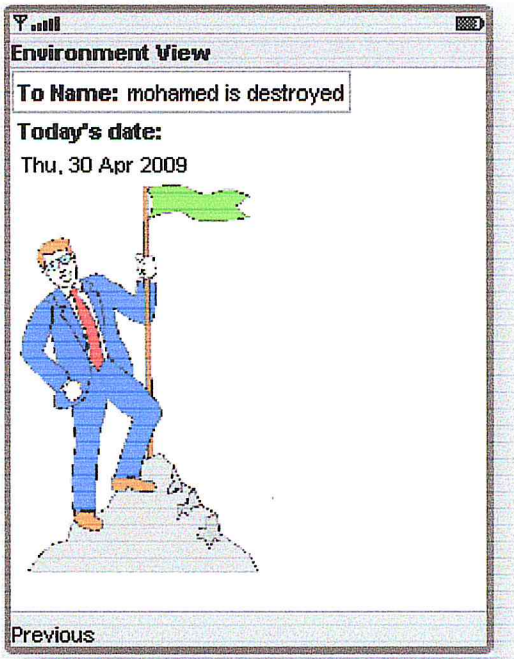


figure4.16: la vue de détruire un agent.

3.3. Le test de l'exécution d'un KMessage:

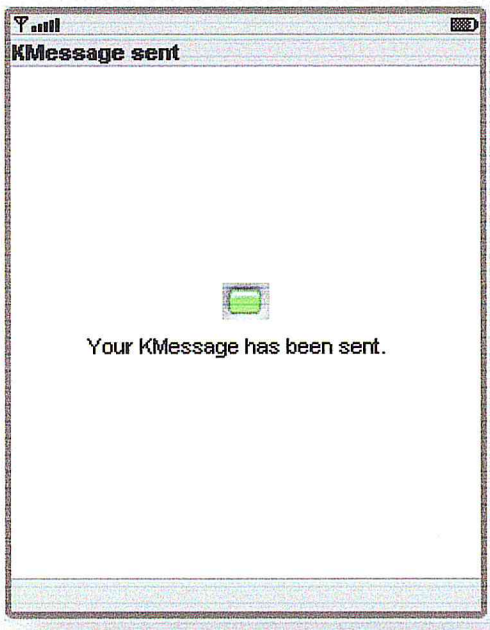


Figure4.17: l'Alert de KMessage.

La réalisation du KMessage était faite dans le To initial.

Mais, dans elle nous a permis de réaliser la pseudo mobilité.

3.4. Le test de l'exécution d'un AMessage:

C'est le cas le plus difficile dans notre développement; la création d'un agent dans un autre environnement, qui permet d'envoyer toutes les caractéristiques d'un agent pour créer un To (un *ToMan ici)*, sur réseau, vers un autre environnement.

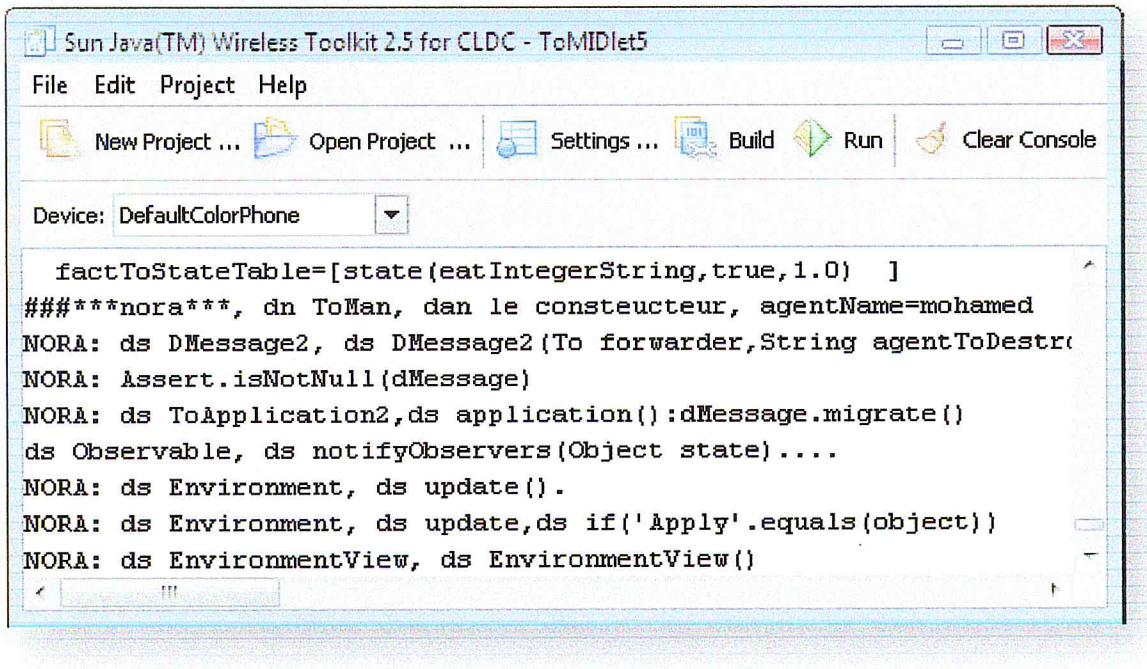


Figure4.18 :le traçage de l'application sur l'émulateur

L'image ci-dessus montre le traçage de l'application sur l'émulateur du WTK.

3.5. La Test de migration de règles sur le serveur http :

La figure 4.19 est une capture d'ecran de l'administrateur PostgreSQL ; qui montre la base de donnée « xmldb » avec les deux tables qu'elle contuent « ruletable » et « facttable » ; pour insérer les règles migrées dans ces derniers.

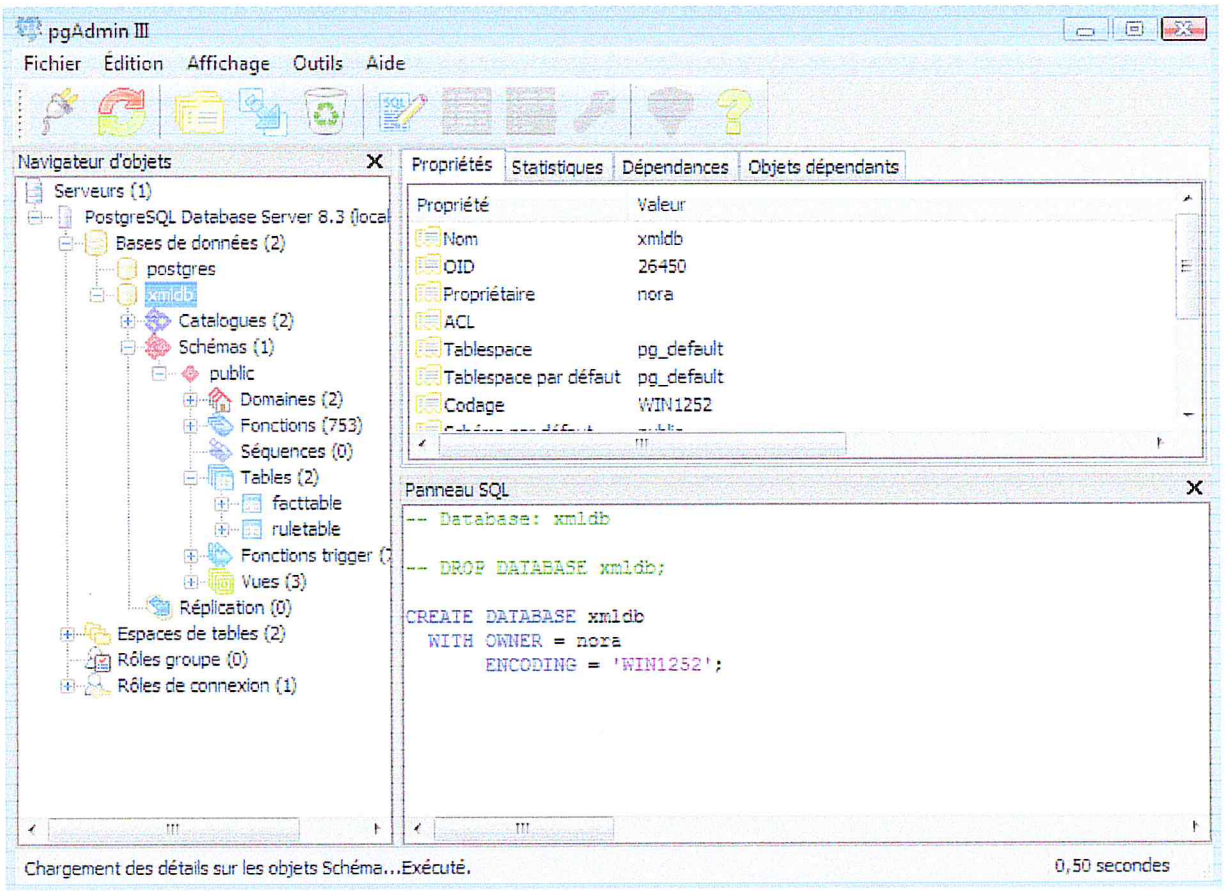


Figure4.19 :pgAdmin

Pour tester cette migration ; on doit se connecter à la base de donnée *xmldb* sur PostGreSQL, comme le montre la figure suivante :

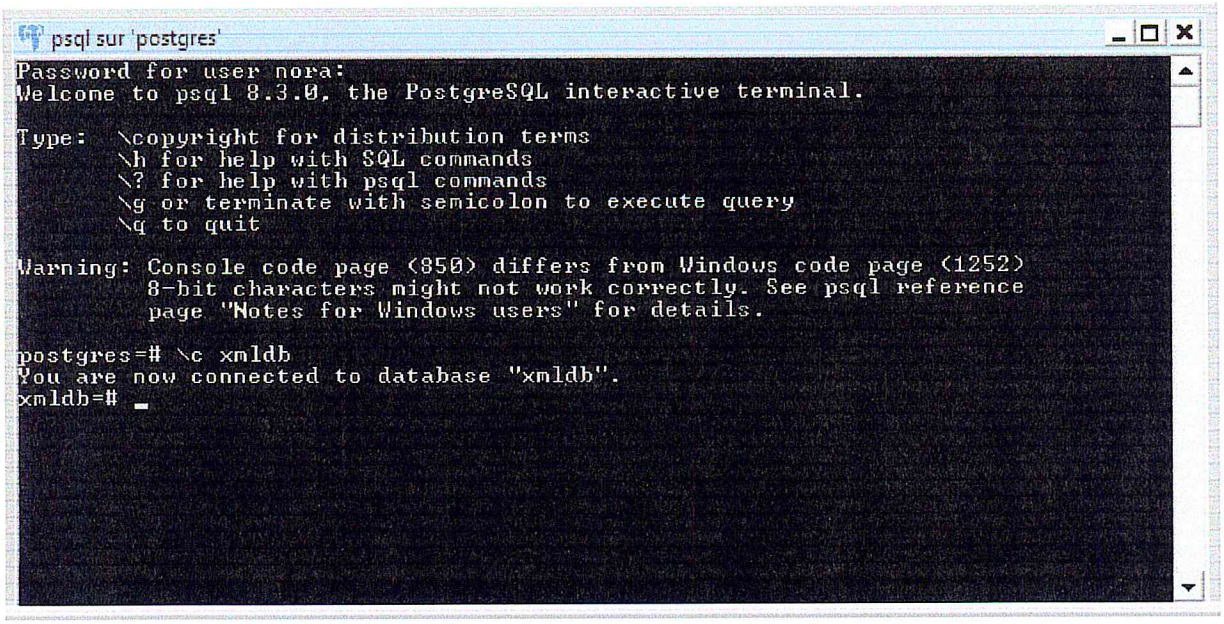


Figure4.20 : la connextion a la base de données "xmldb".

Après la connexion à la base, on lance le serveur *Tomcat*, pour qu'il instancie la servlet *XMLDBServlet*, cette dernière va s'occuper de :

- de parser le *String XMLRule* représentant le schéma XML d'une règle de production en *Rule*, puis
- d'insérer la schéma relationnel de la *Rule*, dans les tables prévues à cet effet, dans la base de données, et
- de faire une requête d'interrogation, dans la base, et ensuite
- d'envoyer le résultat, toujours, sur le même serveur http, vers la *MIDlet* sur téléphone portable.

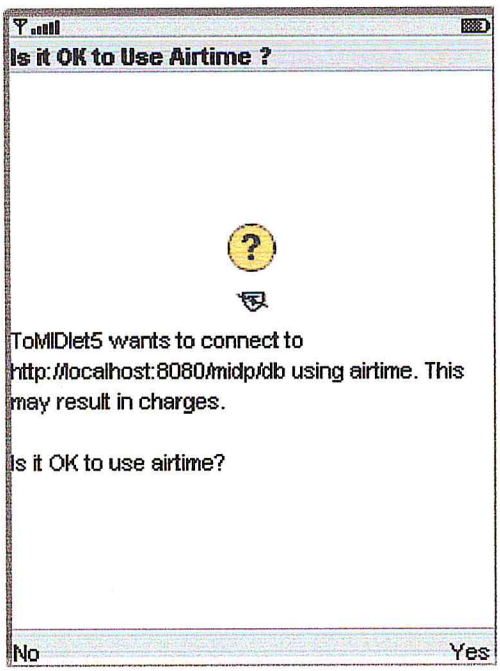


Figure4.21 :la connexion localhost avec le port 8080.

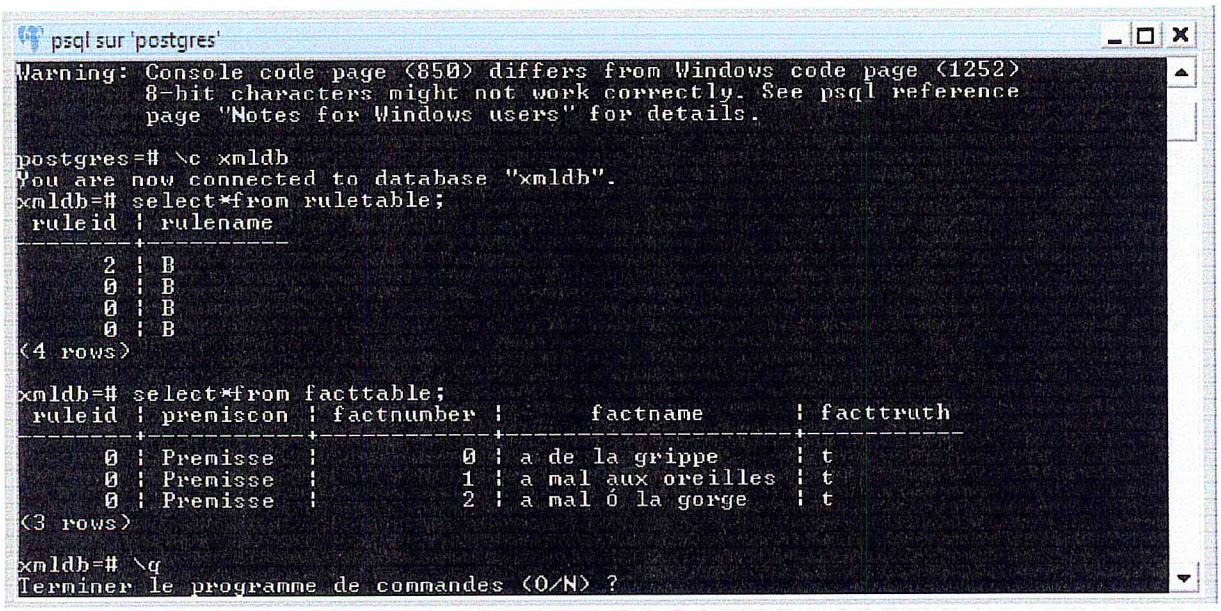


Figure4.22 : premier test d'insertion les règles dans la base.

```
psql sur 'postgres'
xmldb=# select* from facttable;
 ruleid | premiscon | factnumber | factname | facttruth
--------+-----------+------------+----------+-----------
(0 rows)

xmldb=# select* from ruletable;
 ruleid | rulename
--------+----------
      0 | A
      1 | B
      2 | C
      3 | D
      4 | E
      5 | F
(6 rows)

xmldb=# select* from facttable;
 ruleid | premiscon  | factnumber |       factname        | facttruth
--------+------------+------------+-----------------------+-----------
      0 | Premisse   |          5 | a mal au ventre       | t
      0 | Conclusion |          6 | a l'appendicite       | t
      1 | Premisse   |          7 | a de la grippe        | t
      1 | Premisse   |          8 | a mal aux oreilles    | t
      1 | Premisse   |          9 | a mal ó la gorge      | t
      1 | Conclusion |         10 | a les oreillons       | t
      2 | Premisse   |         11 | a la gorge rouge      | t
      2 | Premisse   |         12 | a un rhume            | t
      2 | Conclusion |         13 | a de la grippe        | t
      3 | Premisse   |         14 | a de la grippe        | t
      3 | Premisse   |         15 | a le nez bouchÚ       | t
      3 | Conclusion |         16 | a un rhume            | t
      4 | Premisse   |         17 | a mal au ventre       | t
      4 | Premisse   |         18 | eat too bananas       | t
      4 | Conclusion |         20 | a mal au ventre       | f
      4 | Conclusion |         19 | est guÚri             | t
      5 | Premisse   |         21 | est guÚri             | t
      5 | Conclusion |         22 | eatIntegerBanana      | f
(18 rows)

xmldb=# create table ruletable(ruleid int not null, rulename varchar(32));_
```

Figure4.23 :le dernier test stockage de règles

Les interrogations sont fait par la MIDlet, ce qui donne une réponse depuis la Servlet, comme le montre la figure 4.24.

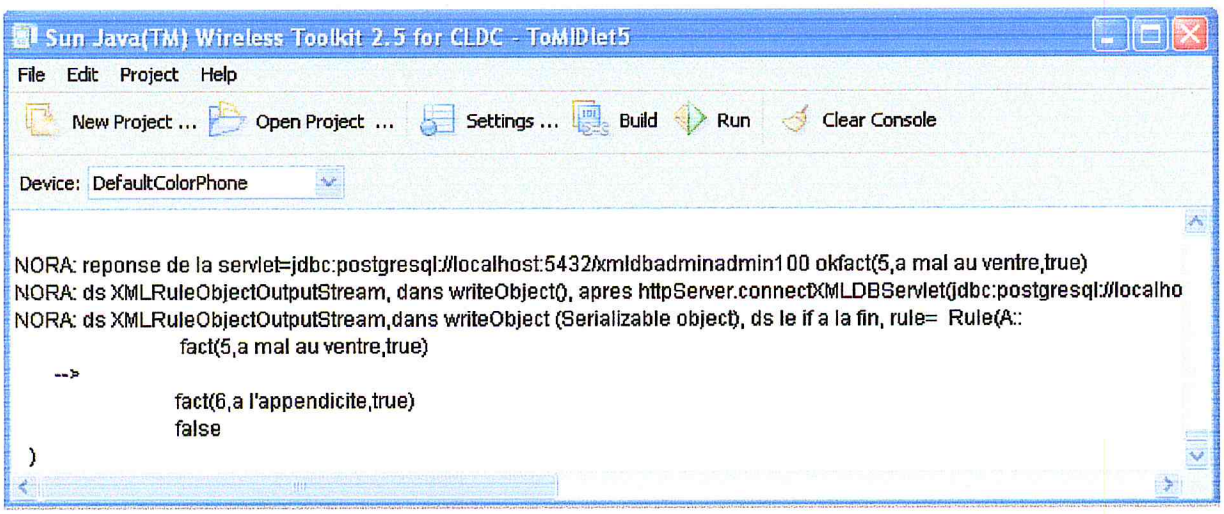
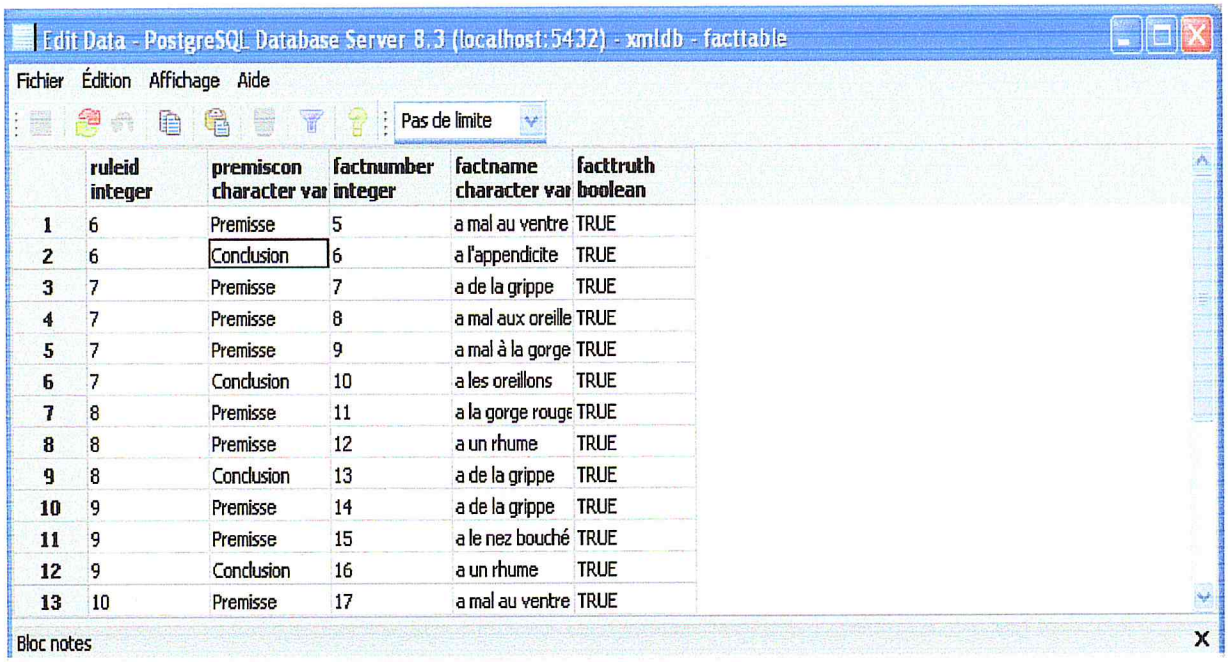


Figure4.24 : la réponse de la Servlet sur l'émulateur.

On voie le résultat sur le Admin de PostgreSQL.



	ruleid integer	premiscon character var	factnumber integer	factname character var	facttruth boolean
1	6	Premisse	5	a mal au ventre	TRUE
2	6	Conclusion	6	a l'appendicite	TRUE
3	7	Premisse	7	a de la grippe	TRUE
4	7	Premisse	8	a mal aux oreille	TRUE
5	7	Premisse	9	a mal à la gorge	TRUE
6	7	Conclusion	10	a les oreillons	TRUE
7	8	Premisse	11	a la gorge rouge	TRUE
8	8	Premisse	12	a un rhume	TRUE
9	8	Conclusion	13	a de la grippe	TRUE
10	9	Premisse	14	a de la grippe	TRUE
11	9	Premisse	15	a le nez bouché	TRUE
12	9	Conclusion	16	a un rhume	TRUE
13	10	Premisse	17	a mal au ventre	TRUE

Figure 4.25 : les tables dans le Admin.

CHAPITRE V

CONCLUSION

En toute chose, c'est la fin qui est essentielle.
ARISTOTE

CONCLUSION

Notre travail se situe dans le contexte de la *maintenance de logiciel*.
Dans notre travail on a utilisé les concepts de l'Orienté Objet et des patterns pour réduire la rigidité, la fragilité, et l'immobilité de notre système.

Les patrons de conception, comme le *Singleton*, *Strategy*, le *Decorator* et surtout l'*Observer* ont joué un grand rôle, que ça soit dans la re-fabrication ou le re-conception du framework *TO*.

L'*Observer* nous a grandement rendu service par les possibilités qu'il offre dans la séparation des composants du framework pour une réutilisation possible du framework.

Contribution

La mobilité est une phase importante dans les systèmes multi agents. Elle permet en effet de déplacer (migrer) un agent, sur une autre machine pour continuer son activité. La non existence de la sérialisation, sous j2me, rend la migration de To sur téléphone portable, presque impossible, si on ne touve pas à la KVM (machine virtuelle).

L'objectif premier de notre travail est de proposer une solution pour adapter la mobilité pour le Framework ***To.***

La première problématique posée dans le cadre de ce mémoire était : comment effectuer une pseudo migration (pseudo mobilité) du *DMessage* et *AMessage*, ou en d'autres termes, comment détruire ou comment créer un To à distance, surtout quand ces dernier sont complexes (ils ont des références sur d'autre objets).

Pour répondre à cette question, nous avons effectué un état de l'art sur les travaux de recherche menés sur l'utilisation des techniques de sérialisation dans ce contexte.

Etant donné, que notre recherche n'était pas fructueuse, nous avons du concevoir notre pseudo sérialisation, et de l'améliorer jusqu'à arriver à un Framework de type *Black-Box*.

Concernant la sécurité, on a aussi conçu et intégré deux approches de framework pour la sécurité de l'envoi d'information sur réseau, dont la deuxième est une véritable solution, basée sur la sur-élevation des flux de j2me, en flux cryptés. Ce sous-framework pourrait directement être incorporé dans J2me, surtout qu'il est basé sur les patrons *Strategy* et le *Decorator*.

La transformation de l'idée de faire migrer un To en la migration des trois types de messages (DMessage, KMessage, et AMessage), est réalisée et réussi.

Il fallait dire que la réussite de l'exécution du *AMessage* n'était pas une mince affaire.

Un des aspects important de notre travail, c'était de faire migrer sur connexion Socket et http, la base de règles.

Nous avons aussi réussi à transformer, un objet Rule, en un schéma XML.

Nous avons ensuite fait migrer le schéma XML de la Rule, à travers un Serveur http, Nous l'avons récupéré ensuite sur un ordinateur, à travers une Servlet sous un Serveur conteneur de *Servlets Tomcat*, puis nous l'avons transformé en un schéma relationnel, qu'on a intégré dans une base de données relationnelle.

Par la suite, nous avons testé l'utilité de la persistance des Rules dans une base de données, en opérant des interrogations SQL de la base, et d'envoi de la réponse à la *MIDlet* sur téléphone portable.

Dans notre travail de re-conception et de re-factoring, nous avons utilisé le développement orienté objet, les patrons et les frameworks dans chaque optimisation,

pour garantir la réduction de la rigidité, de la fragilité, de l'immobilité, en vue d'amélirorer la maintenablité du *Framework To*.

L'Observer, a joué un grand rôle dans notre travail, il nous a permis de faire une conception fortement réutilisable, et de séparer les différents composants du *Framework* de sérialisation, comme toutes les vues, le Receiver, les différents Writer , le Sender, et tous les *Reader* sont tous découplés.

L'*Observer*, nous a aussi beaucoup aidés dans la conception de la migration des arguments de construction de l'objet à la base d'un *To* (en l'occurrence le *Man*, dans notre cas), et sans lui, nous n'aurions pas su comment procéder.

Le patron *Strategy* a permis de généraliser la possibilité d'utiliser tous les algorithmes de cryptage connus.

Le patron Decorator nous a permis de changer les fonctionnalités ou le comportement d'un objet, de manière « douce » et dynamique, comme dans le cas du comportement des flux Socket de J2me.

Dans le processus de re-fabrication, nous avons résolu le problème d'affichage des vues de l'interface graphique, ce qui nous a permis de voir facilement de récupérer, à travers une technique basée sur le stockage dans l'ordre de leur apparition, tous les displays qui existent lancés dans l'application.

Nous n'aurions jamais pu arriver, à concevoir tous ces aspects là et suivre le déroulement de l'exécution, sans l'aide des différents types de tests (unitaire, intégration, régression,..), que nous avons utilisé au moyen de simples classes comme *Assert,* un *mini-Junit,* et *Log* un afficheur générique.

Le travail, que nous avons réalisé était dur et épuisant, mais non moins instructif et fascinant.. Et combien, nous voudrions, dès maintenant commencer, encore un autre cycle de maintenance pour aborder et remédier au tas d'insuffisances et d'omissions. Dont on peut citer :

Travaux futurs

L'objectif principal de notre travail est atteint en un temps record, grâce au dévouement de l'équipe, et surtout au travail précurseur réalisé par Mazari.

Néanmoins, il reste encore beaucoup à faire pour faire mûrir To, dont on pourrait citer, entre autres :

- La refabrication du framework de sérialisation pour résoudre le problème de l'attachement des *Observer*s,
- De faire migrer tout le système expert, de le rapatrier ensuite pour le réutiliser,
- D'utiliser des schémas relationnelles des règles dans le coté apprentissage de To,
- D'introduire un « coté » apprentissage à *To*,
- De généraliser le travail, avec d'autres *To*s,
- D'introduire un autre système de raisonnement,
- De réaliser "a killing application" avec tout ça,
- De tester l'application en "live".

BIBLIOGRAPHIE

Bibligraphie de l'état de l'art :

[Ind02] Eva Indal

"Development of mobile agents in J2ME or similar technologies".

Trondheim,

2002.

[Roa02] San Antonia Road, Palo Alto,

"J2ME Tech Tips".

2002.

[Maz09] Redha Mazari

"Étude et conception d'un Framework pour un système d'agents mobiles et intelligents sur téléphones portables "

Thèse de Magister.

Département d'informatique.

Faculté des sciences.

Université de BLIDA.

[URL] AgentLight – Platform for Lightweight Agents: http://www.agentlight.org

Bibliographie de langage J2ME

[Mah00] Quzay Mahmoud,

"MIDP GUI Proggramming" ,

O'REILLY.

2000.

[Mah01] Mahmoud, Q.H.

"MobiAgent: An Agent-based Approach to Wireless Information Systems".

In Proc. of the 3rd Int.

Bi-Conference Workshop on Agent-Oriented Information Systems,

Montreal,Canada .

2001.

[Kha02] Faisal Khan,

"Java-2 Micro Edition Application Development"

SAMS,

2002.

[Med05] Youssef Medaghri

"Les Sockets - Communication réseau",

Copyright SUPINFO International University,

Château Landon,

11 août 2005.

Bibliographie Framework

[Mat99] Mattsson, M. and Bosch, J.
"Observations on the Evolution of an Industrial OO Framework".
Proceedings of ICSM'99,
International Conference on Software Maintenance,
Oxford, UK, 1999.

[Fay99] Fayad, M. E., Schmidt, D. C., and Johnson, R. E.
"Building Application Frameworks".
Addison-Wesley Pub Co, 1st edition, 1999.

[Chen05] Xin Chen

"Dissection of an Application Frameworks, Part 2".

May 24, 2005.

Bibligraphie de patron de conception :

[Mar99] François Martel,

« *DOCUMENTATION STRUCTURÉE DU DESIGN DES CADRES D'APPLICATION* »,

Mémoire présenté à la Faculté des études supérieures de l'Université de Montréal. Octobre 1999.

[Flo97] G. Florijn, M. Meijers et P. van Winsen,

« *Tool support for object-oriented patterns* »

in Lecture Notes in Computer Science.

(ECOOP97, Finland).

1997.

[JWC98] JAMES W. COOPER

"The Design of Patterns: Java companion"

Addison-Wesley.

1998.

[Gam95] E. Gamma, R. Helm, R. Johnson and J. Vlissides.

"Design Patterns: Elements of Reusable Object-Oriented Software".

Addison-Wesley,

Reading, MA, 1995.

[Eck03] Bruce Eckel. President, MindView, Inc

"Thinking in Patterns: Problem solving Techniques using Java"

Revision 0.9, 5-20-2003

URL: **http://www.Mindview.net**

[Joh97] R.E. Johnson,

"*Frameworks = (components + patterns)*",

Communications of the ACM, vol.40 , No.10, pp. 39-42,

Octobre 1997

[Boi04] Olivier BOISSIER ,

"Analyse, Conception Objet (Design Pattern Introduction)",

SMA/G2I/ENS Mines Saint-Etienne,

Avril 2004

[Lon03] Jacques Lonchamp ,

"Génie Logiciel Quatrième partie:les techniques de conception",

CNAM - CRA Nancy,

2003

[Kho04] Khashayar Khosravi,

"Design Pattern-enabled object-oriented metrics",

Séminaire GÉLO (Laboratoire de génie logiciel, Université de Montréal)

LATECE Technical Report 2004.

18 février 2004.

[SR02] SDSU & Roger Whitney,

"Observer ",

San Diego State University, CS 635 Advanced Object-Oriented Design & Programming,

Spring Semester,

2002.

Bibiliographie : SMA+agent mobile

[Bou04] S. Bouchenak, D. Hagimont, S. Krakowiak, N. De Palma, and F. Boyer,
"Experiences implementing efficient Java thread serialization, mobility and persistence".
Software- Practice and Experience.
April 2004.

[Fer95] Jacques Ferber.
"Les Systèmes Multi-Agents. Informatique Intelligence Artificielle".
InterEdition,
1995.

[Fug98] Alfonso Fuggetta, Gian Pietro Picco, and Giovanni Vigna.
"Understanding Code Mobility".

IEEE Transactions on Software Engineering,
May 1998.

[Sid06] M. R. Sidoumou,

"Etude et réalisation d'un Framework pour agents mobiles intelligents".

Mémoire d'ingénieur.

Département d'informatique.

Faculté des sciences.

Université de BLIDA.

2006.

[Sun98] Sundsted, T.,

"Agents Talking to Agents, "

Javaworld,

September 1998.

Bibliographie de sérialisation

[Höf05] Tom Höfte

"Vector and Object Serialization pour J2ME MIDP".

2005.

[Wek05] Wekmar Sobel

"Serialisation Objects".

O'REILLY.

2005.

Bibliographie de test

[Ale02] Alexander, R. T. and J. Offutt .
"Criteria for Testing Polymorphic Relationships".
ISSRE'00 (Int. Symposium on Software Reliability Engineering), San Jose, CA, USA.
2002

[Ale02b] Alexander, R. T., A. J. Offutt, et al.
"Fault Detection Capabilities of Coupling-Based OO Testing".
ISSRE'02 (Int. Symposium on Software Reliability Engineering), Annapolis, MD, USA.
2002b.

[Kun96] Kung, D. C., J. Gao, et al.
"On Regression Testing of Object-Oriented Programs."
The Journal of Systems and Software
(1996).

[Lab00] Labiche, Y.
"Contribution au test des logiciels orientés-objet :
Ordre de test, modèle et critères associés,
LAAS-CNRS.
2000.

[McG01] McGregor, J. D.
"A Practical Guide To Testing Object Oriented Testing"
Addison Wesley.
2001.

[Bri03] Briand, L. C., Y. Labiche, et al.
"An Investigation of Graph-Symposium on Software Testing and Analysis",
Roma, Italy.

2003.

Bibliographie de maintenance de logiciels

[Abr07] A. Abran, R. Al Qutaish, J. Desharnais, and N. Habra,

"ISO-based Model to Measure Software Product Quality".

Institute of Chartered Financial Analysts of India (ICFAI),

ICFAI Books,

2007.

[Ant07] P. Antonellis, D. Antoniou, Y. Kanellopoulos, C. Makris, E. Theodoridis,

C. Tjortjis, and N.Tsirakis,

"A data mining methodology for evaluating maintainability according to ISO/IEC-9126 software engineering –product quality standard".

In Special Session on System Quality and Maintainability

SQM2007, 2007.

[Bro07] M. Broy, F. Deissenboeck, and M. Pizka,

"Demystifying maintainability,"

In Fourth International Workshop on Software Quality Assurance

(SOQUA 2007).

ACM, 2007.

[Bos00] J. Bosch,

"Design & Use of Software Architectures".

Wesley, 2000.

[Pet00] J. F. Peters and W. Pedrycz,

"Software Engineering, an Engineering Approach"

Wiley,

2000.

[Fir04] D. Firesmith,

"OPEN Process Framework (OPF),"

2004.

http://www.donald-firesmith.com

[Agg02] Aggarwal K K, et.al.,

"An Integrated Measure of Software Maintainability",

Annual Reliability and Maintainability Symposium,

2002.

Bibliographie : re-enginireeng :

[Chi04] Elliot J. Chikofsky and James H.

Cross II, "*Reverse Engineering and Design Recovery: A Taxonomy,"*

IEEE Software,

2004.

[Pre01] R. S. Pressman,

"Software Engineering, A Practitioner's Approach", 5th edition,

McGraw-Hill,

2001.

[Ros05] Linda H. Rosenberg

"Software Re-engineering",

Software Assurance Technology Center

Unisys Federal Systems

2005.

Bibliographie de refactoring

[Fow00] Martin Fowler

"Refactoring. Improving the Design of Existing Code"

Addison-Wesley,

2000,

[Opd82] WILLIAM F. OPDYKE

"Refactoring Object-Oriented Frameworks"

University of Wisconsin - Madison,

1982.

Bibliographie de langage XML:

[Boi00] Bob Boiko

"Understanding XML,"

http://www.metatorial.com/articles/xml.asp,

2000.

[Bon00] Angela Bonifati and Stefano Ceri

"Comparative Analysis of Five XML Query Languages,"

ACM SIGMOD Record 29, No. 1

March 2000.

[Bou02] Ron Bourret

"XML and Databases,"

http://rpbourret.com/xml/XMLAndDatabases.htm

2002.

[Kay03] Michael Kay

"XML Databases,Software AG".

www.softwareag.com

White Paper

March 2003.

[Bun01] Peter Buneman, Wenfei Fan, Jérôme Siméon, and Scott Weinstein

"Constraints for Semistructured Data and XML,"

ACM SIGMOD Record 30, No. 1

(March 2001).