

Projet Multi-Modules - Getting Things Done

Conception d'un serveur de données GTD

S. Begaudeau
B. Gosset
A. Lagarde
C. Renaudineau

Janvier 2010

Table des matières

1 Livrable 1 : Analyse	3
2 Livrable 2 : Spécification d'exigences logicielles	29
2.1 Introduction	29
2.1.1 Objectif	29
2.1.2 Conventions	29
2.1.3 Audience	29
2.1.4 Portée du document	29
2.1.5 Définitions, acronymes et abréviations	30
2.1.6 Références	30
2.1.7 Organisation du chapitre	30
2.2 Description générale	30
2.2.1 Perspectives du produit	30
2.2.2 Fonctions du produit	31
2.2.3 Caractéristiques et classes d'utilisateurs	31
2.2.4 Environnement opérationnel	31
2.2.5 Contraintes de conception et d'implémentation	32
2.2.6 Documentation utilisateur	32
2.2.7 Hypothèses et dépendances	32
2.2.8 Exigences reportées	32
2.3 Fonctionnalités du logiciel	32
2.3.1 Traitement des requêtes	33
2.3.2 Les requêtes de lecture	35
2.3.3 Les requêtes de création, de mise à jour et de suppression	38
2.3.4 Renommer le pseudo de l'utilisateur	41
2.3.5 Créer un compte	44
2.3.6 Supprimer un compte	46
2.3.7 Changer le mot de passe d'un utilisateur	49
2.4 Autres exigences non-fonctionnelles	51
2.4.1 Utilisabilité	51
2.4.2 Fiabilité de l'application	52
2.4.3 Exigences de performance	53
2.4.4 Rapidité du serveur	53
2.4.5 Efficacité du serveur	54
2.4.6 Maintenabilité	54
2.4.7 Exigences de sûreté	54
2.5 Exigences de sécurité	54
2.6 Classification des exigences fonctionnelles	55
3 Livrable 3 : Spécification de l'interface du Serveur GTD	56
3.1 Définition des données GTD partagées par le serveur et les applications clientes	56
3.2 Fonctionnement général de la communication avec le serveur GTD	57
3.2.1 Les étapes fondamentales de la communication avec le serveur GTD	57
3.3 Détails de fonctionnement	58

3.3.1	Le mécanisme de CallBack	58
3.3.2	Le mécanisme d'identification	60
3.3.3	Mode de mise à jour	60
3.4	Signature des méthodes de l'interface Serveur	61
3.4.1	Creation, mise à jour et suppression	61
3.4.2	Créer une Tache	65
3.4.3	Mettre à jour une Tâche	65
3.4.4	Supprimer Tache	65
3.4.5	Créer un projet	66
3.4.6	Mettre à jour un projet sur le serveur	66
3.4.7	Supprimer un projet	66
3.4.8	Récupération des données	66
3.4.9	Gestion des comptes	68
3.4.10	Fonctionnalités proposées à l'Administrateur	69
3.5	Spécifications des composants du serveur GTD	69
3.5.1	Les composants CorbaServer et RMIServer	70
3.5.2	Le composant Reactor	71
3.5.3	Le composant CommandeFactory	71
3.5.4	Le composant MessageQueueing	71
3.5.5	Le composant CommandeManager	72
3.5.6	Le composant DataManager	72
3.5.7	Le composant IHMAdministration	72
3.5.8	Le composant RMIAdministration	72
3.6	Interaction inter-composants	73
3.6.1	Diagramme d'activité du Serveur GTD	73
3.6.2	Diagrammes de séquences du Serveur GTD	74
4	Livrable 4 : Architecture	76
4.1	Architecture Physique du serveur GTD	76
4.2	Schéma de la base de données GTD	77
4.3	Règles de traduction de UML en code source	78
5	Livrable 5 : Conception détaillée	79
5.1	Principes suivis pour la conception/le développement	79
5.1.1	Conception EJB/JEE : un modèle en couche structurant les applications	79
5.1.2	Bonnes pratiques liées à l'implémentation : conventions, style et tests	79
5.2	Données partagées entre client et serveur : logique suivie	80
5.3	Description détaillée de chaque composant	81
5.3.1	Composant DataManager : les entités du système et la logique métier	81
5.3.2	Reactor	82
5.3.3	Le composant CommandeFactory	84
5.3.4	La CommandeQueue : implémentation du composant MessageQueueing	85
5.4	Informations techniques utiles	86
5.5	Conclusion	87
5.5.1	Analyse et conception	87
5.5.2	Organisation interne et externe	87
5.5.3	Implémentation du projet	88
6	Annexes	89

PROJET MULTI-MODULES :

GTD

LIVRABLE 1 : ANALYSE

Réalisé par :

Stephane Begaudeau
Benjamin Gosset
Alex Lagarde
Christophe Renaudineau

SOMMAIRE

<i>Introduction</i> -----	<i>page 3</i>
<i>I. Dictionnaire des données -----</i>	<i>pages 3 à 5</i>
<i>II. Diagramme de classes-----</i>	<i>pages 6 à 8</i>
<i>A. Analyse-----</i>	<i>pages 6-7</i>
<i>B. Contraintes OCL liées au diagramme de classes-----</i>	<i>pages 7-8</i>
<i>III. Cas d'utilisation-----</i>	<i>pages 9 à 26</i>
<i>A. Collect -----</i>	<i>pages 9 à 12</i>
<i>B. Process -----</i>	<i>pages 12 à 16</i>
<i>C. Organize -----</i>	<i>pages 16 à 21</i>
<i>D. Review -----</i>	<i>pages 21 à 22</i>
<i>E. Do -----</i>	<i>pages 22 à 26</i>
<i>Conclusion -----</i>	<i>page 26</i>

Introduction

Le projet GTD s'inscrit dans le cadre de notre parcours visant à assimiler les différentes méthodes et techniques nécessaires à la création d'architectures logicielles. En effet, le projet devant être implémenté par l'ensemble des étudiants du Master, l'architecture de notre application devra être particulièrement bien pensée, et basée sur de solides spécifications.

Dans ce premier livrable, nous étudierons la méthodologie GTD, et produirons une analyse du problème posé, en tâchant de respecter au maximum le cahier des charges proposé dans le sujet.

Nous définirons dans une première partie l'ensemble des termes et notions relatifs à la méthodologie GTD au moyen d'un dictionnaire de données, dictionnaire sur lequel nous nous appuierons par la suite.

Nous présenterons ensuite le diagramme de classe (niveau Analyse) modélisant le problème posé, ainsi que les contraintes OCL que nous avons jugé comme apportant du sens à l'analyse.

Nous déterminerons ensuite l'ensemble des cas d'utilisation correspondant à la méthodologie GTD (présentés au moyen du canevas de Cockburn), en présentant pour chacun de ces cas un diagramme d'activité, des instantanés illustrant ces cas et éventuellement des diagrammes de séquences.

En ce qui concerne les contraintes fonctionnelles que nous avons établies, nous avons préféré les identifier au fur et à mesure de notre analyse et les présenter dans les parties où elles sont les plus pertinentes.

I. Dictionnaire des données

Dans cette partie, nous définirons l'ensemble des termes relatifs à la méthodologie GTD. Une fois définis, nous pourrons nous appuyer sur une terminologie claire lors de notre analyse.

GTD (Getting Things Done) :

La méthode GTD est une démarche d'organisation personnelle applicable à l'ensemble des activités d'un utilisateur. La GTD permet une conception de gestion des priorités associées à des tâches. Le but de la GTD est d'optimiser l'attention de l'utilisateur en se focalisant uniquement sur les actions à effectuer dans un contexte donné, avec des priorités choisies et en tenant compte du temps nécessaire et de l'effort à fournir pour réaliser cette tâche. Dans le cadre de ce livrable, on restera à un niveau très élevé, en considérant GTD comme une méthodologie pouvant s'appliquer sans support informatique (simple feuille de papier).

Idée :

Une idée représente un souhait de l'utilisateur, au sens le plus vague du terme. Cette idée, recensée lors de l'opération de collecte, pourra éventuellement être transformée en tâches ou projets.

Tâche :

Une tâche est une activité pouvant s'inscrire dans un projet et qui doit être réalisée dans un contexte précis. Outre les différents attributs inhérents à une tâche (son nom, sa date de début et fin, ...), chaque tâche possédera un état spécifiant l'avancement de cette dernière.

Avancement (d'une tâche) :

Donne des informations sur le statut de la tâche (à faire, déléguée, en attente, terminée...).

Projet :

Un projet est un regroupement de sous projets et de tâches liés sémantiquement.

Contexte:

Un contexte est défini à partir de différentes contraintes telles que l'endroit où l'utilisateur se trouve, les outils disponibles, les limites et possibilités de l'environnement. Chaque tâche doit être attachée à un et un seul contexte.

Taux d'effort demandé:

Ce paramètre doit être pris en compte lors du calcul de la prochaine tâche à réaliser. Ce taux d'effort peut prendre en compte des paramètres physiques ou psychologiques. Il est compris entre 0 et 99.

Priorité :

La priorité est une valeur associée à chaque tâche, correspondant à l'importance que lui accorde l'utilisateur. Elle permet d'ordonner différentes tâches en concurrence, et reste soumise initialement au jugement de l'utilisateur.

Échéance :

Date de fin associée à une tâche.

Date de création :

Date à laquelle une tâche a été créée.

Fréquence:

On peut associer une fréquence à une tâche, correspondant à sa répétition dans le temps. Par exemple, une tâche peut être unique, quotidienne, mensuelle...

Participants:

Dans le cadre d'une tâche, un participant est la personne chargée de mener à bien cette tâche. On considérera qu'il existe un et un seul participant associé à chaque tâche.

Dans le cadre d'un projet, les participants sont l'ensemble des personnes qui peuvent consulter ce projet.

Tag :

Mot-clé associé à une tâche ou un projet. Une tâche et un projet peuvent disposer de plusieurs tags. Comme nous le verrons par la suite, ces tags permettront d'identifier et de trier les différentes actions à réaliser.

InBox :

Correspond à une "boîte à idée", dans laquelle on place toutes les idées recensées lors de l'opération de collection (voir cas d'utilisations). Ces idées seront ignorées, mises de côté pour être traitées plus tard ou bien transformées en action lors de l'opération "process", entraînant leur suppression de l'InBox.

Liste des projets :

Correspond à la liste des projets et des tâches que contient un projet. Les projets sont triés par ordre alphabétique, et les tâches qu'ils contiennent par taux d'effort, date puis priorité.

Plus tard :

Correspond à la liste des idées à réaliser plus tard.

Archive :

Une tâche ou un projet peuvent être placés dans les archives, lorsque cette tâche ou l'ensemble des tâches qui composent ce projet sont terminées.

Prochaines tâches à réaliser :

Correspond à une liste de tâches triées selon le taux d'effort demandé, la date de fin et la priorité. Elle met en évidence les tâches que l'utilisateur doit exécuter en priorité.

Prochaines tâches à réaliser par contexte :

Il s'agit de la liste des tâches à réaliser mais restreintes à un contexte donné.

Tâches déléguées :

Il s'agit de la liste des tâches déléguées à d'autres participants, afin de suivre l'avancement du travail de ces participants.

Liste des tâches correspond à un tag :

Il s'agit de la liste obtenue lorsque l'on choisit un tag particulier et que l'on désire voir toutes les tâches associées à ce mot-clé.

Calendrier :

Il sert à garder la trace des tâches à faire qui ont une échéance et qu'il ne faut donc pas manquer. Concrètement, correspond à une liste de tâches triées par date d'échéance.

Poubelle :

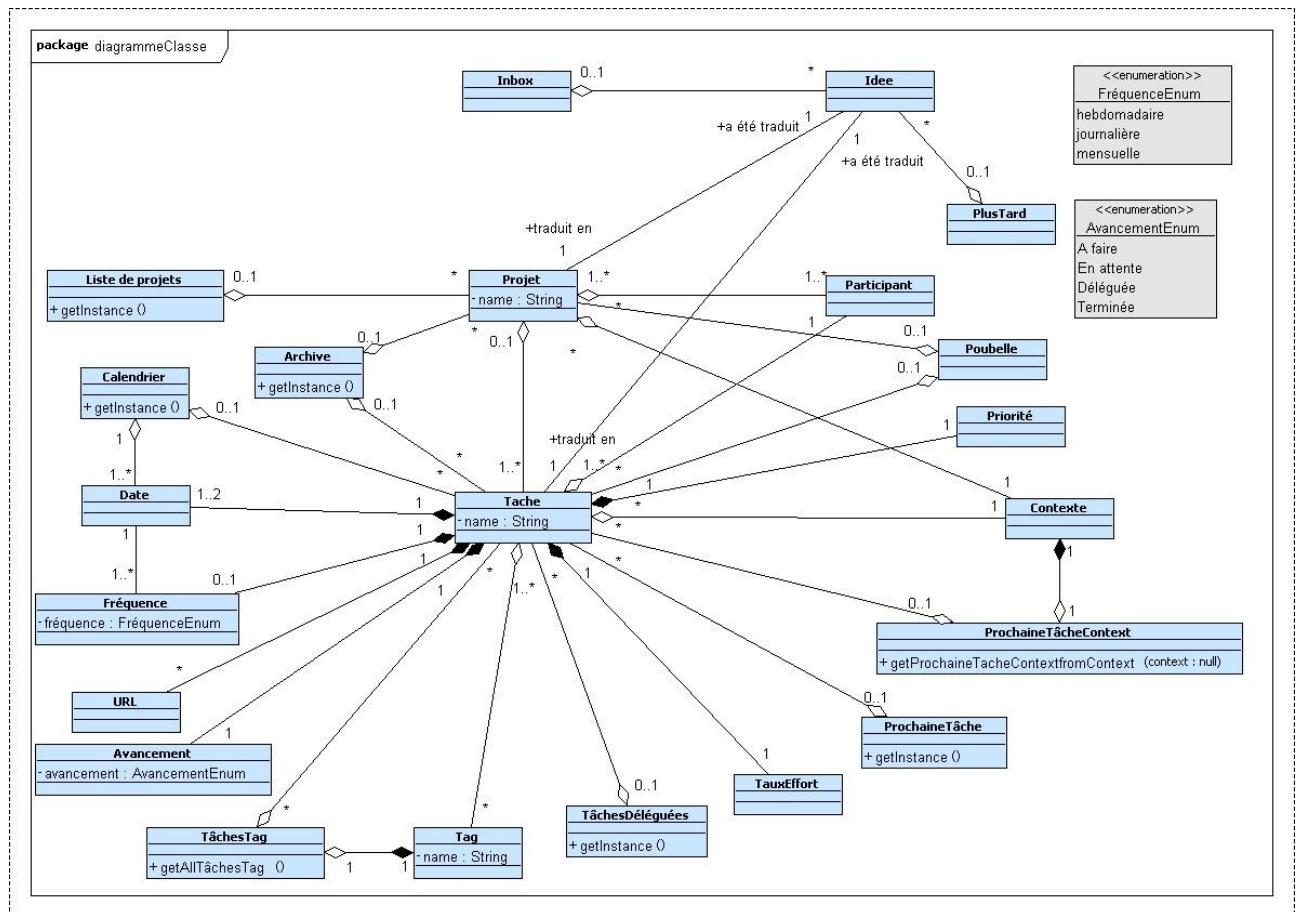
Liste des tâches et des projets supprimés des archives et près à être éliminés définitivement.

II. Diagramme de classes

A. Diagramme de Classe de la méthodologie GTD (niveau Analyse)

Pour réaliser le diagramme de classe (niveau Analyse) représentant la méthodologie GTD, nous nous sommes fortement appuyés sur les différents termes définis dans le dictionnaire de données, en essayant d'identifier les relations entre chaque entité/concept défini.

Nous avons obtenu le diagramme de classe suivant :



Sans revenir sur chacune des entités définies ici, nous reviendront sur quelques points essentiels. Premièrement, intéressons-nous à l'entité Tâche.

Une tâche est définie à partir de deux dates (une date de création et une échéance), une fréquence (énumération possédant une date d'arrêt), une liste d'Urls associés, un avancement représentant l'état de la tâche (finie, déléguée...), une liste de tags (chaque tâche possède au moins un tag), un taux d'effort représentant l'effort nécessaire à la réalisation de cette tâche, un contexte, une priorité, un unique participant, une idée lui correspondant (i.e l'idée qui a entraîné la création de cette tâche) et éventuellement un projet associé.

Il nous a semblé intéressant de représenter un lien entre une idée et les tâches ou projets qu'elle a engendrée. Ce lien est uniquement symbolique, représentant le fait qu'une tâche ou un projet est toujours construit à partir d'une idée. Il ne pourra pas être représenté physiquement, car une idée est supprimée dès qu'elle a engendré les tâches et projets lui correspondant.

On définit différentes listes contenant des tâches, chaque liste fournissant une fonctionnalité de la méthodologie GTD : calendrier trie les tâches par date d'échéance, ProchaineTâcheContext contient les tâches ayant un certain contexte, triées par taux d'effort demandé, date d'échéance et priorité, ainsi qu'une liste de projets (tâches triées par projet), TâchesTags (tâches triées par tags)...

A noter que l'on définira également une Poubelle contenant toutes les tâches supprimées, une Archive contenant toutes les tâches terminées ainsi que l'entité PlusTard, contenant toutes les idées qui ont été écartées pour le moment.

Afin de faciliter l'écriture des contraintes OCL sur ce diagramme de classe, nous avons déterminé que plusieurs entités (notamment Archive, Poubelle, ProchaineTâche...) implémentaient le design patern Singleton, et par conséquent ajoutés plusieurs méthodes à ces classes (en particulier getInstance() permettant de récupérer l'unique instance de la classe).

B. Contraintes OCL liées au diagramme de classes

Nous avons défini plusieurs contraintes OCL sur les différentes entités de ce diagramme de classe. Là encore, le but n'est pas d'être exhaustif mais d'apporter des précisions à notre analyse.

context Tache

La date de début doit être antérieure à la date de fin

inv : self.dateFin==null or self.dateDeb.antérieur(self.dateFin);

Le participant associé à une tâche doit également être associé au projet contenant cette tâche
and (self.projetAssocié == null or self.projetAssocié.participants->includes(self.participant))

priorité comprise entre 1 et 5

context Priorité

inv : self.value>=1 and self.value<=5

Taux d'effort compris entre 0 et 99

context TauxEffort

inv : self.value>=0 and self.value<=99

Toutes les tâches et projets contenues dans Archive doivent être terminées

context Archive

inv : self.taches->forAll(t : Tache | t.Avancement.getValue()=='Terminée')

and self.projets->forAll(p : Projet | p.taches->forAll(t : Tache |

t.Avancement.getValue()=='Terminée')

Toutes les tâches contenues dans la liste des prochaines tâches à réaliser doivent avoir le statut 'A faire'

context ProchaineTâche :

inv : self.taches->forAll(t : Tache | t.Avancement.getValue()=='A faire'

De plus, toutes les tâches contenues dans ProchaineTâche doivent être contenues dans la liste ProchaineTâcheContext associée au contexte de cette tâche

and ProchainetâcheContext.getProchaineTacheContextfromContext(t.getContexte()).taches->includes(t)
)

Toutes les tâches contenues dans la liste des prochaines tâches à réaliser par contexte doivent avoir le statut 'A faire' et être associées au contexte de la liste des prochaines tâches à réaliser par contexte considéré

context ProchaineTâcheContext :

inv : self.taches->forAll(t : Tache | t.Avancement.getValue()=='A faire' and t.getContexte() == self.contexte

De plus, toutes les tâches contenues dans cette liste doivent être également contenues dans la liste ProchaineTâches

and ProchaineTâche.getInstance().taches->includes(t)
)

Toutes les tâches contenues dans la liste des tâches déléguées doivent avoir le statut 'Délégué'

context TâchesDélégues :

inv : self.taches->forAll(t : Tache | t.Avancement.getValue()=='Déléguée')

Tous les projets et tâches contenus dans la Poubelle ne doivent pas se trouver également à un autre endroit du système

context Poubelle :

inv : self.taches->forAll(t: Tache |
 Prochainetâche.getInstance().taches->excludes(t)
 and Archive.getInstance().taches->excludes(t)
 and TâchesDélégues.getInstance().taches->excludes(t)
 and TâchesTag.getAllTâchesTag()->forAll(tt : Tâchetag | tt.taches->forAll(td :
 TâchesDélégues | td->excludes(t))
 and Calendrier.getInstance().taches->excludes(t)
 and ListedeProjets.getInstance().projets->forAll(p : Projet | p.taches->excludes(t))
)
and self.projets->forAll(p : Projet |
 Archive.getInstance().projets->excludes(p)
 and ListedeProjets.getInstance().projets->excludes(p)
)

III. Cas d'utilisation

A. Collect

1 – Canevas de Cockburn

INFORMATIONS CARACTÉRISTIQUES

But dans ce contexte :

Collecter toutes les idées que l'utilisateur souhaite réaliser.

Portée :

L'inbox.

Niveau :

Top

Pré-conditions :

aucune

Condition de succès :

L'inbox contient les idées que l'utilisateur souhaitait ajouter et sauvegarder.

En définissant CollectionIdees comme étant une collection contenant les idées que l'utilisateur souhaitait ajouter :

CollectionIdees->forAll(i : Idee | Inbox.getInstance().idees->includes(i))

Condition d'échec :

L'inbox ne contient pas ces nouvelles idées.

En définissant CollectionIdees comme étant une collection contenant les idées que l'utilisateur souhaitait ajouter :

CollectionIdees->exists(i : Idee | Inbox.getInstance().idees->excludes(i))

Acteur primaire :

L'utilisateur.

Déclencheur :

L'utilisateur lance la création d'une nouvelle idée.

SCÉNARIO DE SUCCÈS PRINCIPAL

1- l'utilisateur déclenche la création d'une idée.

2- une idée vide est créée.

3- l'utilisateur nomme cette idée.

4- l'utilisateur sauvegarde son idée.

EXTENSIONS

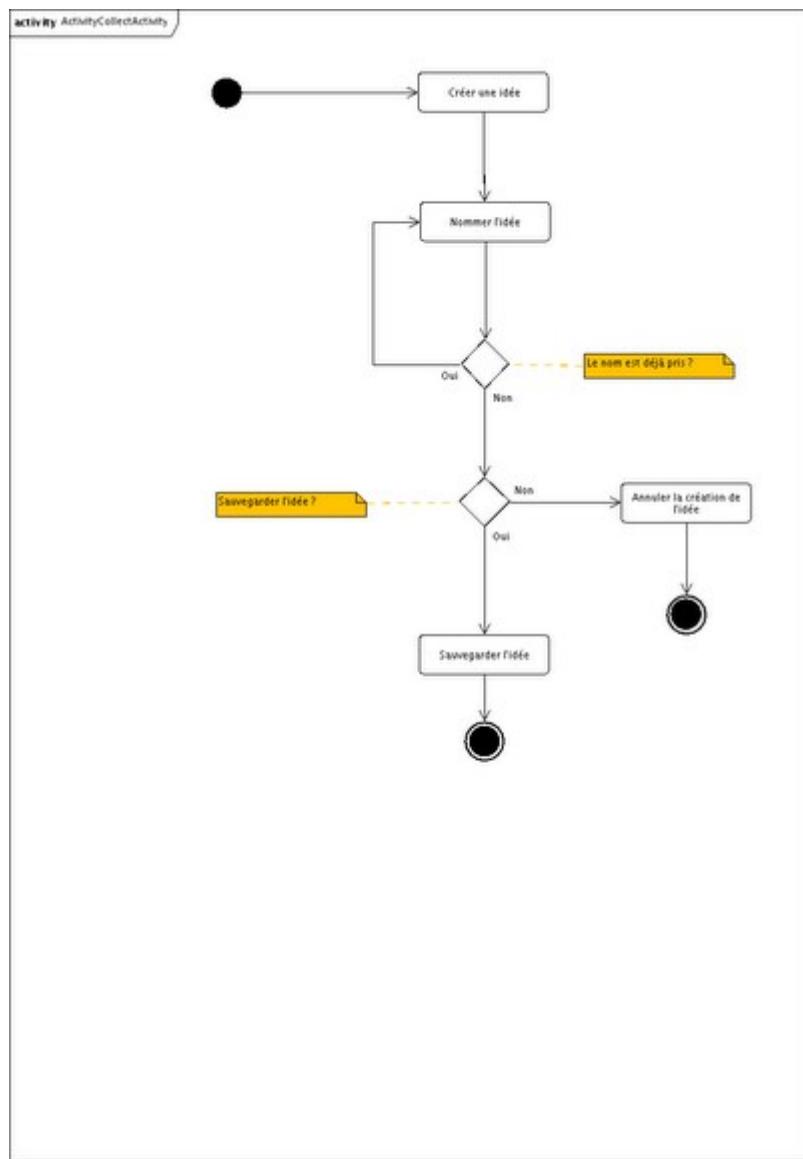
1- si l'utilisateur annule la création de l'idée, l'idée n'est pas créée et le système reste dans son état d'origine.

2- si l'idée à créer possède le nom d'une idée existante alors elle n'est pas créée.

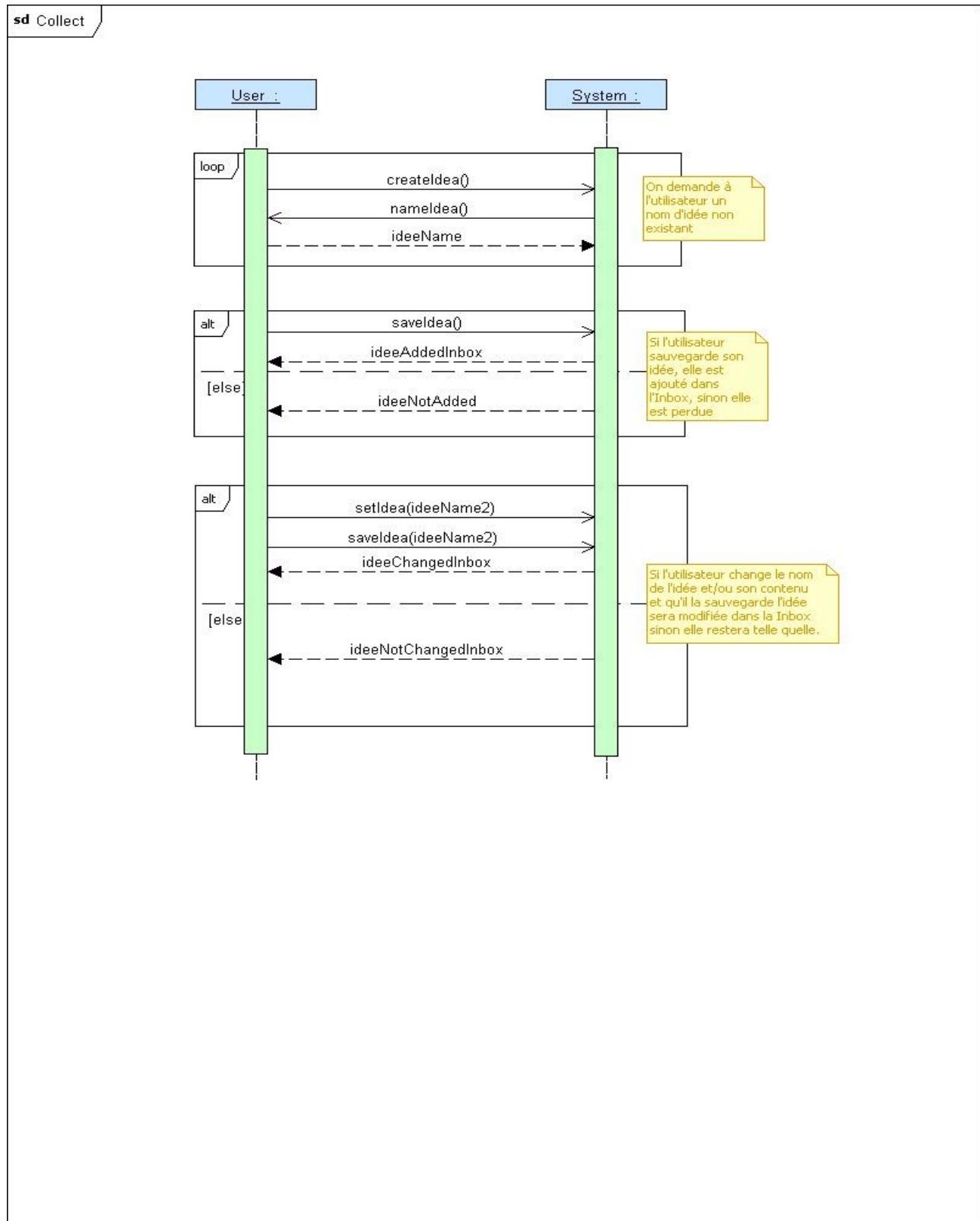
SOUS-VARIATIONS

1- l'utilisateur prend une idée de l'inbox, la modifie en la renommant et la sauvegarde.

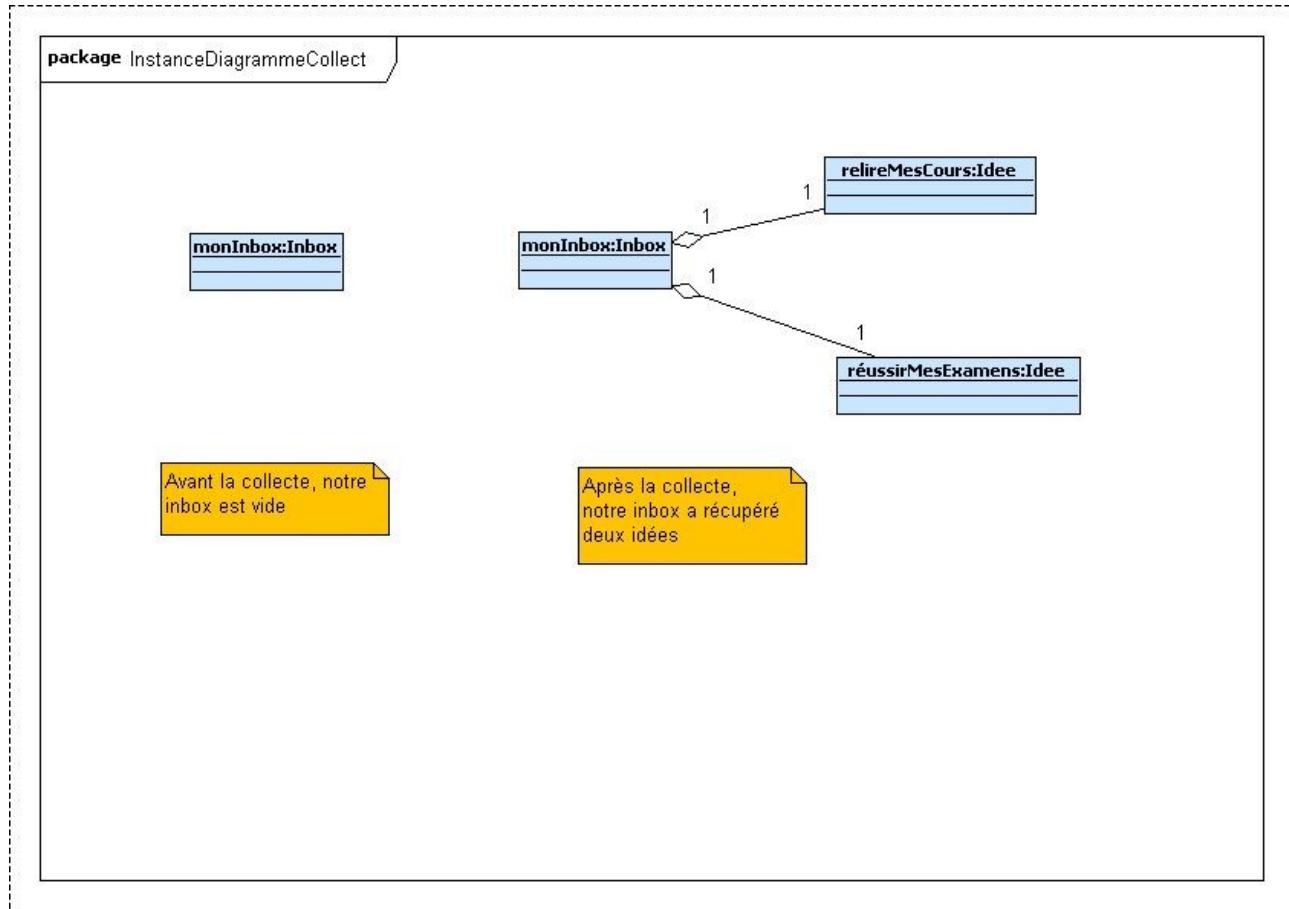
2 – Diagramme d'activité :



3 - Diagramme de séquence



4 - Instantané correspondant



B. Process

1 – Canevas de Cockburn

INFORMATIONS CARACTÉRISTIQUES

But dans ce contexte :

Traiter les idées brutes définies lors de la phase *collect*.

Portée :

L'inbox.

Niveau :

Top

Pré-conditions :

Des idées doivent avoir été définies lors de la phase *collect*.

`Inbox.getInstance().idees->notEmpty()`

Condition de succès :

Toutes les idées ont été traitées, c'est-à-dire transformées en tâches ou projets ou bien mises dans la liste "plus tard" ou dans la poubelle.

En définissant CollectionTP étant une collection contenant les tâches et projets qui ont été définies dans cette phase process :

```
Inbox.getInstance().idees@pre->forAll(i : Idee | PlusTard.getInstance().idees->includes(i) ||  
CollectionTP->exists(t : Tache | t.idee = i) || CollectionTP->exists(p : Projet | p.idee = i))  
and Inbox.getInstance().idees->isEmpty()
```

Condition d'échec :

Il reste des idées non traitées.

En définissant CollectionTP étant une collection contenant les tâches et projets qui ont été définies dans cette phase process :

```
Inbox.getInstance().idees@pre->exists(i : Idee | PlusTard.getInstance().idees->excludes(i) and  
CollectionTP->forAll(t : Tache | t.idee <> i)  
and CollectionTP->forAll(p : Projet | p.idee <> i) or Inbox.getInstance().notEmpty()
```

Acteur primaire :

L'utilisateur

Déclencheur :

L'utilisateur désire traiter les idées qu'il a définies lors de la phase *collect*.

SCÉNARIO DE SUCCÈS PRINCIPAL

- 1- l'utilisateur parcourt les idées définies lors de la phase *collect*.
- 2- l'utilisateur sélectionne une idée qui demande plus de deux minutes pour être réalisée.
- 3- l'utilisateur détermine que cette idée est élémentaire et qu'elle doit donc être gérée par une tâche.
- 4- l'utilisateur crée une tâche ayant pour base l'idée sélectionnée.

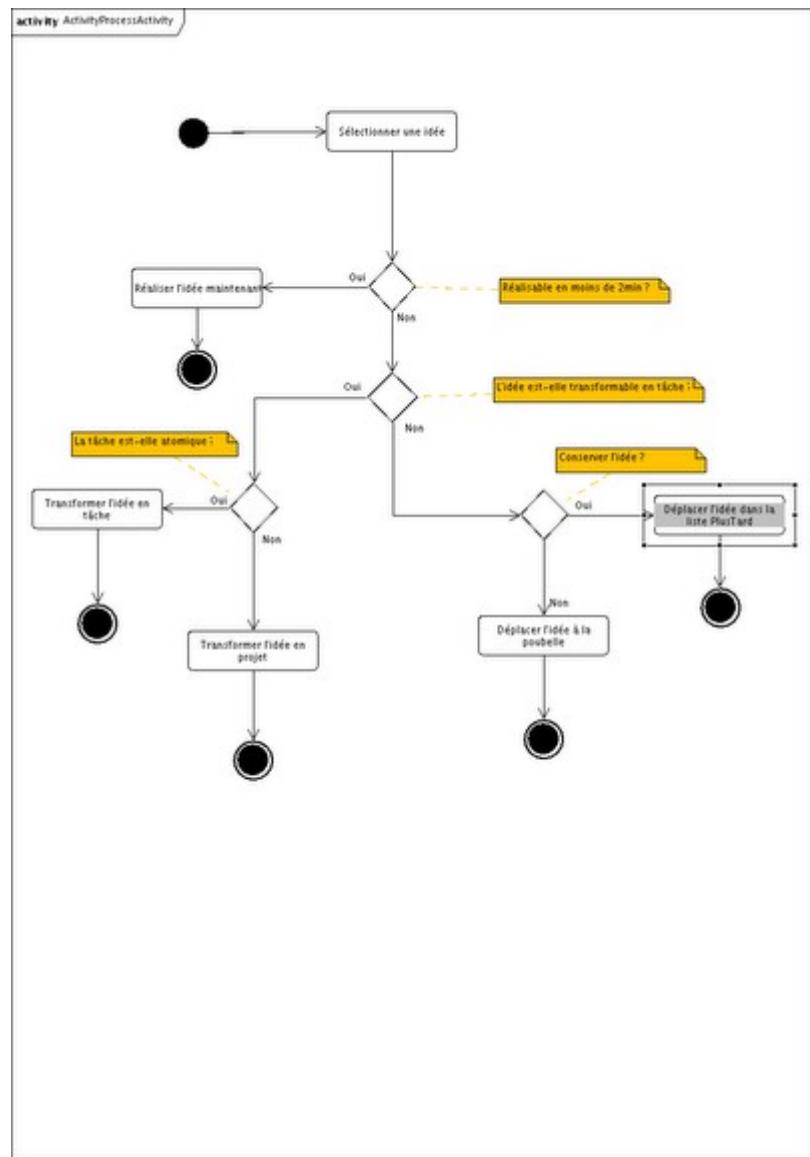
EXTENSIONS

- 1- si une idée requiert moins de deux minutes, elle doit être traitée dès maintenant.
- 2- si une idée ne peut être transformée en tâche ou en projet alors elle est supprimée ou conservée pour plus tard, selon la politique choisie par l'utilisateur.

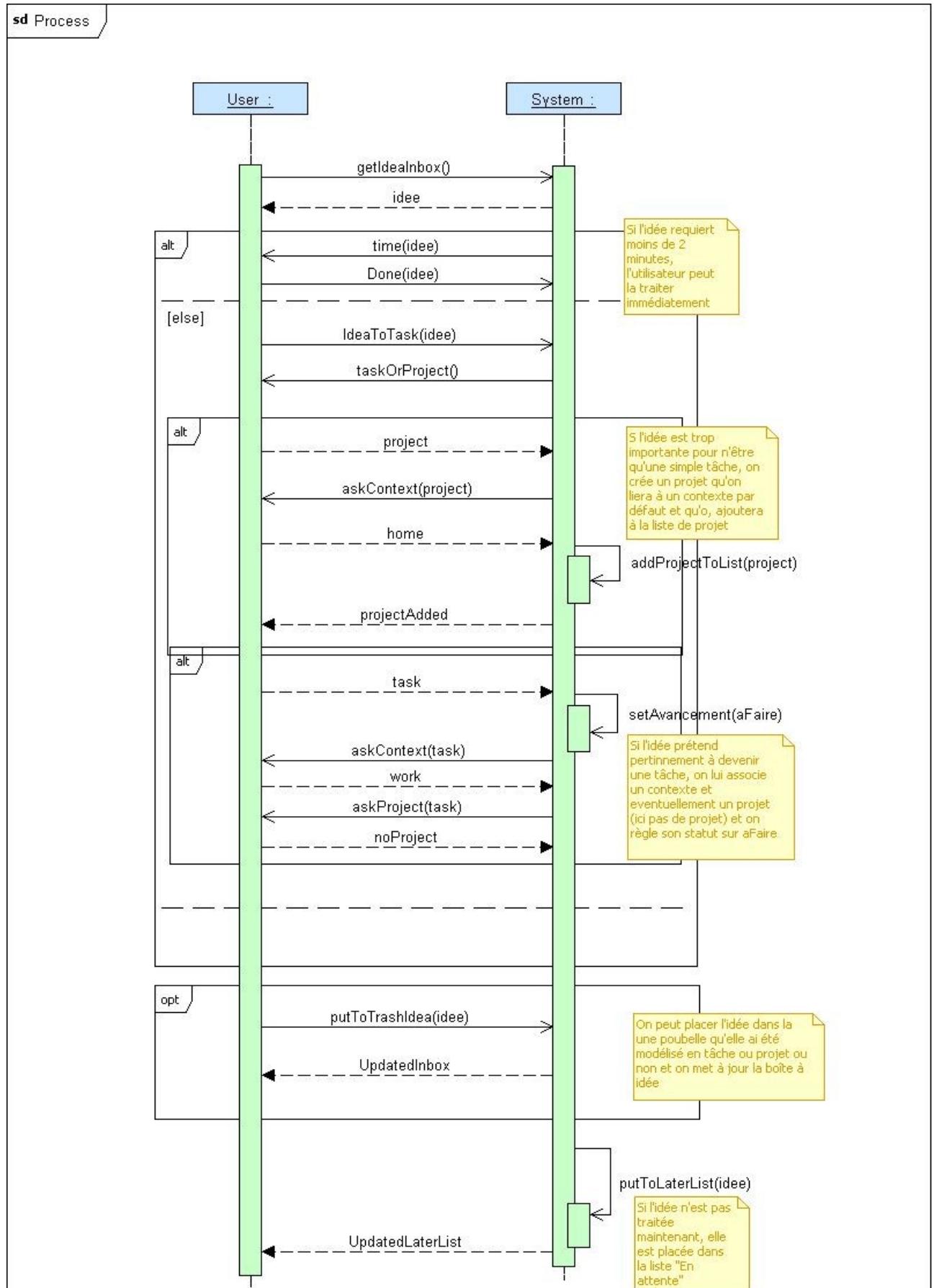
SOUS-VARIATIONS

- 1- si une idée requiert plus d'une tâche pour être réalisée alors elle est transformée en projet. On détermine alors le contexte par défaut des tâches qui seront ajoutées à ce projet (si elles n'ont pas de contexte).

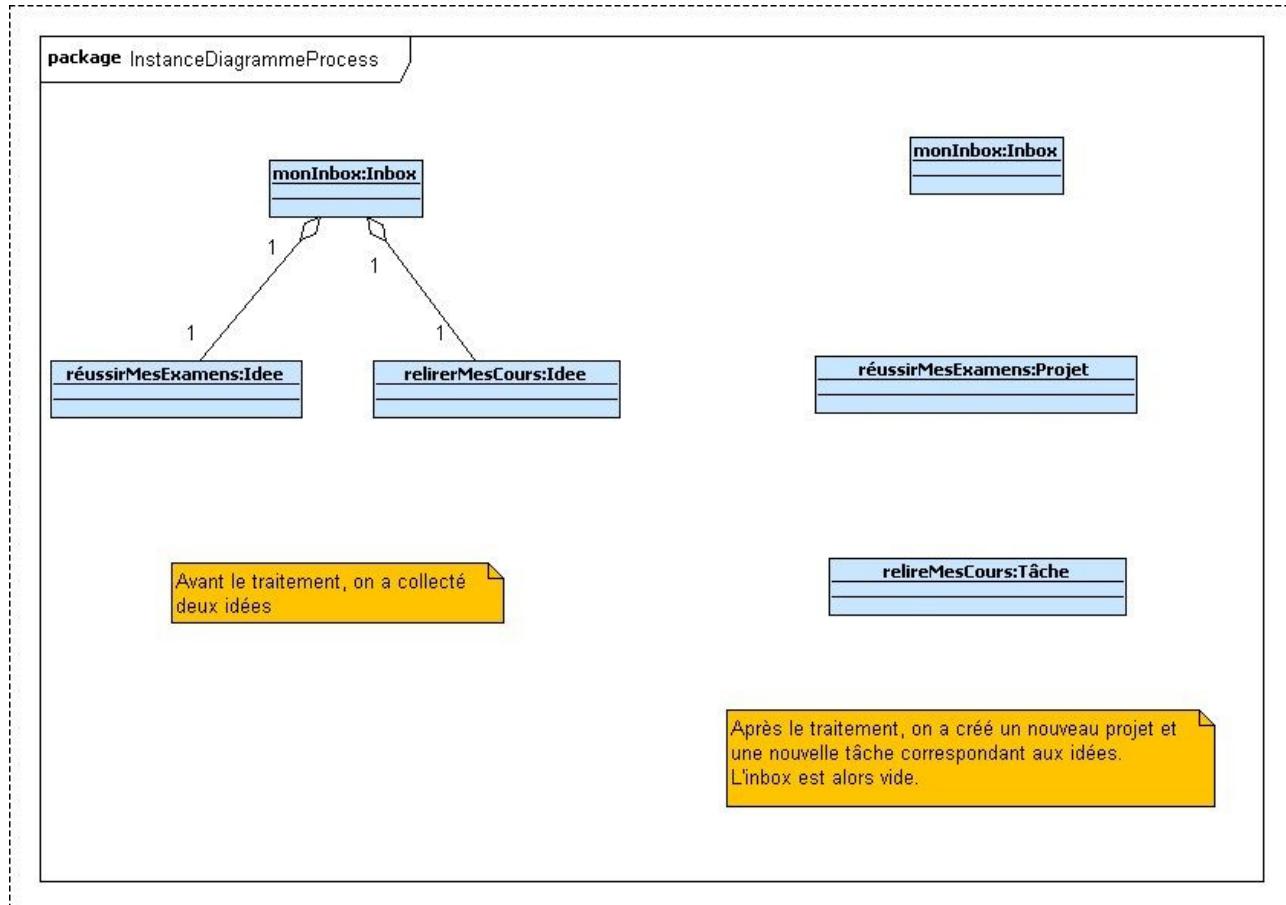
2 - Diagramme d'activité



3 - Diagramme de séquence



4 - Instantané correspondant



C.Organize

1 – Canevas de Cockburn

INFORMATIONS CARACTÉRISTIQUES

But dans ce contexte :

Organiser les tâches et les projets.

Portée :

La liste des projets, la liste des prochaines tâches à réaliser, la liste des prochaines tâches à réaliser par contexte, la liste des tâches déléguées, la liste plus tard, les archives, la poubelle.

Niveau :

Top

Pré-conditions :

Des tâches et des projets ont été créés après la phase de tri des idées *process*.

ListeProjet.getInstance().projets->notEmpty() or Tache.getAllTaches()->notEmpty()

Condition de succès :

Les tâches et les projets ont été organisés.

Condition d'échec :

Les tâches et les projets sont toujours non organisés.

Acteur primaire :

L'utilisateur.

Déclencheur :

L'utilisateur souhaite organiser ses actions.

SCÉNARIO DE SUCCÈS PRINCIPAL

- 1- l'utilisateur analyse la première tâche à traiter.
- 2- l'utilisateur détermine qu'elle doit être déléguée à une autre personne.
- 3- l'utilisateur ajoute un participant au projet.
- 4- l'utilisateur délègue la tâche à ce participant.
- 5- la tâche est ajouté à la liste "*déléguée*".

EXTENSIONS

SOUS-VARIATIONS

1- l'utilisateur sélectionne une tâche, détermine qu'elle doit être déléguée, il choisit un des participant du projet actuel puis il lui délègue la tâche.

2- l'utilisateur sélectionne une tâche, détermine qu'elle fait partie d'un projet, et l'ajoute à ce projet. Le projet contient la tâche ajoutée à la fin.

3- l'utilisateur sélectionne une tâche, détermine qu'elle fait partie d'un projet, et l'ajoute à ce projet. Si la tâche ne contient pas de contexte, elle prend le contexte par défaut défini par ce projet. Le projet contient la tâche ajoutée à la fin.

4- l'utilisateur considère une tâche, et essaye de déterminer l'importance de cette tâche par rapport aux tâches de la liste des prochaines tâches. Il fait alors varier le taux d'effort et la priorité de cette tâche pour la placer à l'endroit voulu dans la liste des prochaines tâches.

5- l'utilisateur sélectionne un projet, détermine au sein de ce projet l'ordre dans lequel organiser les tâches, puis les trie (en modifiant la propriété et le taux d'effort) conformément à l'ordre choisi.

6- l'utilisateur passe en revue les idées de la liste "*plus tard*", trouve une idée qui peut être réalisée maintenant, puis la transforme alors en tâche ou projet (*process*).

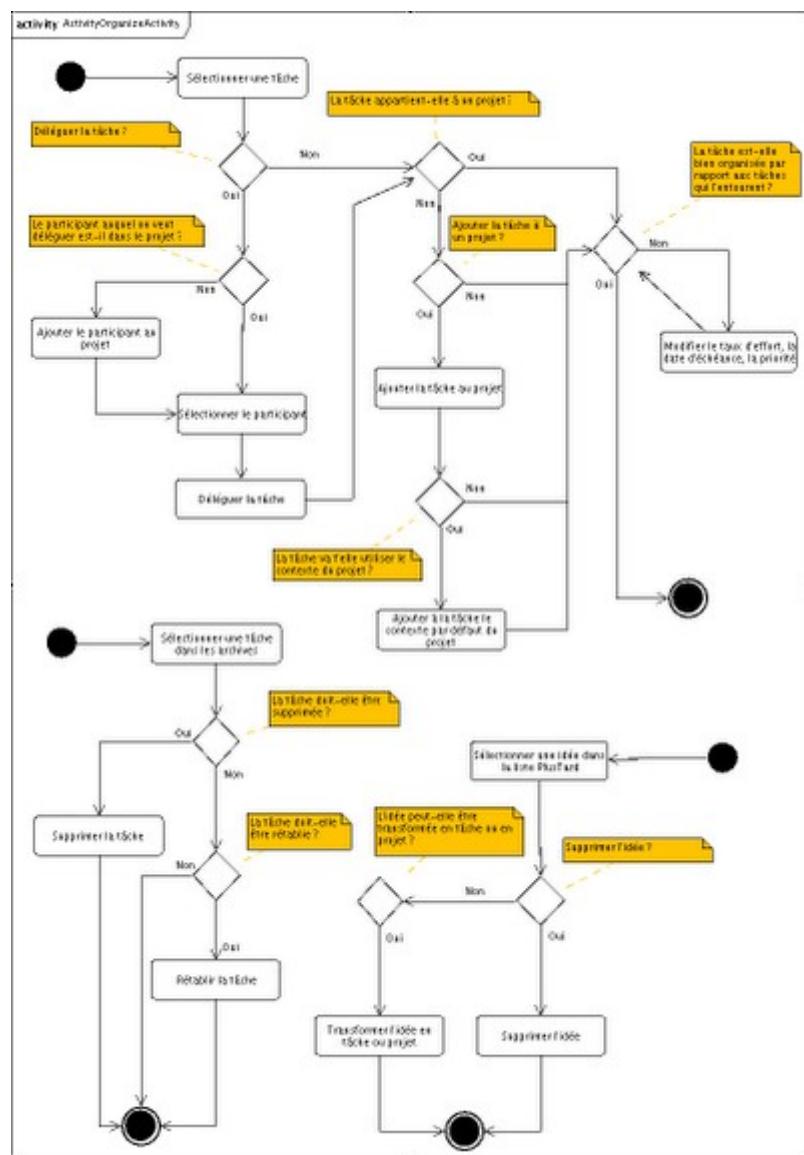
7- l'utilisateur passe en revue les idées de la liste "*plus tard*", trouve une idée qui peut être supprimée, puis l'efface. L'idée est déplacée dans la poubelle.

8- l'utilisateur passe en revue les tâches contenues dans les archives, détermine que certaines tâches et projets terminés peuvent être éliminés, puis les supprime. Ces derniers sont déplacés dans la poubelle.

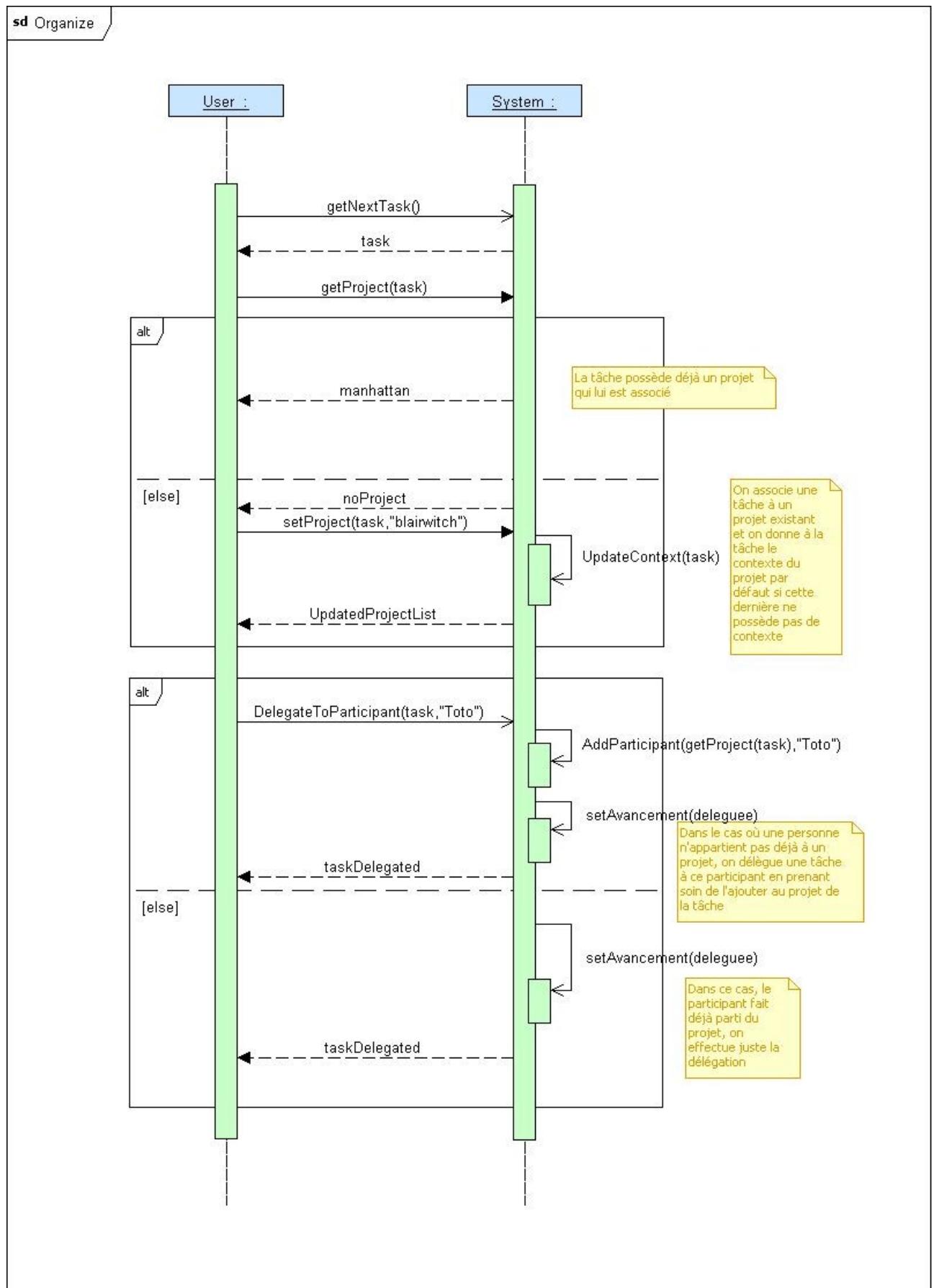
9- l'utilisateur veut faire le ménage, il décide de vider la poubelle.

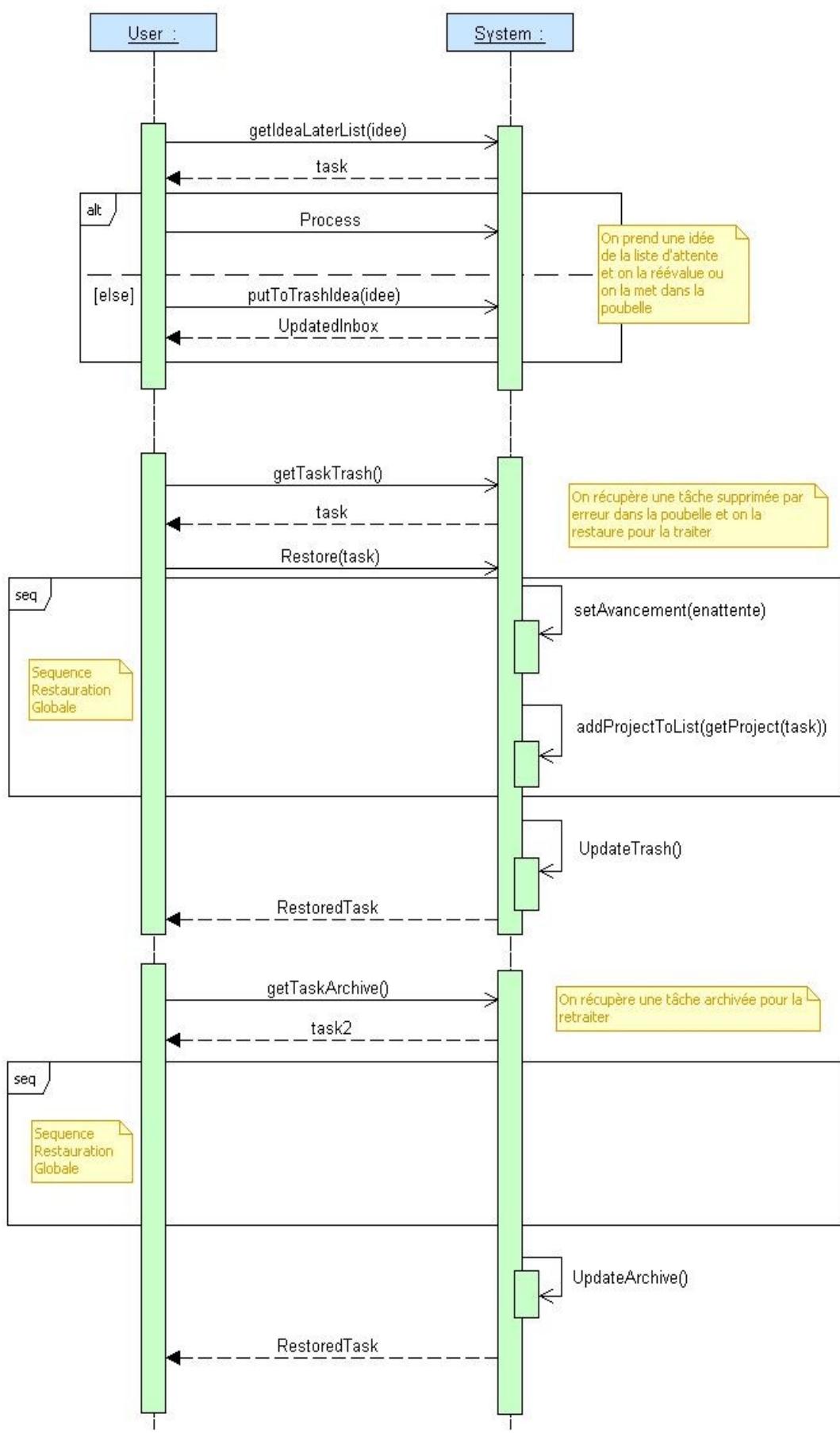
10- l'utilisateur réalise qu'il a archivé ou supprimé une tâche par erreur, il sélectionne une tâche dans les archives ou dans la poubelle, puis décide de la restaure. Elle reprend alors la place qu'elle avait avant d'être supprimée.

2 - Diagramme d'activité :

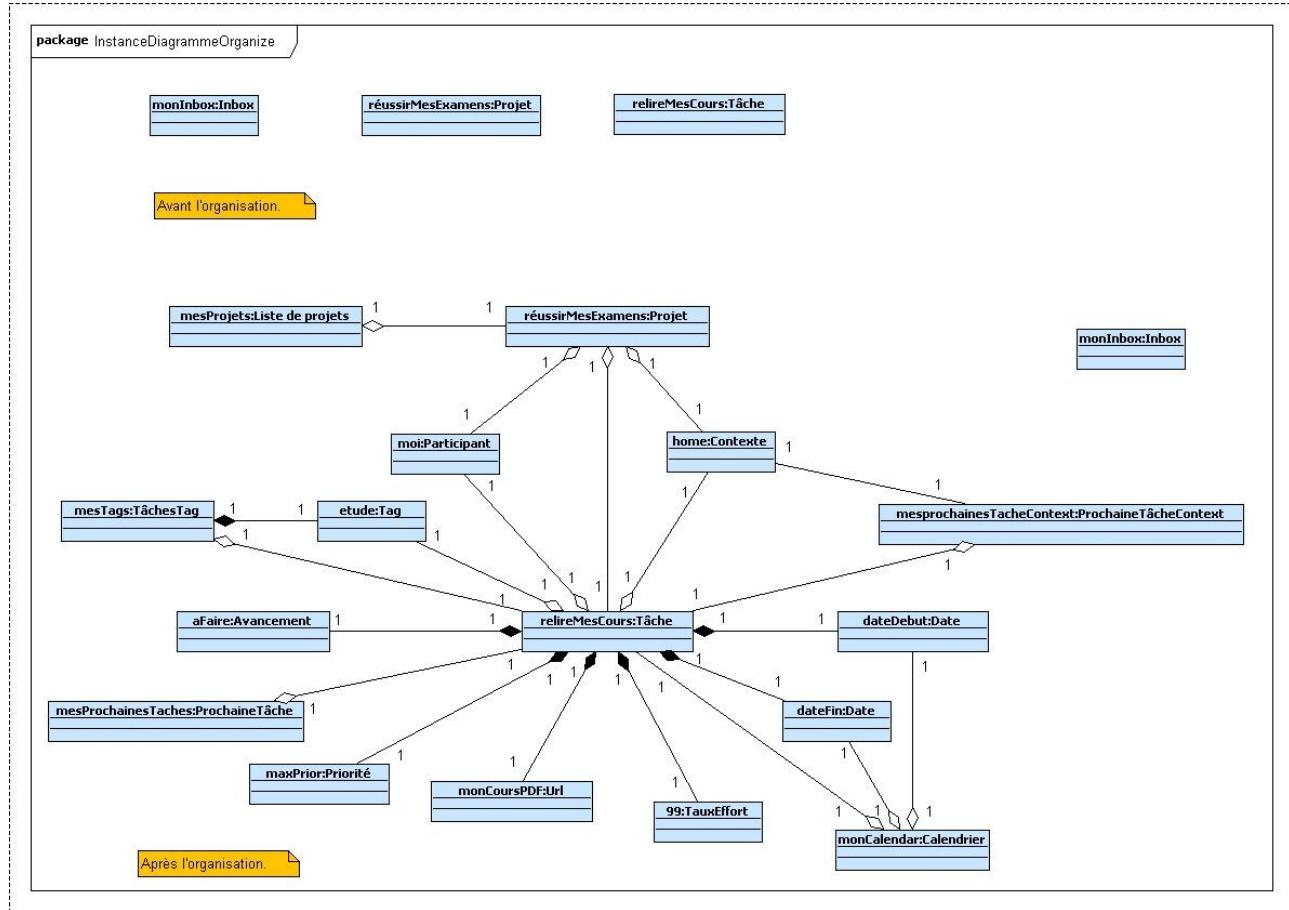


3 – Instantané correspondant



sd Organize2

4 - Instantané correspondant



D.Review

1 – Canevas de Cockburn

INFORMATIONS CARACTÉRISTIQUES

But dans ce contexte :

Passer en revue l'avancement des tâches et des projets. En fonction de l'environnement actuel (contexte...), déterminer la tâche à réaliser en priorité.

Portée :

La liste des projets, la liste des prochaines tâches à réaliser, la liste des prochaines tâches à réaliser par contexte, la liste des tâches déléguées.

Niveau :

Top

Préconditions :

Il existe des tâches et des projets à réaliser.

Condition de succès :

Une tâche réalisable a été trouvée.

Condition d'échec :

Aucune tâche à réaliser n'a été trouvée.

Acteur primaire :

L'utilisateur.

Déclencheur :

L'utilisateur souhaite déterminer les tâches à exécuter.

SCÉNARIO DE SUCCÈS PRINCIPAL

- 1- l'utilisateur passe en revue le contexte dans lequel il se situe (maison, travail...)
- 2- l'utilisateur trouve la liste des tâches/projets associés à ce contexte.
- 3- l'utilisateur détermine la tâche la plus pertinente (effort, date, priorité) à réaliser dans le contexte courant.

EXTENSIONS**SOUS-VARIATIONS**

- 1- l'utilisateur passe en revue les tâches triées par date d'échéance (grâce au calendrier), puis détermine la tâche qu'il veut réaliser.
- 2- l'utilisateur passe en revue les tâches triées par taux d'effort demandé (liste des prochaines tâches), puis détermine ensuite la tâche qu'il veut réaliser.
- 3- l'utilisateur passe en revue les tâches triées par tag (liste TâchesTag), puis détermine ensuite la tâche qu'il veut réaliser.
- 4- l'utilisateur détermine un projet auquel il veut se consacrer, observe alors toutes les tâches d'un projet, puis détermine ensuite la tâche qu'il veut réaliser.

E. do**1 – Canevas de Cockburn****INFORMATIONS CARACTÉRISTIQUES****But dans ce contexte :**

Réaliser une tâche

Portée :

La liste des projets, la liste des prochaines actions, la liste des prochaines, la liste des tâches déléguées, la liste.

Niveau :

Top

Préconditions :

Il existe des tâches et des projets à réaliser.

ProchaineTache.getInstance().taches->notEmpty()

Condition de succès :

Une tâche a été réalisée, elle se trouve dans les archives.

Soit t1 la tâche que l'utilisateur vient de réaliser. On peut alors écrire la condition de succès comme suit :

ProchaineTache.getInstance().taches->excludes(t1)
and t1.Avancement.getValue()=='Terminée'
and Archive.getInstance().taches->includes(t1)

Condition d'échec :

Aucune tâche n'a été réalisée.

ProchaineTache.getInstance().taches->includes(t1)
or t1.Avancement.getValue()<>'Terminée'
or Archive.getInstance().taches->excludes(t1)

Acteur primaire :

L'utilisateur.

Déclencheur :

L'utilisateur souhaite exécuter une tâche.

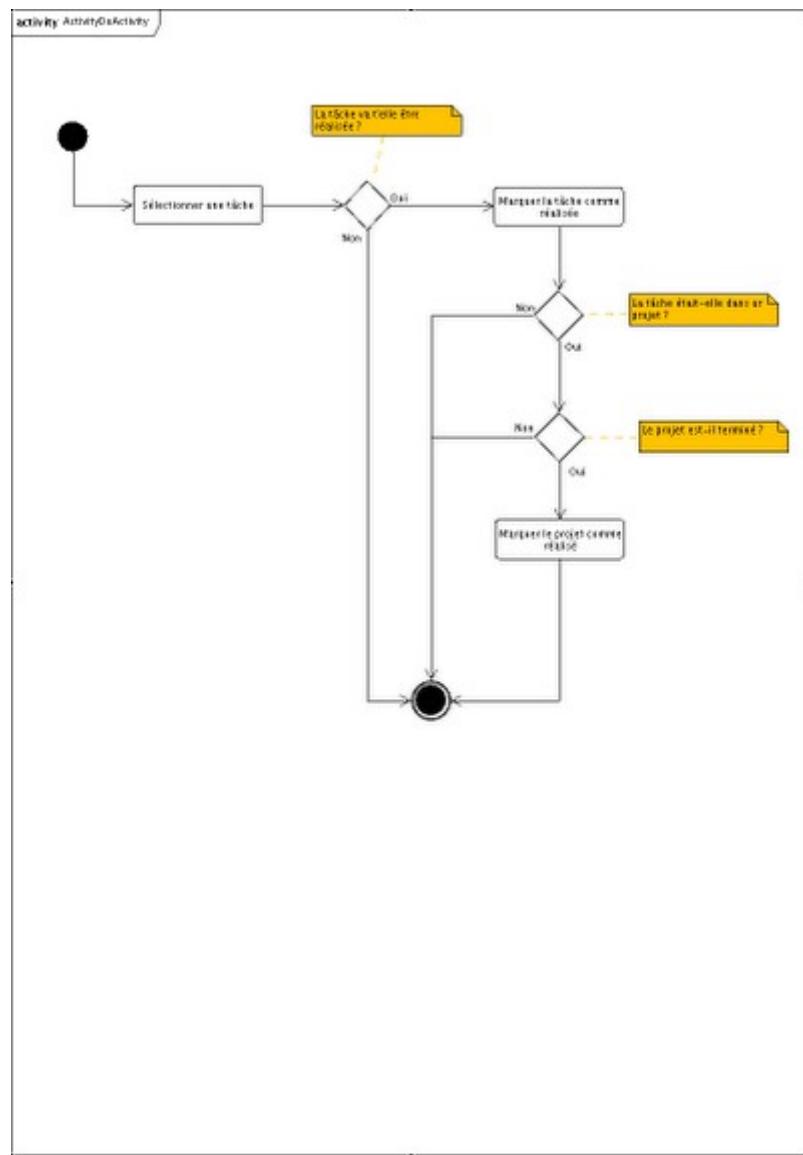
SCÉNARIO DE SUCCÈS PRINCIPAL

- 1- l'utilisateur sélectionne la tâche de son choix.
- 2- l'utilisateur réalise cette tâche.
- 3- l'utilisateur marque la tâche comme réalisée.
- 4- la tâche est déplacée dans les archives

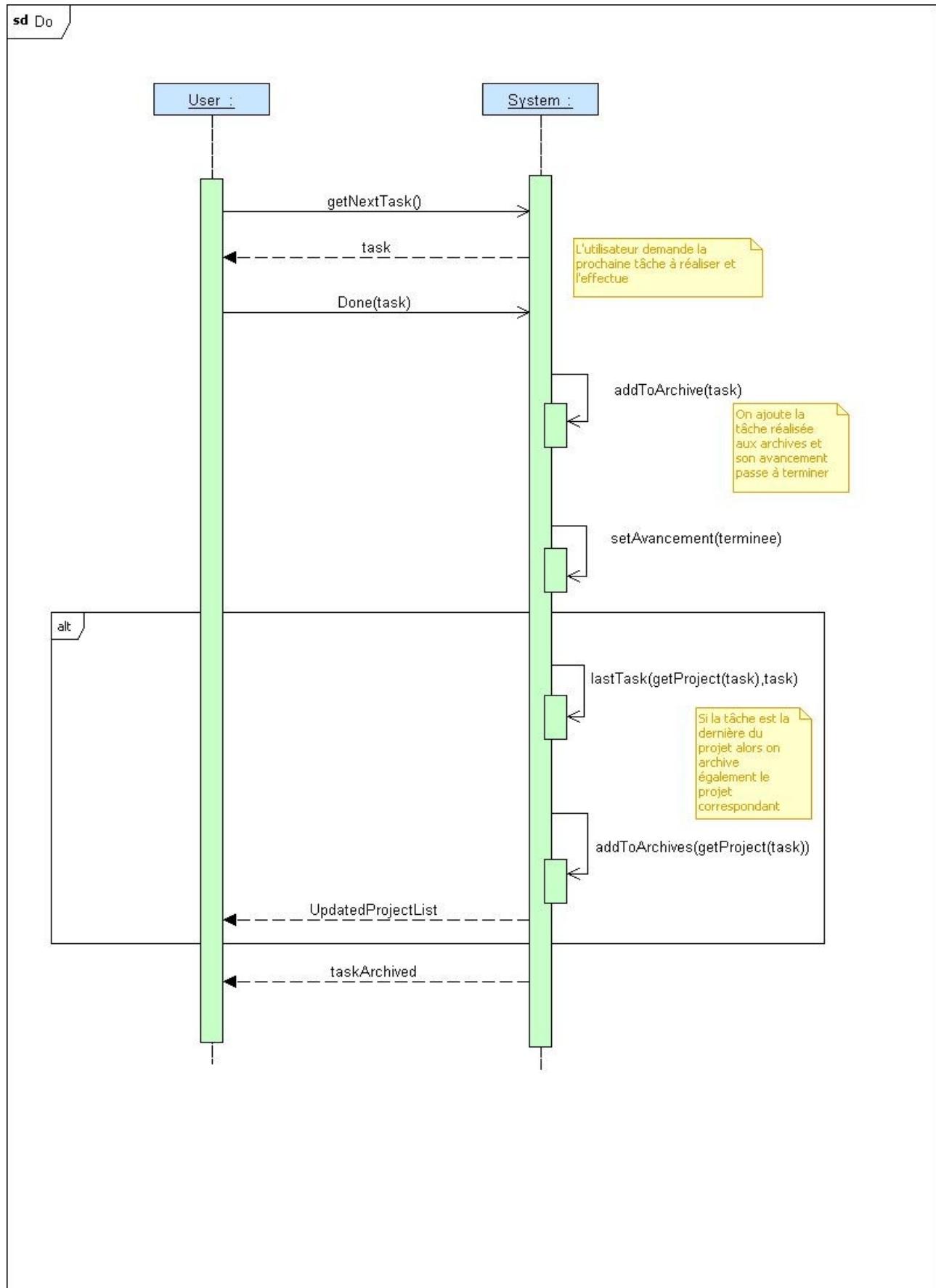
EXTENSIONS**SOUS-VARIATIONS**

- 1- la tâche était la dernière tâche du projet, l'utilisateur marque le projet comme terminé. Le projet entier doit alors être déplacé dans les archives.

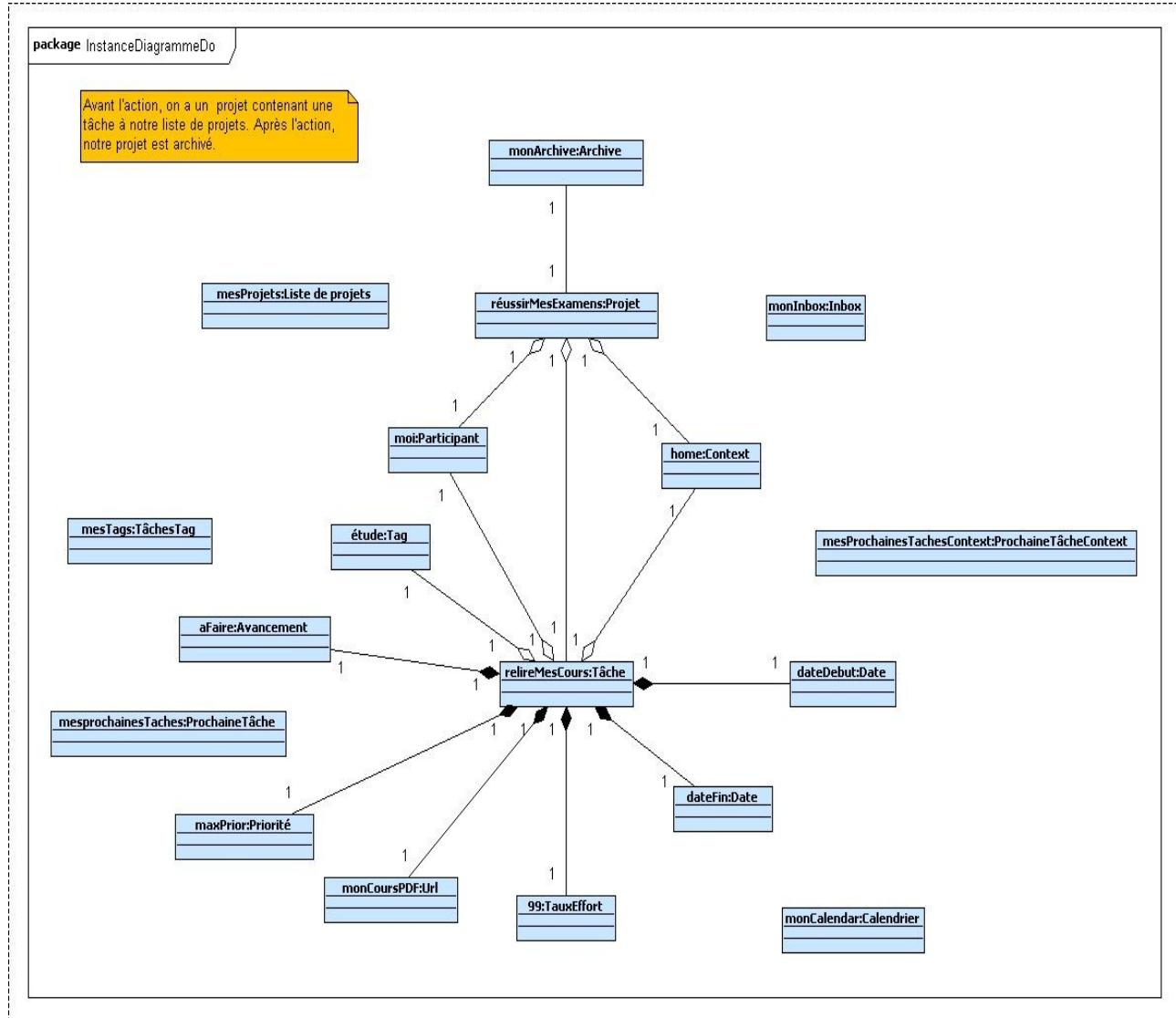
2 - Diagramme d'activité :



3 - Diagramme de séquence



4 - Instantané correspondant



Conclusion

Ce premier livrable nous a permis d'analyser la méthodologie GTD et d'en comprendre tous les rouages et fonctionnalités. Ce travail d'analyse est indispensable pour concevoir une manière efficace de représenter toutes les informations utiles à la méthodologie GTD. Dans le second livrable, nous commencerons à nous intéresser aux besoins propres à l'implémentation dont nous avons la charge, à savoir le serveur central auquel toutes les applications GTD se connecteront pour récupérer les données de l'utilisateur.

Chapitre 2

Livrable 2 : Spécification d'exigences logicielles

2.1 Introduction

Dans ce livrable, nous tâcherons de déterminer l'ensemble des besoins liés à la mise en oeuvre du serveur GTD, tant au niveau des fonctionnalités à concevoir que des contraintes à poser sur l'application (sécurité, performance...).

Au cours de cette introduction, nous définirons l'objectif précis de ce livrable, en exposant l'audience à laquelle ce document est destiné ainsi que les références utilisées. Nous présenterons également l'organisation du chapitre.

2.1.1 Objectif

L'objectif de ce livrable est d'identifier les différents besoins de l'application, à la fois au niveau des fonctionnalités à concevoir (à partir de cas d'utilisations précis) que sur un plan non-fonctionnel (établissement de contraintes sur la sécurité, la fiabilité...).

Notre application (un serveur GTD) devant communiquer avec d'autres applications clientes, une attention particulière sera portée sur les critères de disponibilité, de performance et de sécurité.

2.1.2 Conventions

On définira dans cette partie tous les besoins de l'application. Les critères choisis et les notions-clés apparaîtront en gras.

2.1.3 Audience

Ce document est avant tout destiné aux concepteurs et administrateurs du serveur GTD, ainsi qu'aux concepteurs des applications GTD clientes. Il pourra également servir de base à d'éventuelles mises à jour ou évolution des fonctionnalités.

2.1.4 Portée du document

Ce document spécifie les besoins d'un serveur GTD, interrogé par différentes applications clientes et devant gérer la cohérence des données GTD utilisateurs. Il devra fournir plusieurs modes de communication et une interface claire.

2.1.5 Définitions, acronymes et abréviations

GTD : Getting Things Done
serveur GTD : application chargée de centraliser les données des applications GTD clientes

2.1.6 Références

Références relatives à la méthodologie GTD :

1. Wikipedia sur la méthodologie GTD :
http://en.wikipedia.org/wiki/Getting_Things_Done,
2. Autre approche de la méthodologie GTD :
<http://www.scribd.com/doc/2164710/Getting-Things-Done-The-Science-behind-StressFree-Productivity>
3. Site du créateur de la méthodologie GTD :
http://www.davidco.com/what_is_gtd.php.

Références utilisées pour l'identification des besoins logiciels :

1. How to write a software requirements specification :
<http://www.microtoolsinc.com/Howsrs.php>
2. Wikipedia :
http://en.wikipedia.org/wiki/Software_Requirements_Specification

Références exposant les besoins d'un serveur de données :

1. Besoins d'un serveur SQL :
<http://www.mssqltips.com/tip.asp?tip=1456>
2. Sécurité pour un serveur de données :
http://docs.google.com/gview?a=v&q=cache:cdIkziURT9kJ:csrc.nist.gov/publications/nistpubs/800-123/SP800-123.pdf+server+security&hl=fr&gl=fr&pid=bl&srcid=ADGEESjrF08ZQ_svKzw5Hk1vrPAELSoXYF1AP20anXEsaQbszh94KOHR1AF-vGVU58f6h6pNfPZ6ZxWCOQQmzfF-Kde7XIwbDv111Adh-V5PlxA1CIzQ9E5TxaoQ9M&AFQjCNH_Ep0v6fisbgFa_9cNYLQ44jPzUw

2.1.7 Organisation du chapitre

Dans ce chapitre, nous commencerons par effectuer une description générale de l'application voulue, afin d'identifier les limites de notre analyse des besoins. On identifiera notamment les différentes classes d'utilisateurs du serveur GTD, l'environnement matériel nécessaire à son bon fonctionnement ainsi que les fonctionnalités que l'on repousse à une prochaine phase d'analyse.

Nous exposerons ensuite, dans une seconde partie, l'ensemble des besoins fonctionnels du serveur GTD, mis en évidence par des cas d'utilisation précis.

Nous terminerons par spécifier l'ensemble des besoins non-fonctionnels de l'application, en insistant sur les besoins en terme de sécurité, de fiabilité et de performance.

2.2 Description générale

Nous nous plaçons ici dans un contexte distribué, dans lequel différentes applications GTD clientes communiquent avec un serveur GTD chargé de stocker des données potentiellement concurrentes. Dans cette partie, nous effectuerons une description du comportement du serveur GTD, dont nous devons faire l'analyse complète avant de passer à son implémentation.

2.2.1 Perspectives du produit

L'application décrite ici est un serveur GTD devant être capable de fonctionner avec différentes applications GTD clientes sur un intranet (clients locaux) ou dans un contexte distribué (clients WEB).

2.2.2 Fonctions du produit

Bien que les fonctionnalités précises du serveur seront établies dans la deuxième partie, on peut tout de même identifier les fonctions clés que le serveur GTD devra proposer :

1. Gestion des actions concurrentes entre applications clientes (voir Besoins Non-Fonctionnels - Fiabilité)
2. Gestion de la persistance des données utilisateurs
3. Gestion sécuritaire de la communication avec les applications clientes (voir Besoins Non-Fonctionnels - Sécurité)
4. Proposer des outils d'analyse statistique et une interface paramétrable à l'administrateur du serveur

Ces fonctionnalités seront développées dans les prochains chapitres.

2.2.3 Caractéristiques et classes d'utilisateurs

Les utilisateurs du serveur GTD peuvent être de deux natures :

1. L'administrateur du serveur GTD : il aura la possibilité de paramétrier le serveur (choix de la stratégie utilisée pour gérer les actions concurrentes, gestion des backups de la base de données...) et de prendre connaissance de données statistiques, et ne sera par conséquent concerné que par l'interface graphique (voir Utilisabilité).
On a donc à faire à un type d'utilisateur qualifié en informatique et en connaissance du domaine (termes techniques liés au serveur, compréhension des graphiques statistiques...).
2. Les applications GTD clientes : elles interrogeront le serveur pour synchroniser leurs données, et pourront modifier leur contenu. La communication entre le serveur et les applications clientes sera spécifié dans un prochain livrable.

2.2.4 Environnement opérationnel

Matériel nécessaire au bon fonctionnement du serveur

A propos du matériel nécessaire à la création du serveur, les contraintes définies auparavant dans les points Performance et Efficacité imposent de disposer d'une machine avec des composants de bonne qualité, surtout en ce qui concerne la puissance du ou des processeurs (afin de fournir aux applications clientes un service optimal).

Nous attacheront également une attention particulière à la rapidité et à la capacité de la RAM et du ou des disques durs afin de ne pas trop limiter en espace le volume des données attribué à chaque client ou en temps les requêtes lancées par les applications GTD.

Le choix d'un matériel de qualité est aussi motivé par le fonctionnement quasi-permanent du serveur. La maintenance de ce dernier devra être extrêmement réduite pour ne pas entraver le bon fonctionnement de ce dernier.

D'un point de vue sécurité, il faudra prévoir un système de backup de la base de données ainsi que la mise en place d'un firewall bloquant tous les ports excepté ceux utilisés pour la communication avec les applications clientes.

Déploiement

Le déploiement du serveur GTD sur une machine se déroulera en deux étapes distinctes : le déploiement de l'application serveur sur la machine et la création d'un réseau permettant aux applications clientes de communiquer avec ce serveur.

Le déploiement de l'application serveur passera par l'installation d'un serveur d'applications (comme JBoss par exemple), qui permettra au serveur de faire tourner notre application, la mise en place d'une base de données afin de gérer la persistance des données utilisateur (via le framework HIBERNATE)

et bien sûr le déploiement de notre application sur ce serveur d'applications. On imposera un système d'exploitation qui sera déterminé par le langage de programmation utilisé.

En ce qui concerne la création d'un réseau, il suffira que le serveur soit connecté à Internet ou à un réseau Intranet.

2.2.5 Contraintes de conception et d'implémentation

Etant donné le fait que nous procédons à la spécification des besoins au niveau analyse, nous ne poserons pas de contraintes sur la conception ou l'implémentation du serveur si tôt. On supposera que le programme sera suffisamment performant et générique pour satisfaire à nos besoins.

2.2.6 Documentation utilisateur

Il nous faut là-encore différencier les utilisateurs humains (i.e l'administrateur du serveur) des applications clientes.

L'administrateur du serveur devra bénéficier, comme il sera exposé dans la partie Utilisabilité, d'un mécanisme d'aide en ligne et d'une documentation relative à l'utilisation de l'interface graphique du serveur.

Les développeurs des applications clientes auront besoin de nos spécifications, tant au niveau des données stockées que du mode de communication utilisé. Ces spécifications devront être fournies à tout concepteur d'une application GTD amenée à communiquer avec notre serveur.

On rédigera également un dossier d'analyse et de conception complet, en vue d'éventuelles mises à jour du serveur, et réaliserons une documentation complète afin que les développeurs puissent faire évoluer le serveur sans trop de difficulté.

2.2.7 Hypothèses et dépendances

Toutes les applications clientes dépendent du serveur GTD, en particulier pour gérer la synchronisation des données. On pourrait donc considérer que le serveur GTD n'a pas de dépendances.

Cependant, l'existence de notre serveur réside dans le fait qu'il interagit avec des applications clientes ; on peut donc considérer que le serveur dépend des applications GTD.

2.2.8 Exigences reportées

Notre serveur pourrait tenir à jour un historique des modifications effectuées sur les données par les applications clientes, afin de permettre à un utilisateur d'annuler certaines actions ou rétablir son système à une date antérieure.

Cependant, nous avons jugé que ce service, bien que très utile, n'était pas indispensable au bon fonctionnement du serveur. Compte tenu du peu de temps dont nous disposons, cette fonctionnalité pourra faire l'objet d'une prochaine mise à jour.

2.3 Fonctionnalités du logiciel

Dans cette partie, nous réaliserons différents cas d'utilisation précis relatifs à notre serveur GTD, au biais d'une description CockBurn, de diagrammes d'activité, et de séquence. Pour chaque cas d'utilisation, on identifiera les besoins fonctionnels qui lui sont liés, en essayant de procéder à une classification de ces besoins.

Les besoins fonctionnels ainsi établis serviront de guide lors de la conception et de l'implémentation, puisqu'ils correspondent aux fonctionnalités que notre serveur devra proposer.

2.3.1 Traitement des requêtes

Description

Use Case: 1 – Traitement des requêtes

CHARACTERISTIC INFORMATION

Goal in the context : Le serveur recevra différentes requêtes tout au long de son fonctionnement. Les fonctionnalités offertes par le serveur seront accessibles depuis l'extérieur uniquement grâce à des requêtes. Ainsi, nous considérerons que tous les cas d'utilisations qui seront exprimés à la suite de celui ci, correspondent à une requête valide dont on démarre le traitement.

Scope : Le serveur

Level : Tâche principale

Precondition : Le serveur est en train de fonctionner.

Success End Condition : Les requêtes sont valides, leur traitement démarre.

Failed End Condition : Les requêtes sont invalides.

Primary actor : Le serveur.

Trigger : Une requête envoyée par un utilisateur ou par l'administrateur est reçue par le serveur.

MAIN SUCCESS SCENARIO

1. Le serveur démarre.
2. Le serveur attend une connexion entrante.
3. Le serveur reçoit une connexion entrante.
4. Le serveur reçoit le login et le mot de passe de l'utilisateur.
5. Le serveur vérifie la validité du couple login / mot de passe.
6. Le serveur reçoit une requête.
7. Le serveur détermine à quelle fonction correspond la requête.
8. Le serveur délègue à la bonne fonction la requête.

EXTENSIONS

Si le couple login / mot de passe est invalide, le système retourne un message d'erreur et déconnecte l'utilisateur.

Si la requête est invalide, le système retourne un message d'erreur.

Si la requête est une requête de déconnexion, le serveur déconnecte l'utilisateur.

Exigences fonctionnelles

Code	Exigence	Type
REQ-1	Le serveur doit accepter des connexions extérieures	essentielle
REQ-2	Le serveur doit accepter les requêtes entrantes	essentielle
REQ-3	Le serveur doit renvoyer un message d'erreur si une requête est invalide	essentielle

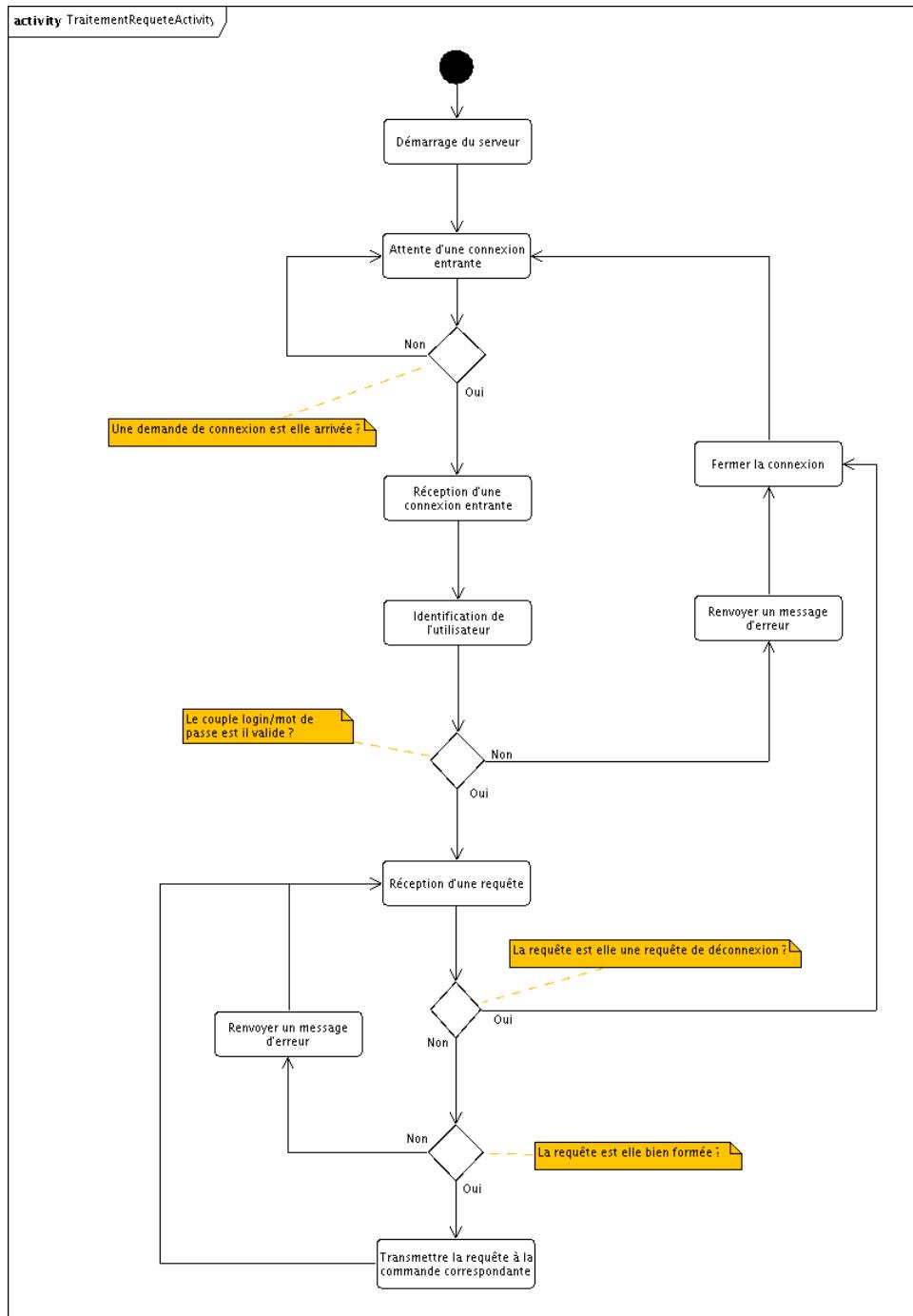


FIGURE 2.1 – Traitement des requêtes

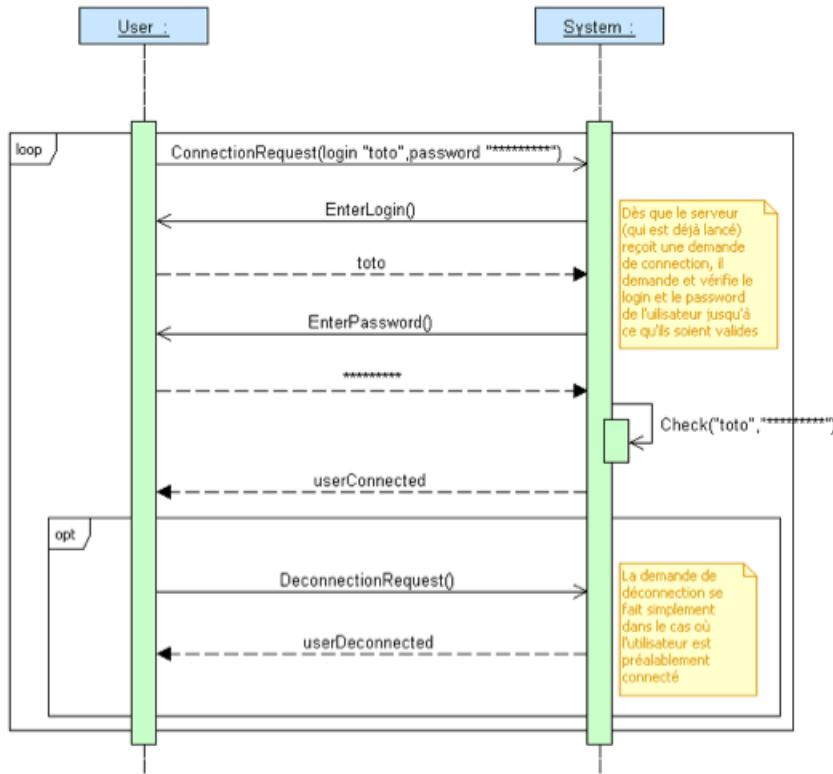


FIGURE 2.2 – Traitement des requêtes

2.3.2 Les requêtes de lecture

Description

Use Case: 2 – Les requêtes de lecture

CHARACTERISTIC INFORMATION

Goal in the context : Le serveur va recevoir différentes requêtes tout au long de son fonctionnement. Parmi ces différentes requêtes, les requêtes de lecture représentent toutes les requêtes qui demandent au serveur une information sans modification de l'état des données sur le serveur. Toutes les requêtes de lecture définies lors du chapitre 1, auront un comportement similaire qui sera défini ici.

Scope : Le serveur

Level : Sous fonction.

Precondition : Le serveur est en train de fonctionner, une requête de lecture bien formée est arrivée.

Success End Condition : Le résultat est renvoyé à l'émetteur de la requête.

Failed End Condition : Le résultat n'est pas renvoyé.

Primary actor : Le serveur.

Trigger : Une requête de lecture envoyée par un utilisateur ou par l'administrateur est reçue par le serveur.

MAIN SUCCESS SCENARIO

1. Le serveur reçoit une requête de lecture des idées d'un utilisateur.
2. Le serveur vérifie que les paramètres sont valides.
3. Le serveur collecte les idées pour répondre à la requête.
4. Le serveur organise ces idées dans une liste triée par date de création.
5. Le serveur enregistre dans son log l'action effectuée.
6. Le serveur renvoie le résultat.

EXTENSIONS

Si la requête est invalide, le système retourne un message d'erreur.

Exigences fonctionnelles

Code	Exigence	Type
REQ-4	Le serveur doit renvoyer un message d'erreur si une requête est invalide	essentielle

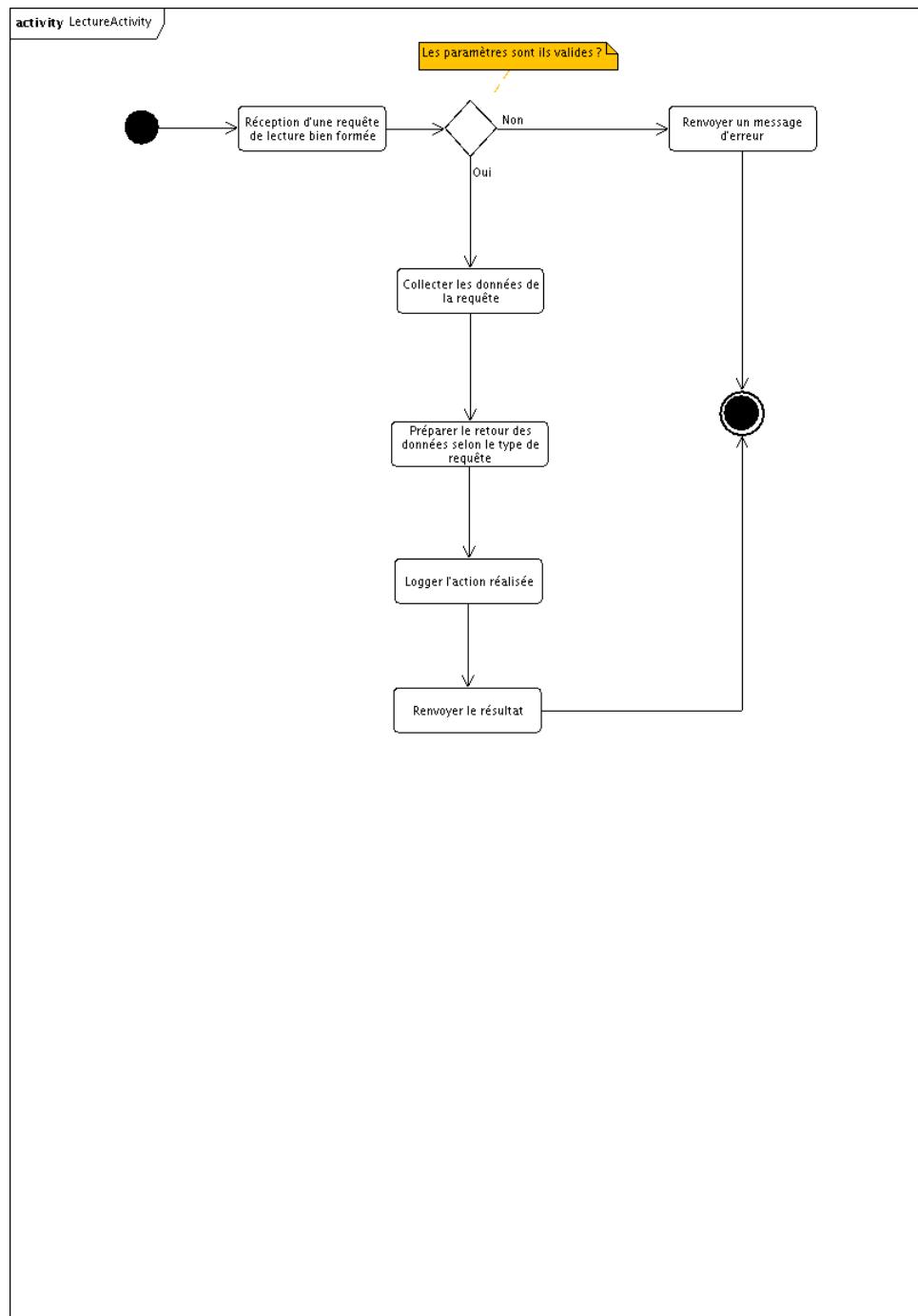


FIGURE 2.3 – Les requêtes de lecture

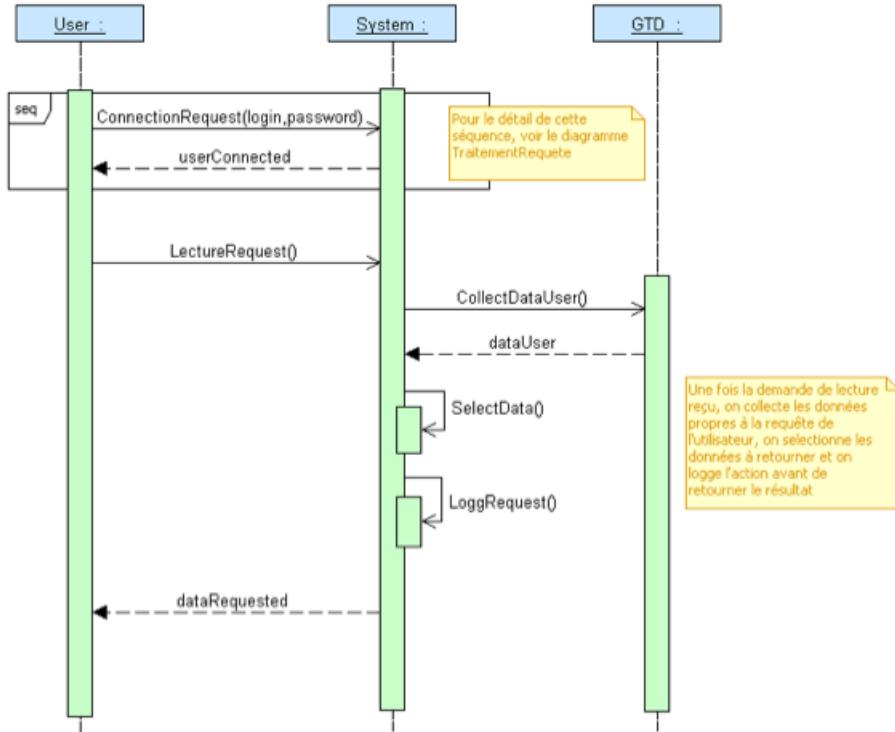


FIGURE 2.4 – Les requêtes de lecture

2.3.3 Les requêtes de création, de mise à jour et de suppression

Description

Use Case: 3 – Les requêtes de création, de mise à jour et de suppression

CHARACTERISTIC INFORMATION

Goal in the context : Le serveur va recevoir différentes requêtes tout au long de son fonctionnement.

Parmi ces différentes requêtes, les requêtes de création, de mise à jour et de suppression représentent toutes les requêtes qui demandent au serveur de modifier les données stockées sur le serveur.

Scope : Le serveur

Level : Sous fonction.

Precondition : Le serveur est en train de fonctionner, une requête de modification bien formée est arrivée.

Success End Condition : Le résultat est renvoyé à l'émetteur de la requête.

Failed End Condition : Le résultat n'est pas renvoyé.

Primary actor : Le serveur.

Trigger : Une requête de modification envoyée par un utilisateur ou par l'administrateur est reçue par le serveur.

MAIN SUCCESS SCENARIO

1. Le serveur reçoit une requête de modification du nom d'idée d'un utilisateur.

2. Le serveur vérifie que les paramètres sont valides.
3. Le serveur récupère l'idée modifiée pour répondre à la requête.
4. Le serveur enregistre dans son log l'action effectuée.
5. Le serveur renvoie le résultat.

EXTENSIONS

Si la requête est invalide, le système retourne un message d'erreur.

Exigences fonctionnelles

Code	Exigence	Type
REQ-4	Le serveur doit renvoyer un message d'erreur si une requête est invalide	essentielle

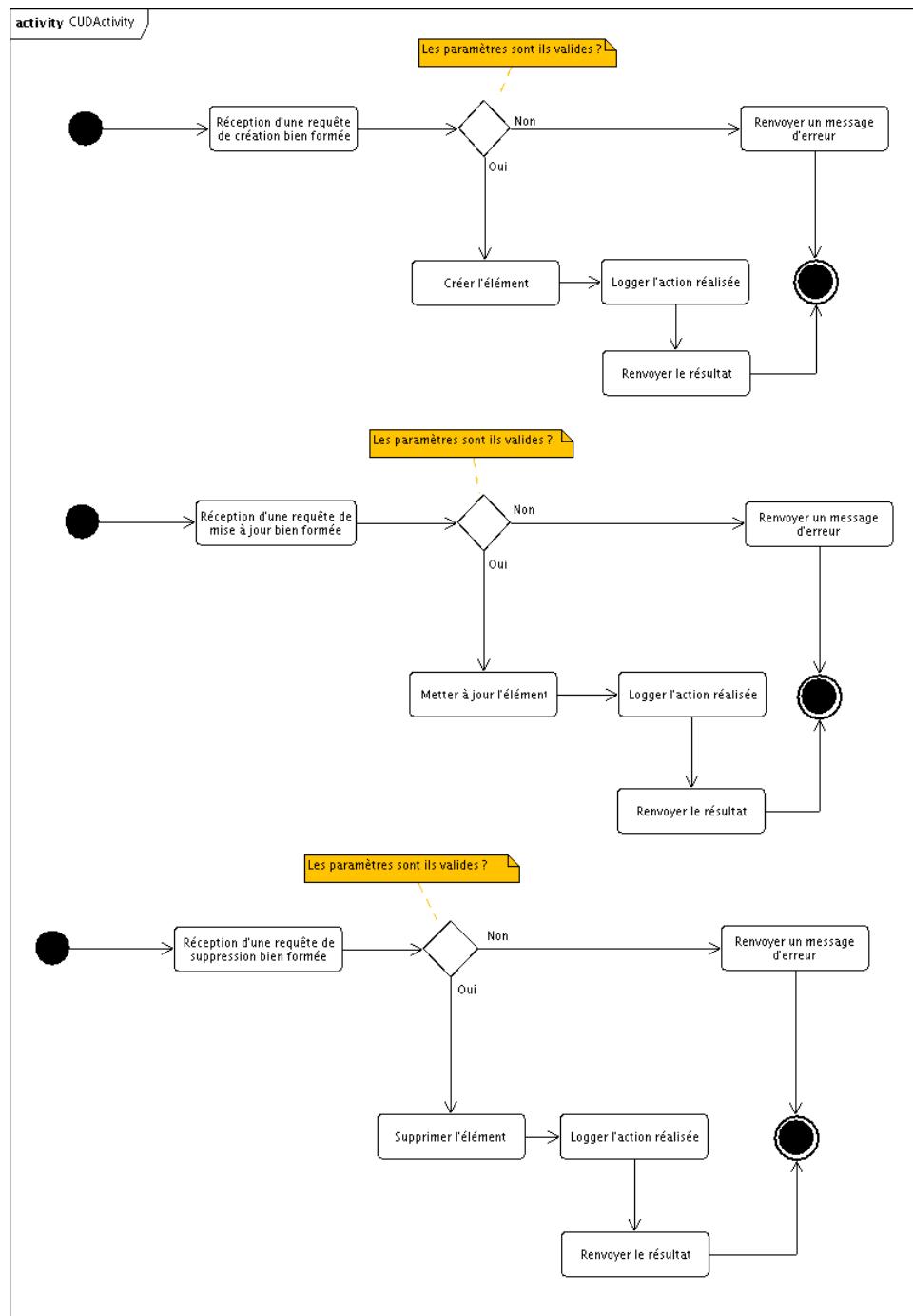


FIGURE 2.5 – Creation, mise à jour et lecture

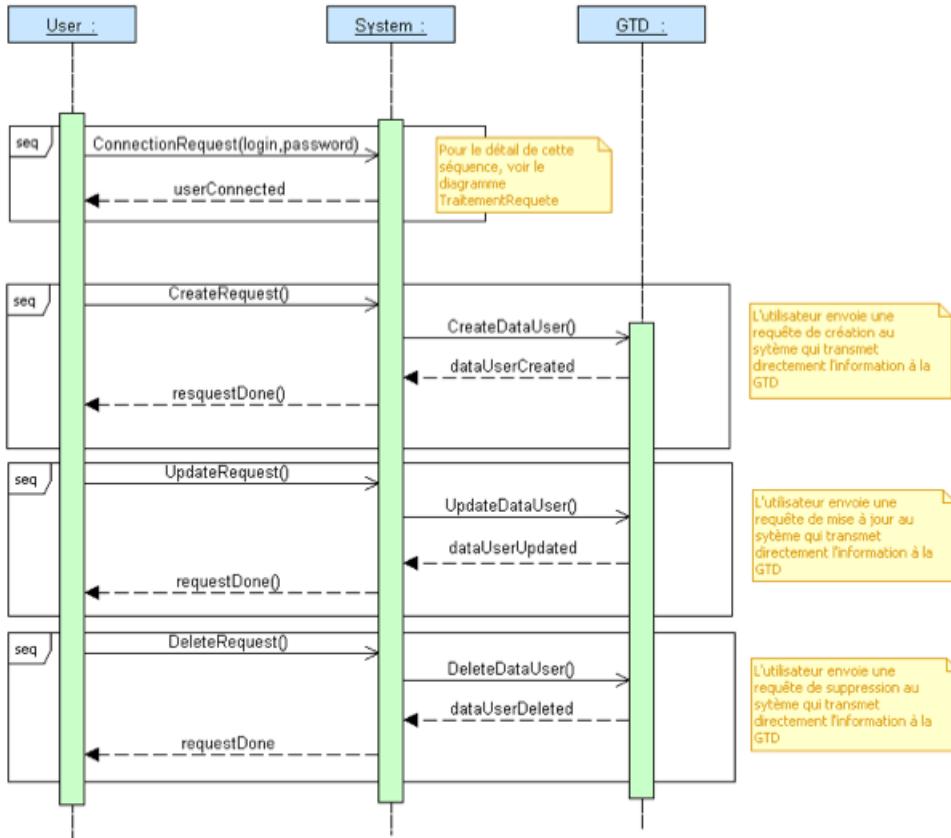


FIGURE 2.6 – Creation, mise à jour et lecture

2.3.4 Renommer le pseudo de l'utilisateur

Use Case: 4 – Renommer le pseudo de l'utilisateur

CHARACTERISTIC INFORMATION

Goal in the context : Modifier le pseudo de l'utilisateur.

Scope : Le compte.

Level : Subfunction.

Precondition : La connexion avec le serveur a bel et bien été effectuée, et une requête bien formée a été reçue.

Success End Condition : La connexion avec le serveur a bel et bien été effectuée, et une requête bien formée a été reçue.

Failed End Condition : Le pseudo de l'utilisateur n'a pas été modifié.

Primary actor : L'utilisateur.

Trigger : Le serveur receptionne une requête de demande de renommage d'un pseudo (voir cas d'utilisation 1).

MAIN SUCCESS SCENARIO

1. Le serveur vérifie le format du nouveau pseudo.

2. Il vérifie l'unicité du nouveau pseudo.
3. Il met à jour les tâches et projets de l'utilisateur avec le nouveau pseudo de l'utilisateur.
4. Il met à jour le compte de l'utilisateur en renommant son pseudo.

EXTENSIONS

- 1 Si le format du nouveau pseudo est incorrect, le serveur renvoie un message d'erreur. L'application cliente pourra réitérer la saisie.
 - 2 Si le nouveau pseudo existe déjà, le serveur renvoie un message d'erreur. L'application cliente pourra réitérer la saisie.
 - 3 Si la mise à jour des tâches et projets ne s'est pas déroulée correctement, le serveur renvoie un message d'erreur.
-

Exigences fonctionnelles

Code	Exigence	Type
REQ-6	Le serveur doit vérifier que les paramètres sont corrects	essentielle
REQ-7	Le serveur doit renvoyer un message d'erreur en cas d'erreur	essentielle

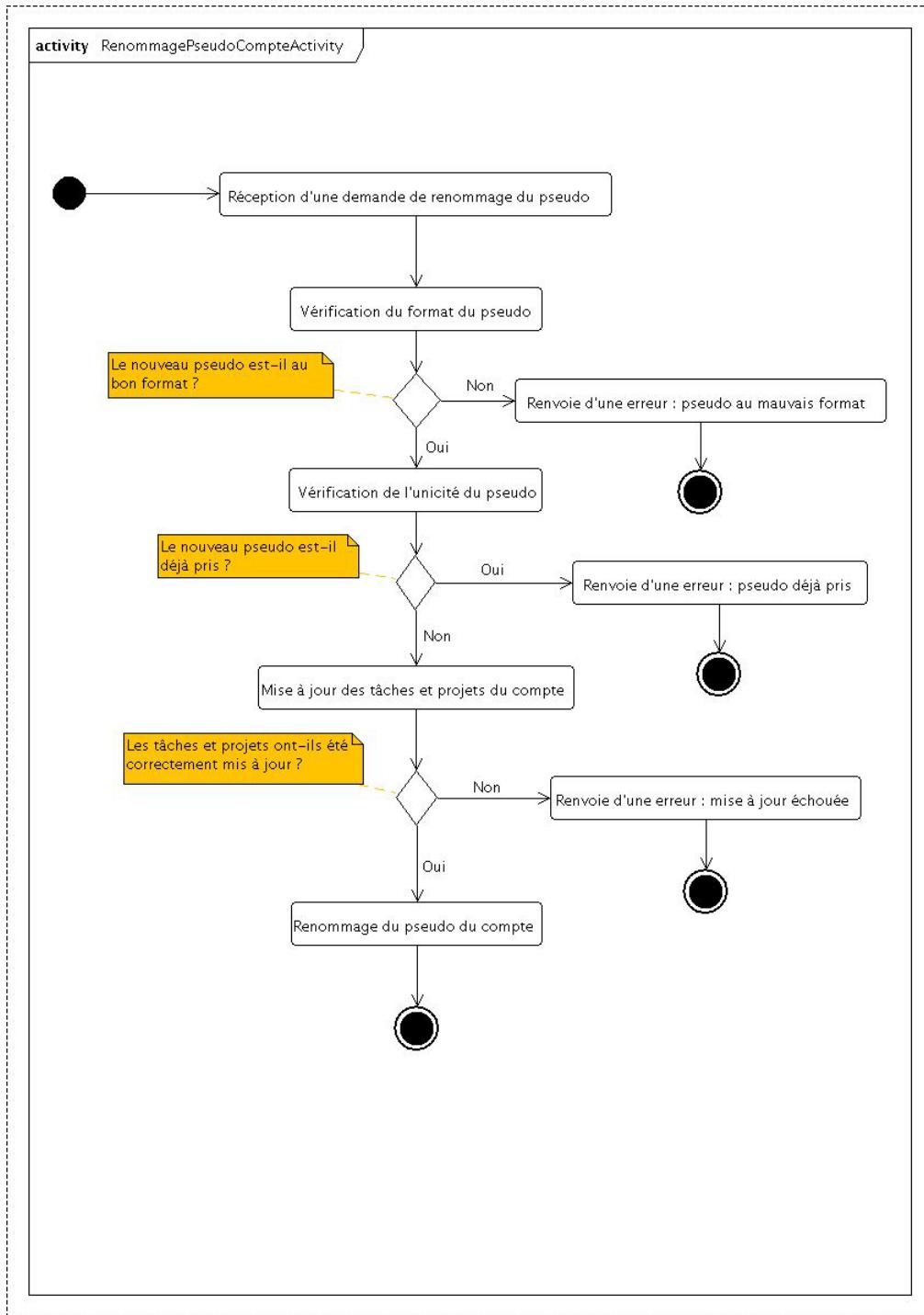


FIGURE 2.7 – Renommage du pseudo de l'utilisateur

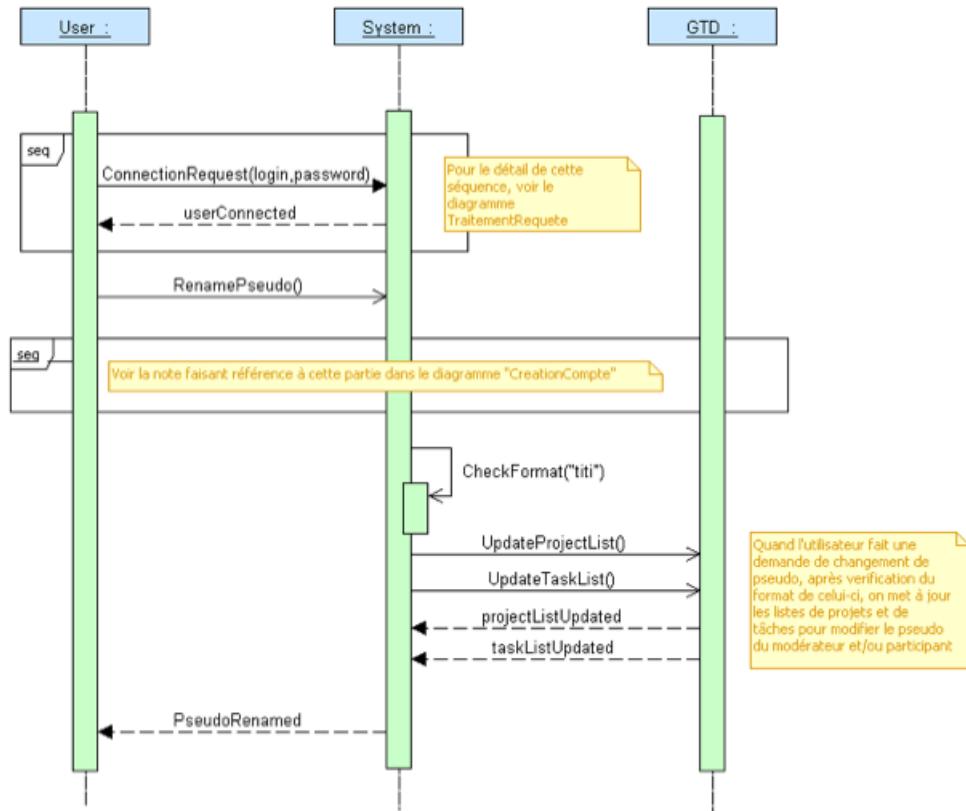


FIGURE 2.8 – Renommage du pseudo de l'utilisateur

2.3.5 Créer un compte

Use Case: 5 – Crée un compte

CHARACTERISTIC INFORMATION

Goal in the context : Crée un nouveau compte utilisateur.

Scope : La liste des participants.

Level : Subfunction.

Precondition : La connexion avec le serveur a bel et bien été effectuée, et une requête bien formée a été reçue.

Success End Condition : Le nouveau compte a été créé.

Failed End Condition : Le nouveau compte n'a pas été créé.

Primary actor : L'utilisateur.

Trigger : Le serveur reçoit une demande de création de nouveau compte (voir cas d'utilisation 1).

MAIN SUCCESS SCENARIO

1. Le serveur teste si le login existe déjà.

2. Il vérifie le format du login, du mot de passe et de toutes les informations du compte (pseudo de l'utilisateur...).
3. Il crée le compte et l'ajoute à la liste des participants.

EXTENSIONS

- 1 Si le login existe déjà, le serveur renvoie un message d'erreur. L'application cliente pourra réitérer la saisie.
 - 2 Si le format des informations du compte est incorrect, le serveur renvoie un message d'erreur. L'application cliente pourra réitérer la saisie.
-

Exigences fonctionnelles

Code	Exigence	Type
REQ-8	Le serveur doit vérifier que les paramètres sont corrects	essentielle
REQ-9	Le serveur doit renvoyer un message d'erreur en cas d'erreur	essentielle

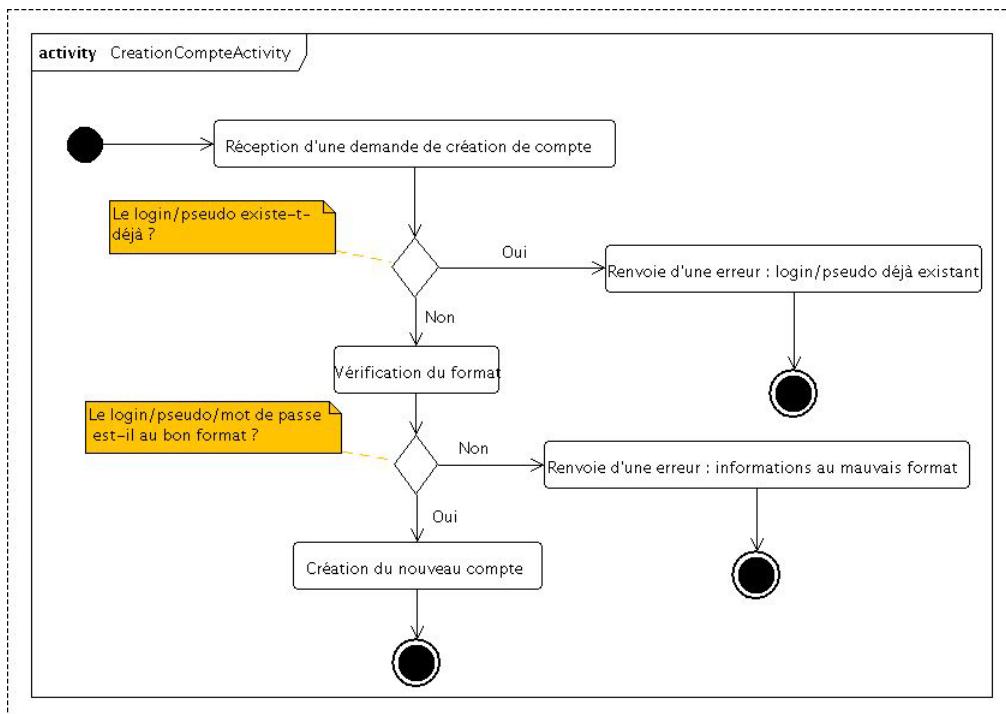


FIGURE 2.9 – Creation dun compte

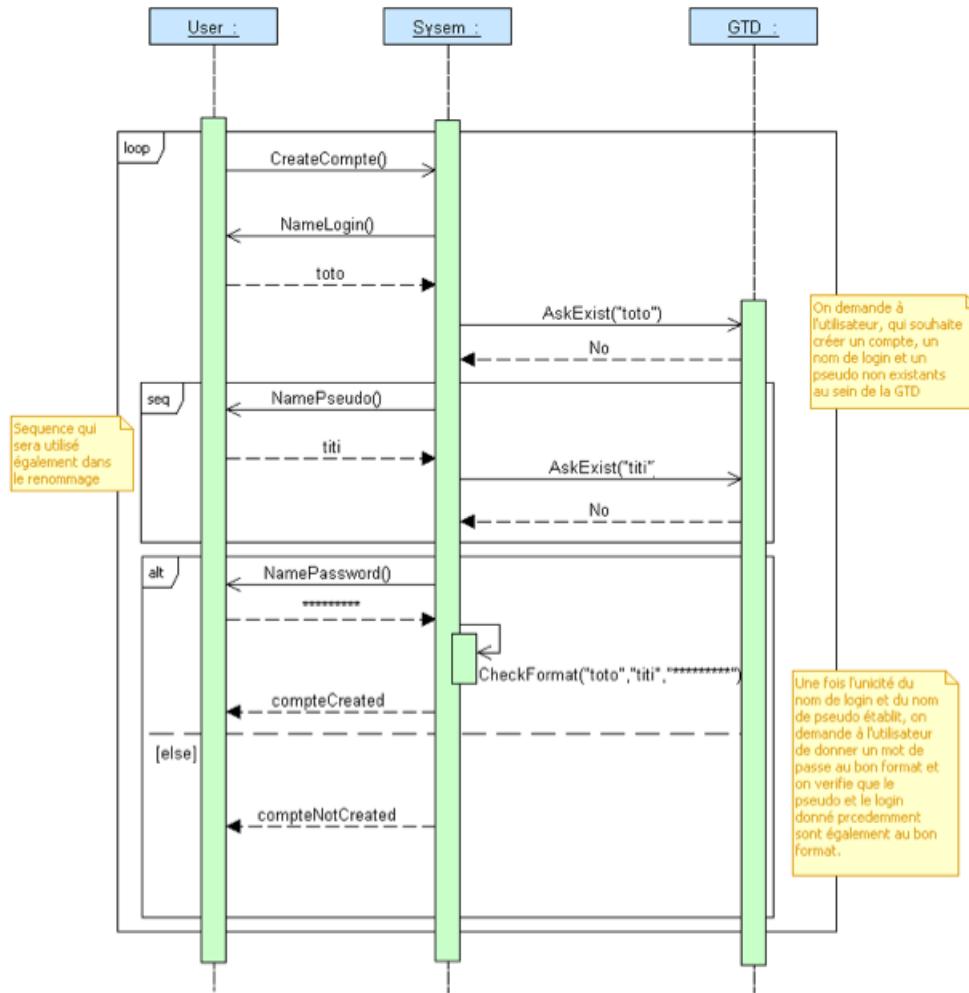


FIGURE 2.10 – Creation dun compte

2.3.6 Supprimer un compte

Use Case: 6 – Supprimer un compte

CHARACTERISTIC INFORMATION

Goal in the context : Supprimer le compte utilisateur.

Scope : La liste des participants.

Level : Subfunction.

Precondition : La connexion avec le serveur a bel et bien été effectuée, et une requête bien formée a été reçue.

Success End Condition : Le compte de l'utilisateur a bel et bien été supprimé.

Failed End Condition : Le compte de l'utilisateur n'a pas été supprimé.

Primary actor : L'utilisateur.

Trigger : Le serveur receptionne une requête de demande de suppression du compte (voir cas d'utilisation 1).

MAIN SUCCESS SCENARIO

1. Le serveur supprime les tâches et projets où l'utilisateur est modérateur.
2. Il met à jour les tâches et projets où l'utilisateur est simplement participant.
3. Il supprime le compte de l'utilisateur et le retire de la liste des participants.

EXTENSIONS

- 1 Si la suppression des tâches et projets ne s'est pas déroulée correctement, le serveur renvoie un message d'erreur.
 - 2 Si la mise à jour des tâches et projets ne s'est pas déroulée correctement, le serveur renvoie un message d'erreur.
-

Exigences fonctionnelles

Code	Exigence	Type
REQ-10	Le serveur doit renvoyer un message d'erreur en cas d'erreur	essentielle

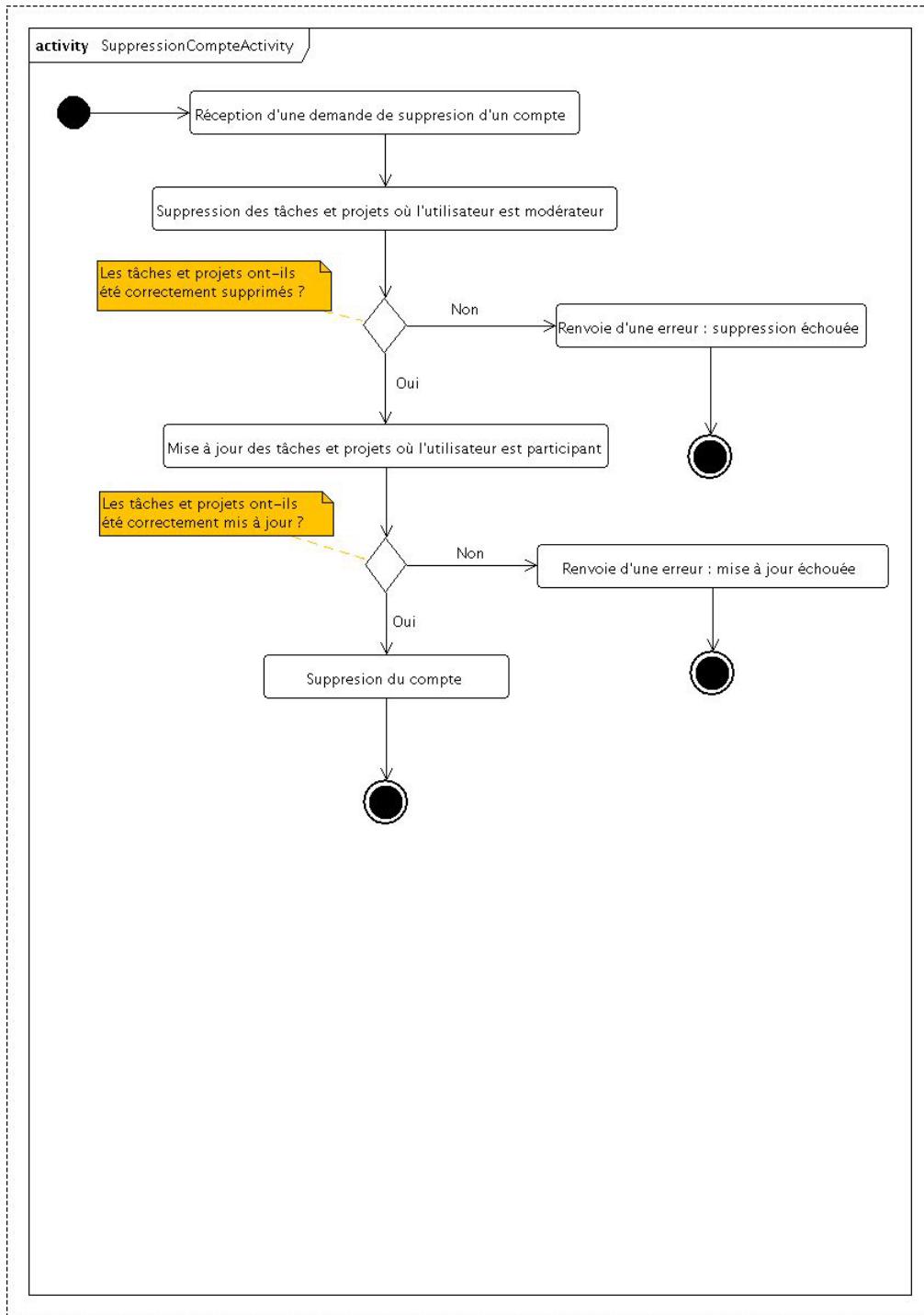


FIGURE 2.11 – Suppression d'un compte

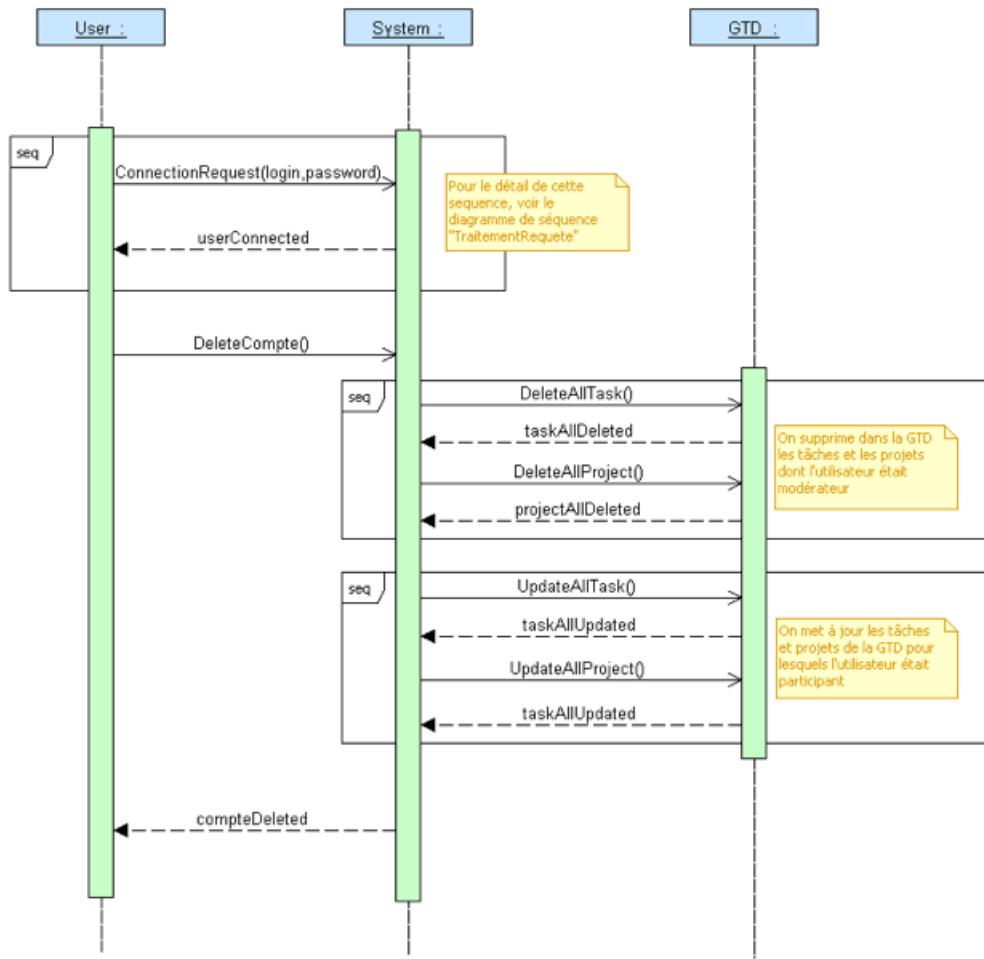


FIGURE 2.12 – Suppression d'un compte

2.3.7 Changer le mot de passe d'un utilisateur

Use Case: 7 – Changer le mot de passe d'un utilisateur

CHARACTERISTIC INFORMATION

Goal in the context : Modifier le mot de passe de l'utilisateur.

Scope : Le compte.

Level : Subfunction.

Precondition : La connexion avec le serveur a bel et bien été effectuée, et une requête bien formée a été reçue.

Success End Condition : Le mot de passe de l'utilisateur a bel et bien été modifié.

Failed End Condition : Le mot de passe de l'utilisateur n'a pas été modifié.

Primary actor : L'utilisateur.

Trigger : Le serveur réceptionne une requête de demande de changement du mot de passe (voir cas d'utilisation 1).

MAIN SUCCESS SCENARIO

1. Le serveur vérifie le format du nouveau mot de passe.
2. Il met à jour le compte de l'utilisateur en modifiant son mot de passe.

EXTENSIONS

- 1 Si le format du nouveau mot de passe est incorrect, le serveur renvoie un message d'erreur. L'application cliente pourra réitérer la saisie.
-

Exigences fonctionnelles

Code	Exigence	Type
REQ-11	Le serveur doit vérifier que les paramètres sont corrects	essentielle
REQ-12	Le serveur doit renvoyer un message d'erreur en cas d'erreur	essentielle

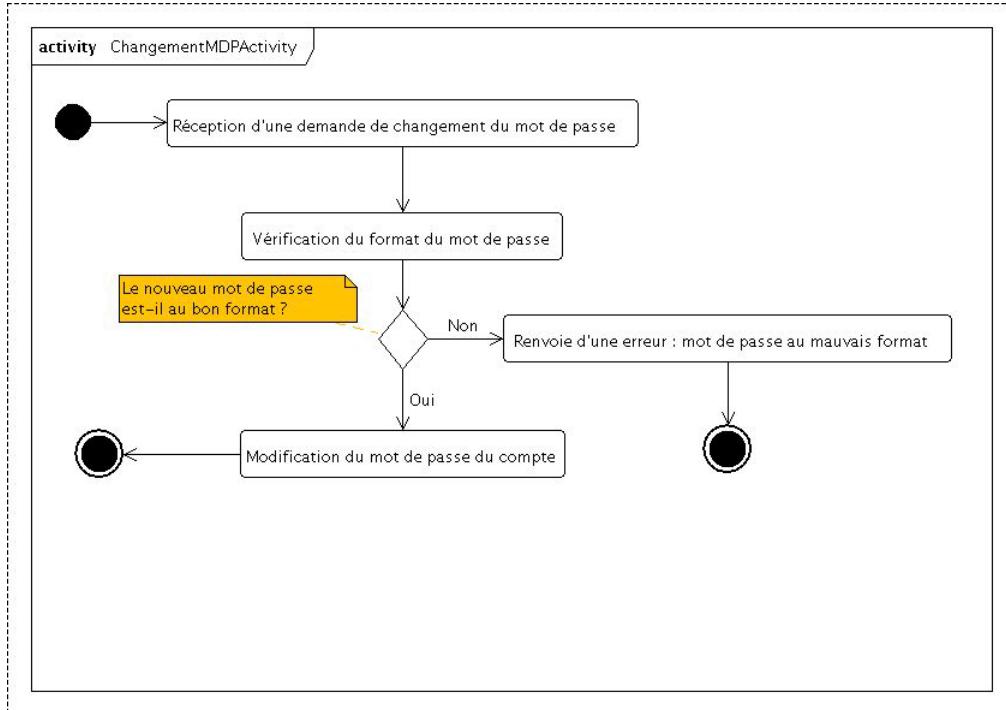


FIGURE 2.13 – Changement du mot de passe de l'utilisateur

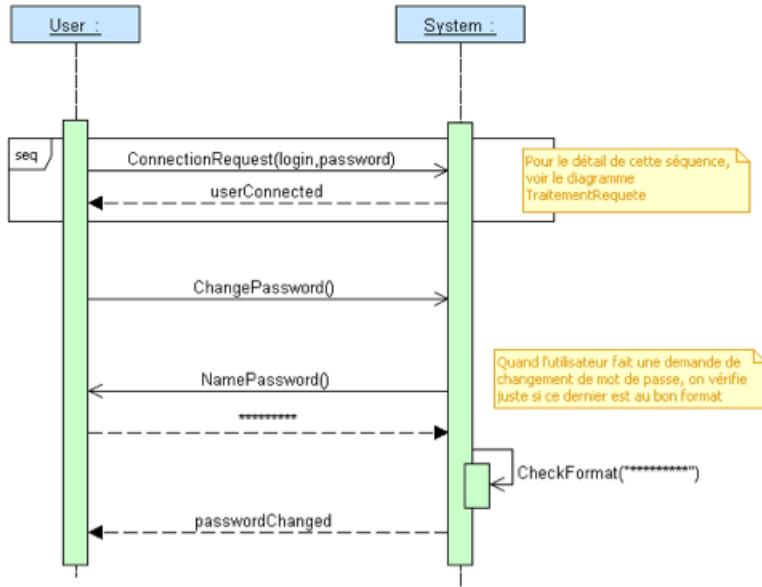


FIGURE 2.14 – Changement du mot de passe de l’utilisateur

2.4 Autres exigences non-fonctionnelles

Après avoir établi tous les besoins du serveur GTD relatifs aux fonctionnalités à concevoir, intéressons-nous maintenant aux contraintes non-fonctionnelles que nous pouvons imposer. Pour déterminer chacun des besoins, nous nous sommes davantage basés sur l’étude de la situation et notre bon-sens plutôt que sur des standards de règles relatives à un serveur de données.

Ce choix peut être motivé par la simplicité de l’application (les besoins non-fonctionnels d’un serveur de données étant trop poussés par rapport au sujet) et le temps qui nous est attribué pour implémenter ces contraintes.

D’autres exigences pourraient être apportées lors d’une prochaine phase d’analyse des besoins.

2.4.1 Utilisabilité

On s’intéressera ici à l’ergonomie de notre programme, au sens large.

Identification des acteurs potentiels de l’interface graphique du serveur GTD

Comme établi précédemment, le seul acteur humain qui utilisera notre programme sera l’administrateur, les clients ne pouvant interagir avec notre serveur que par le biais d’une application.

On se place donc ici dans un contexte d’utilisation où l’ensemble des utilisateurs sont des spécialistes du domaine : l’administrateur est censé comprendre les termes techniques ou les graphes statistiques (courbes, diagrammes...) proposés.

Efficacité exigée pour l’interface graphique

En terme d’efficacité, notre interface graphique devra permettre une gestion poussée du serveur. En particulier, l’administrateur devra pouvoir :

1. configurer un certain nombre de paramètres du serveur (choix de la stratégie utilisée pour gérer les actions concurrentes, gestion des backup de la base de données...),
2. modifier ses données personnelles (contact, nom, login ...),
3. prendre connaissance de données statistiques relatives à l'utilisation du serveur (requêtes effectuées, historique...).

Afin d'obtenir une efficience optimale, on mettra en place un certain nombre de raccourcis (icônes, raccourcis clavier...) en minimisant la description de chaque tâche, l'administrateur étant censé maîtriser les actions qu'il effectue. L'action sera donc exécutée en un temps minimal et avec un effort réduit.

Clarté de l'information, aide en ligne et temps de formation

On prendra également soin à fournir des résultats clairs et concis, afin de satisfaire au mieux l'administrateur. En particulier, des mécanismes d'infobulles et de documentations en ligne seront mis en place afin que le temps d'apprentissage nécessaire à la bonne utilisation de l'interface soit aussi court que possible.

Cependant, un court temps de formation (quelques heures maximum) sera nécessaire pour une bonne productivité de l'utilisateur, ce qui est très courant pour les interfaces de gestion de serveur, souvent complexes.

Dans le but d'améliorer l'utilisabilité de l'interface client, on pourra mettre en place sur le serveur un Forum de question/réponse où les utilisateurs des applications clientes pourront s'adresser aux administrateurs du serveur.

2.4.2 Fiabilité de l'application

Disponibilité du serveur GTD

On s'attachera à ce que l'application serveur fonctionne 24 heures sur 24 et 7 jours sur 7, et ce pour le rendre la plus disponible possible. En effet, les applications clientes devront à tout moment pouvoir interagir avec le serveur pour garantir à l'utilisateur final une bonne disponibilité.

Gestion des erreurs : prise en compte de la concurrence des actions utilisateurs

Notre serveur GTD étant fréquemment amené à gérer des actions provenant de différents utilisateurs mais portant sur les mêmes données, il doit permettre un traitement efficace des actions concurrentes. En effet, plus le nombre d'applications clientes connectées au serveur augmente et plus les actions concurrentes augmentent et risquent de poser problème.

Une fois les erreurs identifiées, on utilisera des algorithmes de résolution de conflit pour gérer ces actions concurrentes de manière déterministe (guidée par des critères formels).

On ne peut donc pas considérer ces actions concurrentes comme des échecs, puisque notre serveur sera à même de les corriger.

Identification des échecs potentiels de l'application

Les échecs sont intimement liés à l'environnement physique sur lequel tourne le serveur GTD. Ces échecs peuvent être graves (congestion du réseau, panne matérielle...), et sont, de par leur nature, imprévisibles.

Cependant, toutes les données gérées par le serveur étant sauvegardées dans une base de données non-embarquée et pouvant être copiées par mesure de précaution, un tel échec ne remet pas en cause la fiabilité de l'application. La récupération de notre serveur sera basée sur un rechargeement de la base de données dans un état cohérent.

Durée moyenne de fonctionnement avant défaillance

De par la nature matérielle et donc imprédictible des échecs potentiels, il est impossible de fournir une durée moyenne de fonctionnement pour notre application.

Durée moyenne de rétablissement

Rétablissement le système implique de revenir à un état cohérent du serveur, via le chargement de données cohérentes. Un tel chargement s'effectue en peu de temps ; on peut donc prédire, pour une panne n'étant pas liée au réseau, un temps de rétablissement inférieur à la minute, redémarrage du serveur compris.

En revanche, pour des problèmes de réseau ou matériels (panne physique du disque dur du serveur, câbles défectueux...), la durée de rétablissement peut se révéler beaucoup plus longue, car elle nécessite un diagnostic et une intervention humaine. C'est le seul genre de défaillance qui peut se révéler critique, c'est à dire non solvable (par exemple si le disque dur est endommagé et que les données utilisateurs sont définitivement perdues). Pour palier à ce type d'anomalie, on pourrait imposer que le serveur envoie des backups à une machine distante.

2.4.3 Exigences de performance

Sans rentrer dans le détail des solutions que nous pourrons proposer lors de l'implémentation du serveur, nous listerons ici les besoins liés à la performance générale du serveur.

2.4.4 Rapidité du serveur

La rapidité du serveur est liée à trois paramètres :

1. le temps d'exécution des requêtes permettant d'obtenir, de modifier, d'ajouter ou de supprimer les tâches, projets ou idées (fournies par l'application de l'utilisateur). Puisqu'il s'agit de requêtes sur une base de données, nous avons la garantie qu'elles s'effectueront dans un temps satisfaisant (quasi-instantané, de l'ordre de la milliseconde).
2. le temps de traitement des opérations serveur. Par exemple, la gestion d'une modification concurrente sur une tâche. On imposera donc à nos algorithmes d'avoir une complexité minimale.
3. le temps de la transmission d'un message sur le réseau. En effet, plus ce temps est grand, plus les risques d'augmenter la concurrence des modifications sont importants.

On imposera que le temps de réponse du serveur suite à n'importe quelle action utilisateur doit être strictement inférieur à deux secondes. Pour arriver à un tel temps, plusieurs autres besoins doivent être imposés :

1. les mises à jour des données utilisateurs devront être unitaires, afin de réduire le temps de traitement des actions concurrentes (voir Fiabilité - gestion des erreurs).
2. Un débit (en terme de transactions traitées par seconde) devra être imposé. Cependant, ce débit dépendant grandement à la fois du type de transaction, de la puissance physique de la machine et de la complexité de nos algorithmes de traitement, il est impossible de définir un tel débit à ce niveau d'analyse.

On peut cependant imposer que le serveur devra être capable de traiter au moins une dizaine de transactions par secondes, afin de garantir une disponibilité suffisante à toutes les applications clientes.

3. Il en va de même pour la capacité en terme de nombre de client connectés simultanément que le serveur devra supporter, qui, bien que devant être fixée, et très difficile de définir avant d'avoir testé la complexité des traitements de l'application serveur. On peut tout de même imposer que le serveur doive supporter au moins une centaine de clients connectés simultanément, afin qu'un nombre suffisant d'utilisateurs puisse utiliser leur application GTD convenablement.

2.4.5 Efficacité du serveur

Le serveur devra être bien configuré, disposer d'une bande passante importante et tourner sur une machine suffisamment puissante pour être le plus disponible possible, et avec le plus petit temps de réponse possible.

Dans un cas idéal, on pourrait garantir la précision de nos performances, c'est à dire que pour une même action effectuée à différents moments on aura les mêmes performances d'exécutions. Cependant, la performance dépendant grandement du nombre de requêtes à traiter, de l'état du réseau et de nombreux autres paramètres externes, on ne pourra réellement certifier la précision des performances du serveur.

On s'attachera tout de même à ce que l'application serveur fonctionne 24 heures sur 24 et 7 jours sur 7, et ce pour le rendre la plus disponible possible.

Afin de ne pas ralentir le serveur GTD, on s'efforcera à détecter les différentes erreurs le plus tôt possible (dès réception du message), et ce afin de les identifier rapidement en vue d'éventuels corrections.

Etant donné que notre application sera déployée sur un serveur spécifique et dédié à cette application, la consommation des ressources est ici un facteur négligeable puisqu'au contraire il doit disposer d'un maximum de ressources dans le but d'être performant temporellement.

2.4.6 Maintenabilité

Notre application étant un serveur, il est souhaitable de minimiser le temps de mise à jour et de maintenance du serveur (quelques heures par an seraient idéales). Par conséquent, la maintenance du serveur consistera essentiellement à changer du matériel physique et faire évoluer les technologies utilisées (Serveur d'applications, Base de données...).

2.4.7 Exigences de sûreté

Puisque nous analysons les besoins d'un serveur utilisés par d'autres applications, il est évident que la sûreté repose avant tout sur la disponibilité du serveur (voir Fiabilité - Disponibilité du serveur GTD) et sur la cohérence de la base de données (notamment via un rétablissement des données en cas d'échec et le bon traitement des actions concurrentes).

2.5 Exigences de sécurité

La sécurité du serveur GTD pourra être définie en imposant les contraintes suivantes :

1. sécurité des données utilisateurs : on acceptera les opérations de CRUD (création, lecture, modification et suppression) d'idées, tâches et projets uniquement si cette opération est réalisée par le créateur (modérateur) de la tâche, de l'idée ou du projet. Ainsi, on garantit qu'un utilisateur ne perdra aucune donnée dont il est responsable sans qu'il en ait donné explicitement l'ordre.
On permettra cependant au participant de la tâche de mettre à jour son statut (dans le cas d'une tâche déléguée).
2. sécurité des opérations : afin de garantir la cohérence des données stockées par le serveur, on mettra en place un système de transactions : une opération ne sera validée que si toutes les opérations de la transaction qui lui sont associées sont réalisées avec succès.
3. gestion des actions concurrentes : notre serveur devra proposer un mécanisme de gestion des actions concurrentes (voir 3. Fiabilité).
4. sécurité de la connexion Application/Serveur : notre serveur n'acceptera aucune requête provenant d'un utilisateur non authentifié. On pourra par exemple imaginer un système de login permettant à un utilisateur de s'identifier.
5. cryptage des informations circulant sur le réseau : on utilisera un algorithme de cryptage des données, qui garantira l'impossibilité de récupérer des informations confidentielles en "sniffant" les messages circulant sur le réseau.

6. sécurité de la base de données : toutes les applications clientes devront passer par les méthodes (services) fournies par le serveur : en aucun cas on n'autorisera une modification directe de la base de données contenant les idées, tâches et projets. On peut également imaginer un cryptage des informations (par exemple à l'aide de SHA-512) dans la base de données, notamment en ce qui concerne les mots de passe ou autres données sensibles.

2.6 Classification des exigences fonctionnelles

Nous rappellerons ici les différents besoins fonctionnels que nous avons identifiés dans ce chapitre.

Code	Exigence	Type
REQ-1	Le serveur doit accepter des connexions extérieures	essentielle
REQ-2	Le serveur doit accepter les requêtes entrantes	essentielle
REQ-3	Le serveur doit renvoyer un message d'erreur si une requête est invalide	essentielle
REQ-4	Le serveur doit renvoyer un message d'erreur si une requête est invalide	essentielle
REQ-6	Le serveur doit vérifier que les paramètres sont corrects	essentielle
REQ-7	Le serveur doit renvoyer un message d'erreur en cas d'erreur	essentielle
REQ-8	Le serveur doit vérifier que les paramètres sont corrects	essentielle
REQ-9	Le serveur doit renvoyer un message d'erreur en cas d'erreur	essentielle
REQ-10	Le serveur doit renvoyer un message d'erreur en cas d'erreur	essentielle
REQ-11	Le serveur doit vérifier que les paramètres sont corrects	essentielle
REQ-12	Le serveur doit renvoyer un message d'erreur en cas d'erreur	essentielle

Chapitre 3

Livrable 3 : Spécification de l'interface du Serveur GTD

3.1 Définition des données GTD partagées par le serveur et les applications clientes

Voici un diagramme de classes représentant les données que le serveur GTD et les applications clientes seront amenées à échanger.

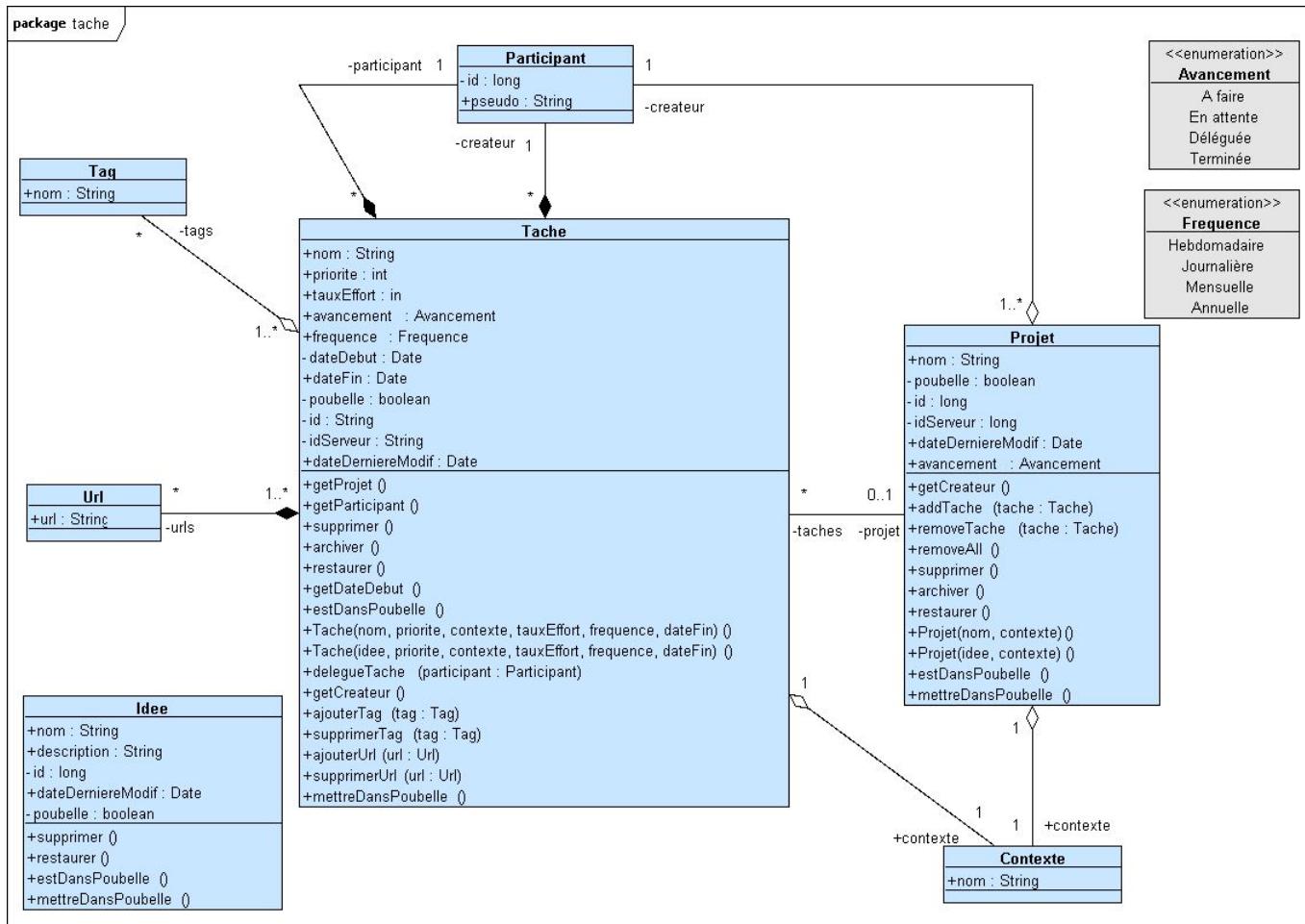


FIGURE 3.1 – Diagramme de classe

Pour des compléments d'information sur la démarche et la logique suivie pour obtenir ce diagramme, veuillez vous référer au premier livrable du serveur.

3.2 Fonctionnement général de la communication avec le serveur GTD

3.2.1 Les étapes fondamentales de la communication avec le serveur GTD

Notre serveur supportera les appels de méthodes via deux protocoles de communication : CORBA et RMI.

La communication avec le serveurs s'effectuera en 4 étapes clés :

1. La **récuperation du serveur** afin de pouvoir effectuer des appels de méthodes distants.

Cette récupération s'effectuera via l'interrogation d'une annuaire de services.

Exemple d'appel :

```
ServeurGTD serv = (ServeurGTD)Naming.lookup("rmi://" + host+ "/ServeurGTD_RMI");
```

2. **Identification** auprès du serveur : chaque utilisateur d'une application GTD possédera un compte, avec un login et un mot de passe, qu'il devra renseigner pour ce connecter.

Suite à cette identification, le serveur renverra un jeton de connexion (chaîne de caractères cryptée), identifiant l'utilisateur de manière unique et sécurisée. Ce jeton devra être passé en paramètre de tous les futurs appels au serveur, jusqu'à la déconnexion du client.

Pour plus de détail, veuillez vous référer au paragraphe décrivant la méthode `login`.

3. **Appel aux fonctionnalités** du serveur, comme par exemple l'insertion d'une tâche. Comme nous le verrons, la réponse renvoyée par le serveur repose sur les mécanismes d'identification et de CallBack (voir partie suivante).
4. **Déconnexion** auprès du serveur : le jeton de connexion fourni devient inutilisable.
Pour pouvoir effectuer de nouveaux appels au serveur, l'application cliente devra refaire une demande d'identification.

3.3 Détails de fonctionnement

3.3.1 Le mécanisme de CallBack

Issu du Design Pattern du même nom, le callback est un objet qui nous permettra de fournir plusieurs fonctionnalités simplement, pour cela regardons un court exemple.

Sans callback :

```
public class Main {  
    public static void main(String [] args) {  
        String port = args[0];  
        MonObjet obj = (MonObjet) c.lookup("rmi://" + port +"MyObject");  
  
        try {  
            Integer result = obj.donneMoiMonResult("mon_nom");  
            traiteur(result);  
        } catch (RemoteException e) {  
            traiteurException(e);  
        } catch (ResultException e) {  
            traiteurException(e);  
        }  
    }  
  
    public static void traiteur(Integer i) {  
        // traiter le resultat  
    }  
  
    public static void traiteurException(Exception ex) {  
        // traiter l'exception  
    }  
}  
  
public interface MonObjet {  
    public Integer donneMoiMonResult(String nom) throws ResultException;  
}
```

Sans mécanisme de callback, les clients seraient confronté à des méthodes synchrones et donc bloquantes en utilisant CORBA ou RMI, de plus le serveur serait aussi bloqué durant le traitement d'une requête, où alors il devrait mettre en place un système d'attente active (`while(true)...`) pour garder la pile d'exécution de l'appel venant du client, jusqu'au moment où la réponse est prête à être transmise, le serveur deviendrait ainsi pratiquement mono-utilisateur et les clients devraient créer de nouveaux threads pour chacune de leurs requêtes avec la gestion de la concurrence qui va avec. Le mécanisme de callback peut être vu comme un pattern observeur amélioré, en effet avec le pattern observeur, un objet observeur va s'enregistrer auprès d'un objet observé et il informe ce dernier (par son interface) que si un événement intéressant à lieu alors il doit le prévenir avec sa méthode `notify`. Avec le pattern callback, l'objet qui observe fourni un objet callback à l'objet qui est observé, lorsque ce dernier a une information intéressante à transmettre à l'observeur, alors il appelle la méthode de son choix sur l'objet callback.

Avec callback :

```
public class Main {
    public static void main(String [] args) {
        try {
            String port = args[0];
            MonObjet obj = (MonObjet) c.lookup("rmi://" + port +"MyObject");
            obj.donneMoiMonResultAsync("monNom", new CallBackClient());
        } catch (Exception e) {
            e.printStackTrace();
        }
    }

    public static void traiter(Integer i) {
        // traiter le resultat
    }

    public static void traiterException(Exception ex) {
        // traiter l'exception
    }
}

public class CallBackClient implements CallBack<Integer> {
    public CallBackClient() {
        UnicastRemoteObject.export(this);
    }

    public void onFailure(AsynchException ex) throws java.rmi.RemoteException {
        Main.traiterException(ex);
    }

    public void onSucces(Integer i) {
        Main.traiter(i);
    }
}

public interface MonObjet {
    public void donneMoiMonResultAsync( String nom, CallBack<Integer> callback );
}

public interface CallBack <T> extends java.rmi.Remote {
    public void onFailure(AsynchException ex) throws java.rmi.RemoteException;
    public void onSuccess(T i) throws java.rmi.RemoteException;
}
```

Dans le cas ci-dessus, il faut imaginer que le serveur reçoit l'appel de "donneMoiMonResultAsync", qu'il crée un nouveau thread avec le callback et le string, puis que la méthode donneMoiMonResultAsync se termine, coupant la pile d'exécution et redonnant la main au client. Il est à noter que dans la version avec callback, le bloc try catch n'est présent que pour les exceptions provenant de rmi (RemoteException) pas pour les exceptions provenant des méthodes du serveur qui elle serait transmise par l'appel à "onFailure".

```
public void donneMoiMonResultAsync( String nom, CallBack<T> callback ) {
    ThreadDeCalcul tdc = new ThreadDeCalcul(nom, callback);
    tdc.start();
}
```

Le thread "ThreadDeCalcul" prend alors 25min pour effectuer l'opération côté serveur, puis à la fin de ce calcul, il appelle la méthode "onSuccess" de l'objet callback, celui ci déclenche alors l'exécution du code "traiter(i)" côté client. Ce code est déclenché côté client car le callback n'est pas sérialisable, donc seul une référence sur le callback est passé au serveur, l'objet callback reste toujours côté client. Le serveur peut réaliser l'appel sur le callback car lorsque le client utilise la méthode lookup pour demander au registre rmi du serveur une référence sur le serveur distant, il s'enregistre sur le registre rmi aussi, permettant ainsi

à la méthode ”UnicastRemoteObject.export(this)” du constructeur du callback d’exporter une référence du callback dans le registre rmi, permettant alors au serveur de rappeler le client ultérieurement. Le principe s’applique aussi dans le cadre de CORBA.

3.3.2 Le mécanisme d’identification

Il existe deux mécanismes d’identification, le premier nous permet d’identifier un utilisateur, le second nous permet d’identifier une tâche, une idée ou un projet de manière unique. Lorsque l’utilisateur s’identifie sur le serveur, le serveur renvoi au client un string par le biais de son callback. Ce string sera le jeton d’identification, il correspond à un identifiant unique de l’utilisateur. Avec chaque requête, les clients devront réutiliser ce jeton d’identification afin de permettre au serveur de vérifier l’identité de l’utilisateur. Ce jeton sera codé grâce à l’algorithme de hachage SHA-512 il est donc quasiment impossible pour un client, de générer un autre jeton valide qui correspondrait à un client connecté sur le serveur.

Le second mécanisme d’identification permet d’identifier une tâche, un projet ou une idée de manière unique. Pour réaliser ceci le serveur affecte à une tâche un identifiant serveur sous la forme d’un string. Les clients n’ont jamais à modifier cet identifiant, de plus cet identifiant est codé en SHA-512 pour compliquer la tâche de ceux qui seraient tenter de le faire. Afin d’identifier la tâche localement, un identifiant local est disponible, les clients peuvent en faire ce qu’ils veulent dans l’absolu.

3.3.3 Mode de mise à jour

Afin de satisfaire le plus grand nombre de développeurs, et dans le but de permettre à l’utilisateur de gérer ses données comme il le veut, nous avons défini plusieurs modes de mise à jour pour les données du serveur.

Ce mode de mise à jour (UpdateMode) devra être passé en paramètre lors des appels aux méthodes d’update du serveur GTD. Les applications clientes seront libres de laisser l’utilisateur choisir son mode de mise à jour ou d’en utiliser un d’office, sans permettre à l’utilisateur de le modifier.

UpdateMode numéro un : FORCE

Dans ce mode de mise à jour, si le serveur détecte un conflit (par exemple un utilisateur voulant modifier une donnée dont il n’a pas la version la plus récente), il effectue quand même la mise à jour, sans prévenir l’utilisateur.

Aux applications clientes d’utiliser ce mode avec parcimonie.

UpdateMode numéro deux : WARNING

Dans ce mode de mise à jour, si le serveur détecte un conflit, il appelle l’action **onFailure()** du callback, en passant en paramètre un message d’erreur. Un dictionnaire des messages d’erreurs sera fourni aux développeurs ultérieurement.

Ce mode de mise à jour est similaire à un gestionnaire de dépôt : en cas de conflit, on prévient l’application cliente en indiquant le type d’erreur rencontré. L’utilisateur peut par exemple choisir de forcer la mise à jour, ou de synchroniser cette donnée avec le serveur GTD. Bien évidemment, l’application cliente peut prendre ces décisions à la place de l’utilisateur, ce dernier n’étant pas censé avoir les compétences nécessaires à la gestion des conflits.

UpdateMode numéro trois : PASSIVE

Nous ne pensons pas avoir le temps de proposer un tel mode de mise à jour, mais nous en faisons la description ici en vue d’éventuelles prochaines versions du serveur.

Dans ce mode de mise à jour, le serveur décide lui-même de la façon de gérer un conflit, sans que l'application cliente n'intervienne. Il peut être intéressant, mais son implémentation prendrait du temps (définir une politique de gestion des conflits) et n'est pas fondamentale.

3.4 Signature des méthodes de l'interface Serveur

3.4.1 Creation, mise à jour et suppression

Nous détaillerons ici l'ensemble des fonctionnalités proposées

Creation d'une idée

Description textuelle

Ajoute une idée créée localement à la base de données du serveur.

Signature de la méthode dans l'interface externe du serveur

```
public final void creerIdee(final Idee idee, final String identification, final CallBack<Idee> callback);
```

Le callback renvoie l'idée créée sur le serveur avec son identifiant serveur ou un refus de la demande de création de cette idée si une idée portant un identifiant serveur a été envoyée.

Pre et post-conditions OCL

```
context CommandeCreerIdee::execute()
pre
    -- L'idée doit avoir été créée localement, et pas récupérée
    -- suite à une synchronisation avec le serveur
    -- Il ne s'agit pas proprement parler d'une pre-condition,
    -- puisque notre serveur levera une exception
    self.idee.getIDServeur() == 0

    -- Cette méthode doit être appelée par un utilisateur identifié au
       serveur
    self.utilisateur != null

```

```
post
    -- Si l'idée est issue d'une synchronisation, le mécanisme de callback
       levera une exception de type Invalid Call.
    (( self.idee@pre.getIDServeur() <> 0)
     (self.callback.getMethodCalled() == "onfailure")
     (self.callback.getExceptionType() == "InvalidCallException") )

    -- Dans tous les autres cas :
    ||( ( self.idee@pre.getIDServeur() == 0)

        -- la création de l'idée a fonctionné, c'est à dire :
        -- que l'on a attribué un identifiant serveur à cette idée
        ( self.idee.getIDServeur() <> 0 )
        -- que l'idée ainsi créée appartient bien à l'utilisateur identifié
        ( IdeeService::getInbox(self.utilisateur)->includes(self.idee)
        -- on a retourné un message de validation
        ( self.callback.getMethodCalled() == "onsuccess" )
```

Mettre à jour une idée

Description textuelle

Met à jour une idée devant avoir été créée au préalable sur le serveur.

La résolution des conflits liés à cette mise à jour dépend du mode de mise à jour (UpdateMode) choisi.

Signature de la méthode

```
public final void updateIdee(final Idee idee, final UpdateMode mode, final String  
identification, final CallBack<Idee> callback);
```

Le callback renvoie l'idée mise à jour sur le serveur ou un refus de la demande de modification de cette idée si l'idée possède un identifiant serveur invalide par exemple. Un troisième cas peut également se présenter : si l'UpdateMode vaut "WARNING" et que la demande de modification génère un conflit (si l'idée a mettre à jour est plus ancienne que l'idée sur le serveur par exemple), le callback stockera un message relatif à ce conflit. On imagine que l'application cliente devra proposer à l'utilisateur de forcer la mise à jour ou de se mettre lui-même à jour.

Pre et post-conditions OCL

```
context CommandeUpdateIdee :: execute()  
pre  
    -- L'idée doit avoir été créée sur le serveur  
    -- Il ne s'agit pas proprement parler d'une pre-condition,  
    -- puisque notre serveur levera une exception  
    self.idee.getIDServeur() <> 0  
  
    -- Cette méthode doit être appelée par un utilisateur identifié  
    -- au serveur et étant le modérateur (créateur) de cette idée.  
    self.utilisateur <> null  
    IdeeService::getInbox(self.utilisateur)@pre->one(Idee i |  
        i.getIDServeur() == self.idee.getIDServeur())  
  
post  
    -- CAS 1 : Erreur dans l'appel  
    -- Si l'idée a été créée uniquement localement, c'est à dire :  
    -- Qu'aucun identifiant serveur ne lui a été attribué  
    ( ( self.idee@pre.getIDServeur() == 0  
  
    -- Ou que l'idée n'appartient pas à l'utilisateur  
    || IdeeService::getInbox(self.utilisateur)@pre->forAll(Idee i |  
        i.getIDServeur() <> self.idee.getIDServeur())  
  
    -- alors le mécanisme de callback levera une exception de type Invalid  
    -- Call.  
    ( self.callback.getMethodCalled() == "onfailure"  
    ( self.callback.getExceptionType() == "InvalidCallException" ) )  
  
    -- Dans tous les autres cas :  
    -- L'idée avait été créée sur le serveur  
    || ( ( self.idee@pre.getIDServeur() <> 0  
  
    -- L'idée designe appartenait bien à l'utilisateur  
    -- l'opérateur OCL "one" impose qu'un seul des éléments de la  
    -- collection satisfait la condition (équivalent à un "exists" renforcé)  
    IdeeService::getInbox(self.utilisateur)@pre->one(Idee i |  
        i.getIDServeur() == self.idee.getIDServeur()) )  
  
    -- CAS 2 : l'appel est correct mais un conflit a été détecté  
    -- et l'utilisateur est en mode "WARNING" => on renvoie ce conflit  
    ( ( (self.updateMode == "WARNING")  
  
    -- le fait qu'il existe sur le serveur une version plus récente de  
    -- l'idée que l'utilisateur veut modifier constitue un conflit
```

```

( self.idee@pre.getDateDeDerniereModification() <
  Inbox.getIdée(self.idee@pre.getIdServeur()) .
  getDateDeDerniereModification() )

-- le mecanisme de callback levera alors une exception de type Warning
( (self.callback.getMethodCalled() == "onfailure")
  (self.callback.getExceptionType() == "WarningException") ) )



-- CAS 3 : l'appel est correct, et :
-- soit l'utilisateur est en mode "FORCE" et il est possible qu'un
-- conflit ait eu lieu
|| ( ( (self.updateMode == "FORCE"))

-- soit l'utilisateur est en mode "WARNING" est aucun conflit ne doit
-- avoir eu lieu
|| ( (self.updateMode == "WARNING") (self.idee@pre.
  getDateDeDerniereModification()
>= IdeeService::getInbox(self.utilisateur).getIdée(self.idee@pre.
  getIdServeur())
  .getDateDeDerniereModification() ) )

-- la modification de l'idée a fonctionné, c'est à dire :
-- que l'on a effectué les modifications voulues l'Inbox de l'utilisateur
-- contient bien la nouvelle mise à jour de l'idée
( IdeeService::getInbox(self.utilisateur)->includes(self.idee)

-- on a renvoyé un message de validation
( self.callback.getMethodCalled() == "onsuccess")))))

```

Suppression d'une idée

Description textuelle

Supprime une idée devant avoir été créée au préalable sur le serveur.

L'idée supprimée doit avoir été mise à la poubelle (via la méthode UpdateIdée).

La résolution des conflits liés à cette mise à jour dépend du mode de mise à jour (UpdateMode) choisi.

Signature de la méthode dans l'interface externe du serveur

```
public final void supprimerIdée(final Idée idée, final String identification, final
  CallBack<String> callback);
```

Le callback renvoie un message correspondant à la validation ou au refus de la demande de suppression de cette idée (si l'idée que le client veut supprimer n'a pas d'identifiant serveur correspondant à une idée possédée par l'utilisateur par exemple).

Pre et post-conditions OCL

```

context CommandeSupprimerIdée :: execute()
pre
  -- L'idée doit avoir été créée sur le serveur
  -- Il ne s'agit pas proprement parler d'une pre-condition,
  -- puisque notre serveur levera une exception
  self.idée.getIdServeur() <> 0

  -- Cette méthode doit être appelée par un utilisateur identifié
  -- au serveur et étant le modérateur (créateur) de cette idée.
  self.utilisateur <> null
  IdeeService::getInbox(self.utilisateur)@pre->one(Idée i |

```

```

    i.getIdServeur() == self.idee.getIdServeur())))
post
    -- CAS 1 : Erreur dans l'appel
    -- Si l'idée a été créée uniquement localement, c'est à dire :
    -- Qu'aucun identifiant serveur ne lui a été attribué
    ( ( self.idee@pre.getIdServeur() == 0

        -- Ou que l'idée n'appartient pas à l'utilisateur
    || IdeeService::getInbox(self.utilisateur)@pre->forAll(Idee i |
        i.getIdServeur() <> self.idee.getIdServeur()))

        -- Ou que l'idée n'a pas été mise à la corbeille sur le serveur
    || IdeeService::getInbox(self.utilisateur)@pre->exists(Idee i |
        (i.getIdServeur() == self.idee.getIdServeur() |
        not(i.estDansPoubelle())))
    )

        -- alors le mécanisme de callback levera une exception de type Invalid
        -- Call.
        (self.callback.getMethodCalled() == "onfailure")
        (self.callback.getExceptionType() == "InvalidCallException") )

        -- Dans tous les autres cas :
        -- L'idée avait été créée sur le serveur
    || ( ( self.idee@pre.getIdServeur() <> 0

        -- L'idée designait bien à l'utilisateur
        -- l'opérateur OCL "one" impose qu'un seul des éléments de la
        -- collection satisfait la condition (équivalent à un "exists" renforcé)
        IdeeService::getInbox(self.utilisateur)@pre->one(Idee i |
            i.getIdServeur() == self.idee.getIdServeur())

        -- De plus, cette idée avait bien été mise à la corbeille sur le serveur
            i.estDansPoubelle()
    )

        -- CAS 2 : l'appel est correct mais un conflit a été détecté
        -- et l'utilisateur est en mode "WARNING" => on renvoie ce conflit
    ( ( self.updateMode == "WARNING" )

        -- le fait qu'il existe sur le serveur une version plus récente de
        -- l'idée que l'utilisateur veut modifier constitue un conflit
    ( self.idee@pre.getDateDeDerniereModification() <
        IdeeService::getInbox(self.utilisateur).getIdee(
            self.idee@pre.getIdServeur()).getDateDeDerniereModification() )

        -- le mécanisme de callback levera alors une exception de type Warning
    ( ( self.callback.getMethodCalled() == "onfailure")
    (self.callback.getExceptionType() == "WarningException") ) )

        -- CAS 3 : l'appel est correct, et :
        -- soit l'utilisateur est en mode "FORCE" et il est possible qu'un
        -- conflit ait eu lieu

```

```

|| ( ( self.updateMode == "FORCE")
-- soit l'utilisateur est en mode "WARNING" est aucun conflit ne doit
-- avoir eu lieu
|| ( ( self.updateMode == "WARNING") ( self.idee@pre.
    getDateDeDerniereModification()
>= Inbox.getIdée( self.idee@pre.getIdServeur() ).
    getDateDeDerniereModification() ) )
-- la suppression de l'idée a fonctionné, c'est à dire :
-- que l'on a retiré l'idée de l'Inbox de l'utilisateur
-- contient bien la nouvelle mise à jour de l'idée
( IdeeService::getInbox( self.utilisateur )->excludes( self.idee )
-- on a retourné un message de validation
( self.callback.getMethodCalled() == "onsuccess" )))))

```

3.4.2 Créer une Tâche

Description textuelle

Ajoute une tâche créée localement à la base de données du serveur.

L'utilisateur ayant créé cette tâche sera considéré comme le modérateur de cette tâche.

Signature de la méthode dans l'interface externe du serveur

```
public final void creerTache( final Idee idee, final Tache tache, final String
    identification, final CallBack<Tache> callback );
```

Pré et post-conditions

Pré-conditions similaires à la création d'une idée.

3.4.3 Mettre à jour une Tâche

Description textuelle

Met à jour une tâche devant avoir été créée au préalable sur le serveur.

La résolution des conflits liés à cette mise à jour dépend du mode de mise à jour (UpdateMode) choisi. On permettra, dans le cas d'une tâche déléguée, au participant de la tâche de modifier son statut ("En cours" ou "Terminée").

Signature de la méthode dans l'interface externe du serveur

```
public final void updateTache( final Tache tache, final UpdateMode mode, final String
    identification, final CallBack<Tache> callback );
```

Pré et post-conditions

Pré-conditions similaires à la modification d'une idée.

On permettra cependant, dans le cas d'une tâche déléguée, que le participant de la tâche modifie son statut ("En cours" ou "Terminée").

3.4.4 Supprimer Tâche

Description textuelle

Supprime une tâche devant avoir été créée au préalable sur le serveur.

La tâche supprimée doit avoir été mise à la poubelle (via la méthode UpdateTache).

La résolution des conflits liés à cette mise à jour dépend du mode de mise à jour (UpdateMode) choisi.

Signature de la méthode dans l'interface externe du serveur

```
public final void supprimerTache( final Tache tache, final String identification, final
    CallBack<String> callback );
```

Pré et post-conditions

Pré-conditions similaires à la suppression d'une idée.

Seul le modérateur de la tâche peut supprimer une tâche, son participant pouvant uniquement mettre à jour son statut.

3.4.5 Créer un projet

Description textuelle

Ajoute un projet créé localement à la base de données du serveur.

L'utilisateur ayant créé ce projet sera considéré comme le **modérateur** du projet.

Lui seul pourra ajouter des tâches ou des participants à ce projet, ces participants ne pourront que changer le statut des tâches auxquelles ils participent.

Signature de la méthode dans l'interface externe du serveur

```
public final void creerProjet(final Idee idee, final Projet projet, final String identification, final CallBack<Projet> callback);
```

Pré et post-conditions

Pré-conditions similaires à la création d'une idée ou d'une tâche. De plus, on impose (logiquement) que toutes les tâches et participants associés à ce projet ait été au préalables créées sur le serveur.

3.4.6 Mettre à jour un projet sur le serveur

Description textuelle

Met à jour un projet devant avoir été créée au préalable sur le serveur.

La résolution des conflits liés à cette mise à jour dépend du mode de mise à jour (UpdateMode) choisi. Cette mise à jour comprend l'ajout ou la suppression de participants ou de tâches au projet.

Signature de la méthode dans l'interface externe du serveur

```
public final void updateProjet(final Projet projet, final UpdateMode mode, final String identification, final CallBack<Projet> callback);
```

Pré et post-conditions

Pré-conditions similaires à la mise à jour d'une tâche.

Plusieurs mécanismes de résolution de conflits seront à mettre en place.

Seul le modérateur du projet peut le mettre à jour.

3.4.7 Supprimer un projet

Description textuelle

Supprime un projet devant avoir été au préalable créé sur le serveur.

Le projet supprimé doit avoir été mis à la poubelle (via la méthode UpdateProjet).

La résolution des conflits liés à cette suppression dépend du mode de mise à jour (UpdateMode) choisi.

Signature de la méthode dans l'interface externe du serveur

```
public final void supprimerProjet(final Projet projet, final String identification, final CallBack<String> callback);
```

Pré et post-conditions

Pré-conditions similaires à la suppression d'une tâche.

Seul le modérateur du projet peut le supprimer.

3.4.8 Récupération des données

Description textuelle

Récupère l'ensemble des idées présentes dans l'Inbox, pour un utilisateur donné.

Il est possible de passer une date en paramètre, afin de récupérer uniquement les idées ajoutées dans l'Inbox après cette date.

Signature des méthodes dans l'interface externe du serveur

```
public final void downloadInbox(final Date date, final String identification, final  
    CallBack<List<Idee>> callback);  
public final void downloadInbox(final String identification, final CallBack<List<Idee>>  
    callback);
```

Description textuelle

Récupère l'ensemble des idées, tâches et projets ayant été mis à la poubelle, pour un utilisateur donné. Il est possible de passer une date en paramètre, afin de récupérer uniquement les données mises à la poubelle après cette date.

Signature des méthodes dans l'interface externe du serveur

```
public final void downloadPoubelle(final Date date, final String identification, final  
    CallBack<List<Object>> callback);  
public final void downloadPoubelle(final String identification, final CallBack<List<  
    Object>> callback);
```

Description textuelle

Récupère l'ensemble des tâches et projets présents ayant été archivés.

Il est possible de passer une date en paramètre, afin de récupérer uniquement les données archivées après cette date.

Signature des méthodes dans l'interface externe du serveur

```
public final void downloadArchive(final Date date, final String identification, final  
    CallBack<List<Object>> callback);  
public final void downloadArchive(final String identification, final CallBack<List<  
    Object>> callback);
```

Description textuelle

Récupère l'ensemble des prochaines tâches à réaliser, pour un utilisateur donné.

Il est possible de passer une date en paramètre, afin de récupérer uniquement les tâches ajoutées à la liste des prochaines tâches après cette date.

Signature des méthodes dans l'interface externe du serveur

```
public final void downloadProchainesTaches(final Date date, final String identification,  
    final CallBack<List<Tache>> callback);  
public final void downloadProchainesTaches(final String identification, final CallBack<  
    List<Tache>> callback);
```

Description textuelle

Récupère l'ensemble des prochaines tâches à réaliser en fonction du contexte passé en paramètre, pour un utilisateur donné.

Il est possible de passer une date en paramètre, afin de récupérer uniquement les tâches ajoutées à la liste des prochaines tâches après cette date.

Signature des méthodes dans l'interface externe du serveur

```
public final void downloadProchainesTaches(final Contexte contexte, final Date date,  
    final String identification, final CallBack<List<Tache>> callback);  
public final void downloadProchainesTaches(final Contexte contexte, final String  
    identification, final CallBack<List<Tache>> callback);
```

Description textuelle

Récupère l'ensemble des tâches présentes au sein du calendrier, pour un utilisateur donné.

Il est possible de passer une date en paramètre, afin de récupérer uniquement les tâches ajoutées au calendrier après cette date.

Signature des méthodes dans l'interface externe du serveur

```
public final void downloadCalendrier(final Date date, final String identification, final  
    CallBack<List<Tache>> callback);
```

```
public final void downloadCalendrier(final String identification, final CallBack<List<Tache>> callback);
```

Description textuelle

Récupère l'ensemble des tâches associées au tag passé en paramètre, pour un utilisateur donné. Il est possible de passer une date en paramètre, afin de récupérer uniquement les tâches auxquelles on a ajouté le tag passé en paramètre après cette date.

Signature des méthodes dans l'interface externe du serveur

```
public final void downloadTacheParTag(final Date date, final Tag tag, final String identification, final CallBack<List<Tache>> callback);
public final void downloadTacheParTag(final Tag tag, final String identification, final CallBack<List<Tache>> callback);
```

Description textuelle

Récupère l'ensemble des idées, tâches ou projets, pour un utilisateur donné.

Signature des méthodes dans l'interface externe du serveur

```
public final void downloadAllIdee(final String identification, final CallBack<List<Idee>> callback);
public final void downloadAllTache(final String identification, final CallBack<List<Tache>> callback);
public final void downloadAllProjet(final String identification, final CallBack<List<Projet>> callback);
```

3.4.9 Gestion des comptes

Description textuelle

Crée un compte utilisateur afin de pouvoir se connecter au serveur et ainsi permettre à l'utilisateur de réaliser différentes actions. L'utilisateur fournit son login, son mot de passe et son pseudo afin de créer son compte. La méthode retourne au sein du Callback le jeton d'identification de l'utilisateur.

Supprime un compte utilisateur. L'utilisateur fournit son login et mot de passe associé ainsi que son identification afin de s'assurer de supprimer le bon compte utilisateur.

Signature des méthodes dans l'interface externe du serveur

```
public final void creerCompte(final String username, final String password, final String pseudo, final CallBack<String> callback);

public final void supprimerCompte(final String username, final String password, final String identification, final CallBack<String> callback);
```

Description textuelle

Modifie le pseudo de l'utilisateur en fournissant simplement son nouveau pseudo ainsi que le jeton permettant de l'identifier.

Modifie le mot de passe de l'utilisateur en fournissant, en plus de son nouveau mot de passe et de son jeton associé, son ancien mot de passe pour des raisons de sécurité.

Signature des méthodes dans l'interface externe du serveur

```
public final void modifierPseudo(final String pseudo, final String identification, final CallBack<String> callback);
public final void modifierMotDePasse(final String oldPassword, final String newPassword, final String identification, final CallBack<String> callback);
```

Description textuelle

La connexion crée un jeton sous forme de chaîne de caractères permettant d'identifier l'utilisateur à partir de son login et de son mot de passe.

La déconnexion déconnecte l'utilisateur à partir de son jeton d'identification et supprime ce dernier.

Signature des méthodes dans l'interface externe du serveur

```
public final void connect(final String username, final String password, final CallBack<String> callback);
public final void disconnect(final String identification, final CallBack<String> callback);
```

Description textuelle

Récupère l'ensemble des utilisateurs enregistrés sur le serveur.

Signature des méthodes dans l'interface externe du serveur

```
public final void downloadListeParticipant(final String identification, final CallBack<List<Participant>> callback);
```

Description textuelle

Récupère les différents logs générés par le serveur concernant les différentes actions de l'utilisateur. L'utilisateur peut récupérer ses logs pour une date donnée.

Signature des méthodes dans l'interface externe du serveur

```
public final void downloadLog(final String identification, final CallBack<List<String>> callback);
```

3.4.10 Fonctionnalités proposées à l'Administrateur

Seul l'administrateur peut user des méthodes suivantes.

```
public final void downloadLogAdmin(final String identification, final CallBack<List<String>> callback);
```

Récupère les différents logs générés par le serveur concernant l'ensemble des actions de tous les utilisateurs.

```
public final void downloadIdee(final String username, final String identification, final CallBack<List<Idee>> callback);
```

Récupère toutes les idées d'un utilisateur. L'administrateur fournit son jeton afin de s'identifier.

```
public final void downloadTaches(final String username, final String identification, final CallBack<List<Tache>> callback);
```

Récupère toutes les tâches d'un utilisateur. L'administrateur fournit son jeton afin de s'identifier.

```
public final void downloadTaches(final String username, final String identification, final CallBack<List<Tache>> callback);
```

Récupère toutes les projets d'un utilisateur. L'administrateur fournit son jeton afin de s'identifier.

```
public final void downloadProjets(final String username, final String identification, final CallBack<List<Projet>> callback);
```

Récupère toutes les logs générés par le serveur concernant l'ensemble des actions d'un utilisateur donné. L'administrateur fournit son jeton afin de s'identifier.

3.5 Spécifications des composants du serveur GTD

Présentons premièrement les différents composants que nous avons jugé nécessaires pour effectuer un traitement valide et modulaire des appels de méthodes distantes provenant des applications clientes.

Ces composants et les interactions qui les unissent sont décrits dans le diagramme de composants suivant :

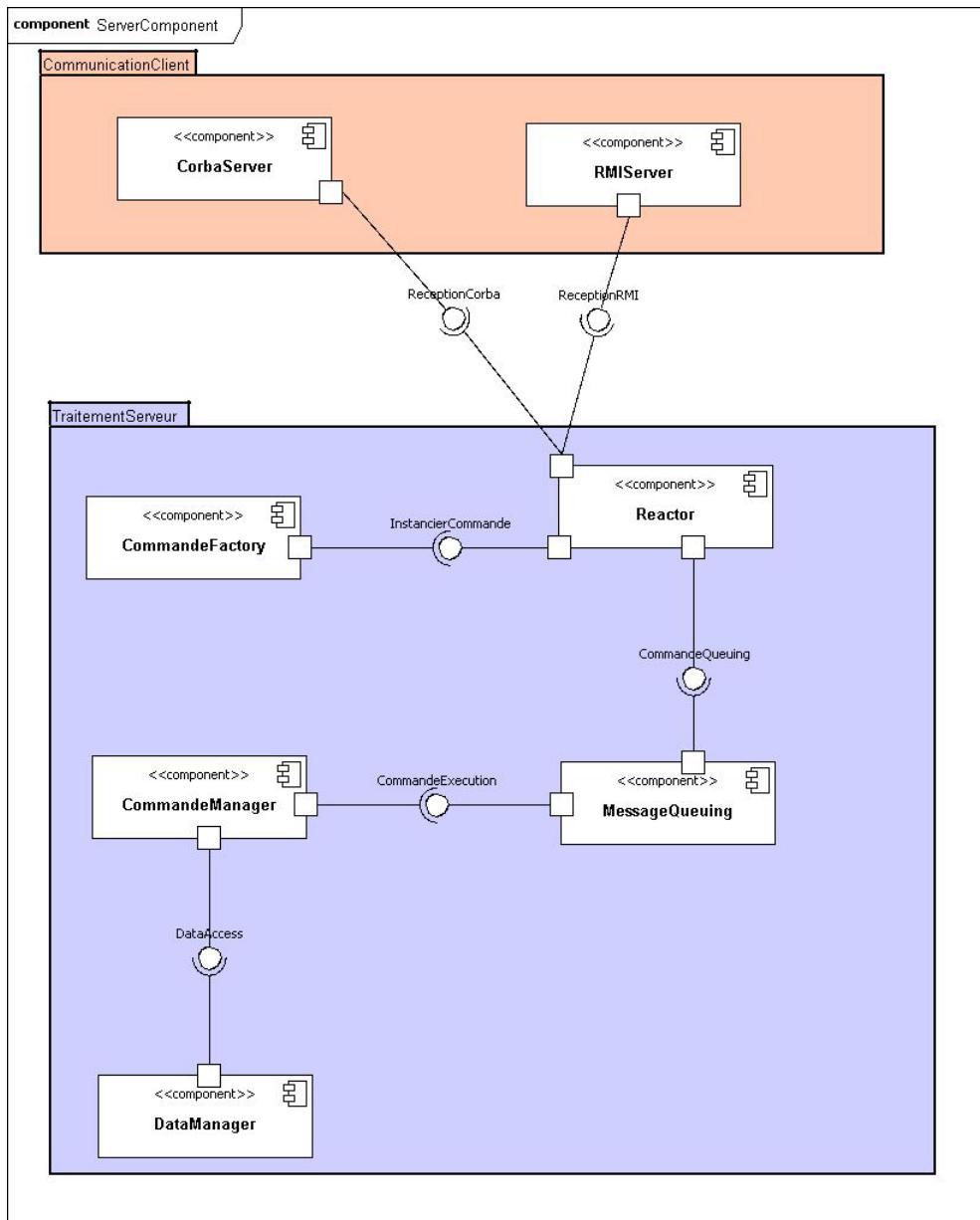


FIGURE 3.2 – Diagramme de Composants du serveur GTD

3.5.1 Les composants CorbaServer et RMIServer

Définition du composant

Le but de ces composants est de recevoir et décoder les appels de méthodes distantes destinés au serveur GTD.

Ils seront tous deux inscrits à un annuaire, et ce sont eux qui seront effectivement récupérés par les applications clientes.

Suite à un appel de méthode sur l'un de ces deux composants, on appellera la méthode **dispatch()** du composant Reactor, en renseignant le "type" de la méthode appelée et en transmettant ses paramètres.

Signature des méthodes

Les méthodes fournies seront strictement les mêmes que celles décrites plus haut, à ceci près que ces deux protocoles de communication imposent qu'elles puissent renvoyer une RemoteException.

3.5.2 Le composant Reactor

Définition du composant

Inspiré par le Design Pattern du même nom, ce composant est un dispatcheur d'évenements ayant pour but de permettre la gestion d'actions concurrentes.

Conformément aux spécifications des besoins définies dans le livrable précédent, c'est ce composant qui va permettre de satisfaire en partie les besoins en terme de :

1. **Disponibilité** : le serveur sera toujours disponible pour traiter une demande, les demandes reçues précédemment étant traitées dans un autre thread
2. **Efficacité** : le serveur maximise la capacité de traitement
3. **Application Multi-Utilisateur** : c'est ce composant qui permet d'interrompre le flot d'exécution des appels entrants, ce qui permet un traitement asynchrone des actions et par conséquent rend le serveur multi-utilisateur.

On remplit également d'autres critères comme la simplicité et l'adaptabilité.

Pour chaque appel de méthode reçu par le composant CorbaServer ou RMIServer, ce composant va premièrement vérifier (grâce à une entité Acceptor) que la méthode appelée provient d'un utilisateur identifié, et effectuer différentes vérifications relatives à la sécurité.

Si l'action est valide, le Reactor va créer un Thread destiné à exécuter cette action. Ce Thread va ensuite réifier l'action souhaitée sous forme de Commande, en faisant appel au composant CommandeFactory, puis transmettre cette commande au MessageQueuing.

Signature des méthodes

```
dispatch(OperationEnum typeAction, Object parametresAction, String identification, Callback<Object> callback)
```

3.5.3 Le composant CommandeFactory

Définition du composant

Ce composant a pour but de transformer l'appel de méthode reçu en une Commande (Design Pattern Commande). Notre décision de définir CommandeFactory en tant que composant (et non en tant que classe) peut se justifier par son importance : c'est ce composant qui va effectuer toutes les vérifications liées à l'appel distant (vérification des paramètres...).

Appelé par le réactor, qui lui fournit le type d'action que l'application distante souhaite réaliser, elle réifie l'action sous forme de Commande, et la renvoie au Thread créé par le Reactor.

Signature des méthodes

```
newCommand(OperationEnum typeAction, Object parametresAction, String identification, Callback<Object> callback)
```

3.5.4 Le composant MessageQueuing

Définition du composant

Ce composant a pour but de séquentialiser l'exécution des commandes, afin d'éviter des problèmes de compétitions inter-Threads, de maximiser l'efficacité du serveur et de gérer l'accès aux données de

manière non concurrente.

Concrètement, chaque Thread créé par le Reactor, au moment où il souhaitera exécuter sa Commande, devra l'insérer dans le composant MessageQueueing. Ce composant va ensuite séquentialiser l'exécution des commandes, via un appel au CommandeManager.

Signature des méthodes

insert(Commande commande)

3.5.5 Le composant CommandeManager

Définition du composant

Ce composant a pour but d'exécuter les commandes fournies par le composant MessageQueueing. En cas d'erreur lors de l'exécution ou de conflits dans l'accès aux données, ce composant appellera l'action onFailure du callback. Toutes les commandes devront accéder aux données du serveur, et donc faire appel (de manière symbolique) au DataManager.

Signature des méthodes

invoker(Commande commande)

3.5.6 Le composant DataManager

Définition du composant

Ce composant a pour but de stocker les données GTD du serveur. Concrètement, il correspond à une base de données abstraite, que nous ne pensons pas implémenter. En effet, nous préférerions utiliser une API de persistance, comme Hibernate.

3.5.7 Le composant IHMAdministration

Définition du composant

Ce composant a pour but de permettre à l'administrateur d'effectuer différents paramétrages du serveur et d'obtenir des données statistiques relatives aux actions que le serveur a exécuté.

La spécification complète de ce composant sera rendue séparément, dans le cadre du module IHM.

3.5.8 Le composant RMIAdministration

Définition du composant

Ce composant a pour but de transmettre les actions déclenchées dans l'IHM au serveur. Il utilisera pour cela le protocole de communication RMI.

3.6 Interaction inter-composants

3.6.1 Diagramme d'activité du Serveur GTD

Ce diagramme a pour but de représenter le comportement général du serveur, en précisant pour chaque action le composant qui l'effectue.

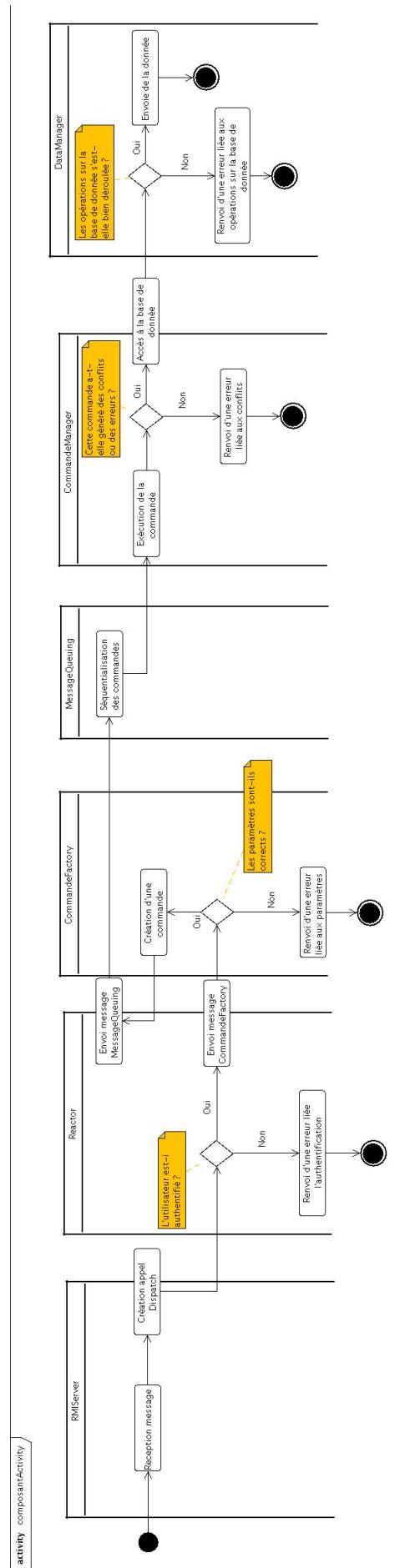


FIGURE 3.3 – Diagramme d'activité

3.6.2 Diagrammes de séquences du Serveur GTD

Traitement d'une requête correcte et ne générant aucun conflit

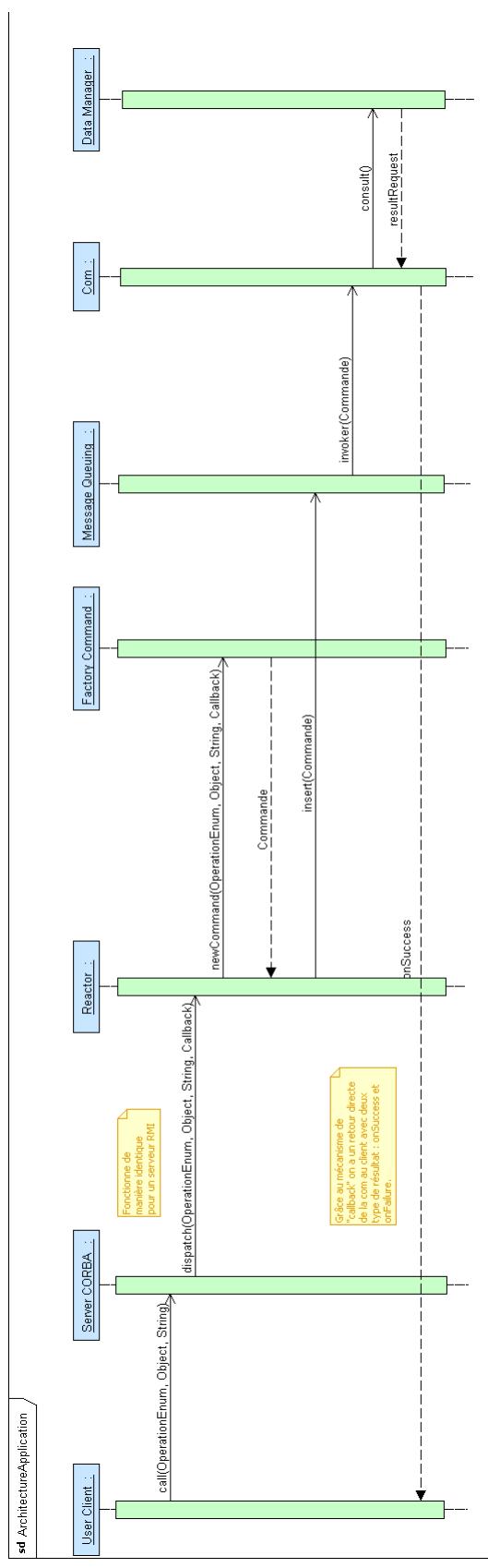


FIGURE 3.4 – Architecture du logiciel

Traitement d'une requête provenant d'un utilisateur non authentifié

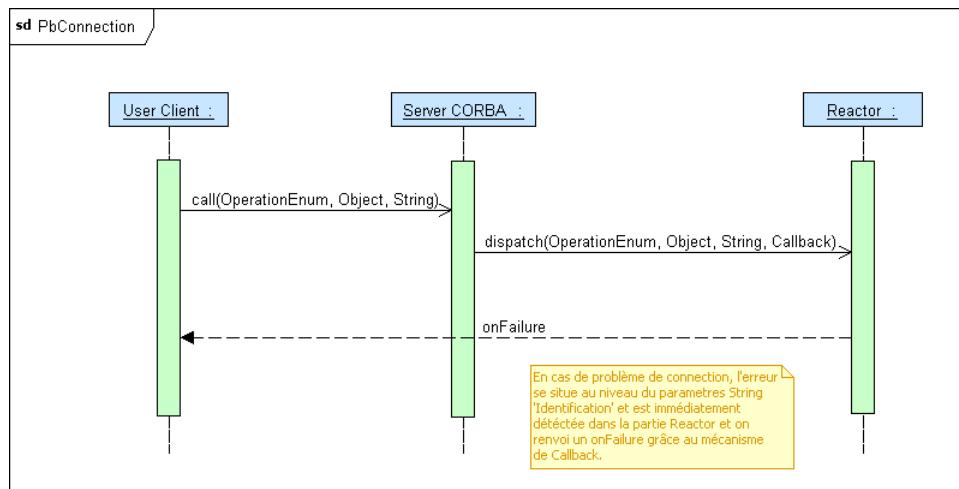


FIGURE 3.5 – Diagramme de sequence de la connexion

Traitement d'une requête dont les paramètres sont incorrects

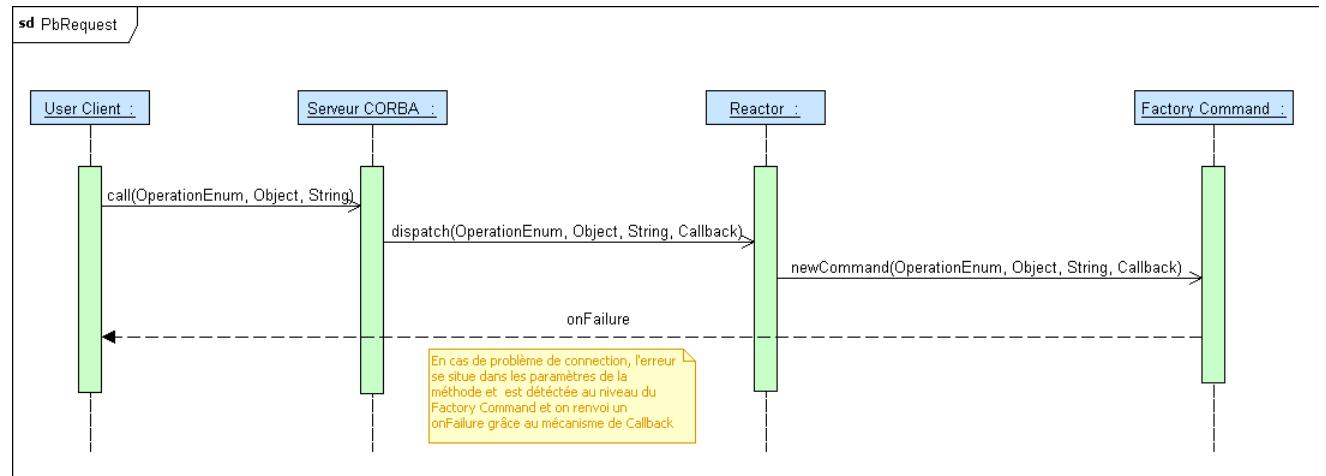


FIGURE 3.6 – Diagramme de sequence des requêtes

Chapitre 4

Livrable 4 : Architecture

4.1 Architecture Physique du serveur GTD

L'architecture physique du serveur GTD étant basée sur les différents composants présentés dans le livrable précédent, nous ne rentrerons pas dans les détails.

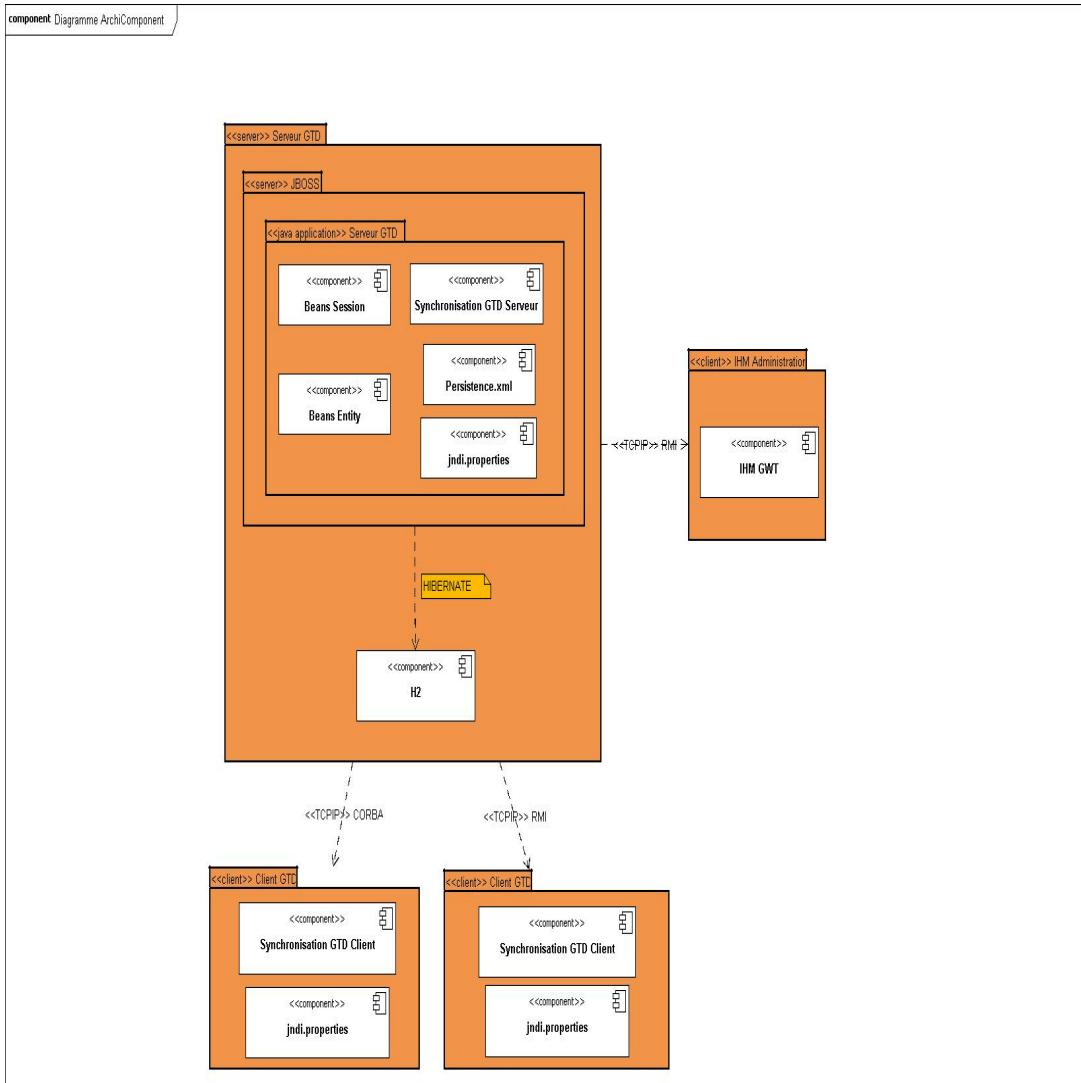


FIGURE 4.1 – Architecture physique du serveur GTD

Nos beans sessions et entités (voir livrable 5), ainsi que tous les fichiers nécessaires à leur déploiement correct sur JBoss (persistance.xml, jndi.properties...) sont regroupés dans une même archive JAR.

Cette archive sera déployé sous JBoss, et fera appel à une base H2 extérieure à JBoss.

Les applications clientes et l'IHM Administrateur pourront communiquer avec le serveur via des protocoles de communication distante (CORBA ou RMI).

4.2 Schéma de la base de données GTD

La base de données sera entièrement générée par Hibernate. A ce stade de développement, nous ne disposons donc d'aucune information sur le schéma concret de notre base de données : nous savons simplement qu'elle permettra de stocker les entités GTD. C'est tout l'intérêt d'utiliser des technologies de persistance haut-niveau : les développeurs n'auront pas à se soucier du schéma de la base de données, manipulant les objets et les classes plutôt que les tuples de la base de données.

Si une évolution dans l'utilisation du serveur entraîne un manque de performance, les développeurs

pourront toutefois accéder à la base de données via une interface Web et proposer différentes solutions (créations d'index, de vues matérialisées...) afin d'optimiser l'accès aux données.

4.3 Règles de traduction de UML en code source

La principale règle suivie a été le bon sens. Globalement, nous nous sommes fixés un ensemble de conventions de nommage et de règles d'écritures :

1. les interfaces sont préfixées par un "I" (IProjet, ITache...)
2. les classes abstraites postfixées par "Abstract" (AbstractProjet, AbstractTache)
3. les classes concrètes ne sont ni préfixées ni postfixées (Projet, Tache)
4. les services seront postfixés par "Service" (ProjetService, TacheService)
5. les interfaces distantes seront postfixées par "Remote" (IProjetServiceRemote)
6. les commandes seront préfixées par "Commande" (CommandeCreerProjet)
7. on traduira une cardinalité multiple (exemple : projets : String[6]) par une liste d'éléments¹.
8. les contraintes OCL ne seront pas traduites sous forme de pre et post-conditions. Nous avons premièrement pensé utiliser JML (Java Modeling Language), un langage sur la programmation par contrat (à l'instar d'Eiffel) qui nous aurait permis de traduire les contraintes OCL. Cependant, nous avons abandonné cette idée pour deux raisons : premièrement à cause de la lourdeur d'une telle tâche (de nombreuses contraintes OCL sont triviales et pourraient être exprimées moins formellement), et surtout parce qu'un comportement par contrat ne nous a pas semblé adapté au serveur : si l'appel à une méthode ne respecte pas la pre-condition, une exception sera levée et le serveur ne pourra pas lancer la méthode onFailure du callback (comportement spécifié dans le livrable 3).
9. globalement, on utilisera des conventions CheckStyle, respectées par l'ensemble des membres du projet, pour fixer une et une seule façon de coder (dérivabilité des attributs et paramètres, conventions de nommage...).

Puisque nous devons obtenir un code fonctionnel le plus rapidement possible, nous ne rédigerais la conception détaillée qu'après l'implémentation du serveur.

1. Comme nous le verrons dans l'implémentation, ce choix s'est révélé mauvais : nous aurions dû utiliser des Sets, aussi bien d'un point de vue sémantique que technique

Chapitre 5

Livrable 5 : Conception détaillée

Puisque nous étions en charge du serveur et que toutes les applications clientes étaient en attente d'une première version pour commencer à coder, ce livrable n'a été rédigé qu'après l'implémentation de la plupart des fonctionnalités du serveur. Pour les mêmes raisons, nous n'avons pu suivre une approche par modèle complète, incompatible avec les délais souhaités par les applications clientes. Le code de notre application n'a donc pas été généré à partir d'un modèle.

Cela dit, nous avons tout de même effectué la conception détaillée de notre application avant de l'implémenter, sans pour autant tout formaliser ou réaliser tous les diagrammes. Nous nous permettrons donc dans ce chapitre d'évoquer notre code, puisqu'il est toujours intéressant d'effectuer un parrallèle entre conception et implémentation.

5.1 Principes suivis pour la conception/le développement

5.1.1 Conception EJB/JEE : un modèle en couche structurant les applications

Dans le cadre du modèle de programmation EJB/JEE, on identifie 4 couches distinctes. Nous avons utilisé ce modèle en couche pour structurer notre serveur GTD :

1. La couche des entités du domaine (Projet, Tache...),
2. La couche des services fondamentaux du domaine (ProjetService, TacheService...),
3. La couche des contrôleurs de cas d'utilisation, qui correspond au pilotage des rendus de services (CommandeCreerProjet, CommandeLogin...),
4. La couche des contrôleurs d'acteurs, aussi appelée « boundaries » (Interface Homme Machine : applications clientes ou interface Administrateur).

L'utilisation d'un tel modèle nous a permis de séparer clairement les problématiques, et de de répartir le travail de manière efficace (au sein de notre groupe tout du moins).

5.1.2 Bonnes pratiques liées à l'implémentation : conventions, style et tests

Lors de l'implémentation de ce projet, nous nous sommes fixés des règles que tous les membres du groupe devaient suivre, afin d'obtenir un code le plus cohérent et lisible possible. Par exemple, des conventions de nommage ont été fixées (toutes les interfaces doivent être préfixés d'un "I", les beans sessions postfixés par "Service"...).

L'utilisation de JavaDoc et de CheckStyle facilitent également une bonne lecture du code : via les conventions CheckStyle, notre code est uniforme, lisible et correctement commenté. La synchronisation entre les membres du groupes a été assurée via un gestionnaire de versions (un dépôt SVN sous GoogleCode), accessible à tous les dévelopeurs, y compris ceux des applications clientes, qui ont pu suivre l'évolution du serveur et nous faire part de leurs remarques. La documentation complète du serveur GTD

sera rendue avec le code.

Nous avions bien évidemment prévu de réaliser des tests unitaires pour chacune de nos entités et services. Nous nous sommes renseignés pour mettre en place JBoss TestSuite, une suite de tests destinés à être exécutés sur nos beans déployés sous JBoss. Cependant, le temps nous manquant, nous avons préféré favoriser le développement des fonctionnalités nécessaires aux clients plutôt que les tests. Dans le cadre d'un projet d'entreprise, nous aurions déployé des tests unitaires pour chaque bean entité et session.

Bien que nous n'ayons pas implémenté de tests unitaires pour chaque bean, nous avons réalisé plusieurs tests en écrivant du code client (localisation puis interrogation des services) afin de vérifier le bon fonctionnement de nos beans. De plus, des tests unitaires JUnit ont été réalisés pour d'autres entités du système (plus facile à tester unitairement car moins de cas sont à prendre en compte). Des outils d'analyse de code ont également été utilisées, comme Checkstyle et EclEmma pour, entre autre, calculer la couverture de code et la complexité cyclomatique.

5.2 Données partagées entre client et serveur : logique suivie

Avant de présenter en détail chacun de nos composants, revenons premièrement sur la logique suivie pour concevoir les données partagées entre client et serveur (i.e les données GTD : Projet, Tache, Participant...).

Dans le cadre de ce projet, nous devions proposer un ensemble d'interfaces représentant les données GTD, afin de fixer une norme commune à toutes les applications clientes, notamment pour garantir une bonne **inter-opérabilité** (un des critères de qualité majeur de notre application).

Au vu de cette problématique particulière, nos beans entités (représentant les entités du domaine, à savoir les Idees, Taches, Projets...) devront implémenter ces interfaces. Amenés à écrire du code utilisable par les applications clientes au sein de ces entités (déclaration commentée des attributs, getters et setters...), nous avons séparé chaque bean entité en deux parties :

1. la partie commune entre clients et serveur ;
2. la partie persistance (annotations hibernate...), uniquement assurée par le serveur.

Pour chaque entité du domaine (par exemple Projet), on proposera une interface (IProjet) présentant l'ensemble des getters, setters et méthodes indispensables à la bonne représentation de cette entité. Cette base minimale a été déterminée à partir de la spécification de la méthode GTD, ou du moins de l'interprétation que nous en avons faite. Toutes les entités envoyées par le client devront implémenter ces interfaces.

Chaque entité se verra également munie d'une classe abstraite (AbstractProjet), proposant un ensemble de fonctionnalités que nous pensons utiles aux clients (constructeur par recopie, setters, méthodes utiles...). Cependant, les clients pourront bien évidemment concevoir leur Projet sans se baser sur cette classe abstraite.

Il est à noter que, puisque ces entités seront amenées à circuler sur le réseau (via des protocoles de communication distante comme RMI ou CORBA), elles devront être sérialisables (et donc implémenter l'interface Serializable en Java).

Nous avons effectué quelques modifications par rapport au modèle présenté, principalement des corrections d'oubli ou des modifications demandées par les clients (ajout d'une liste de tâche antérieures devant s'effectuer avant une certaine tâche...).

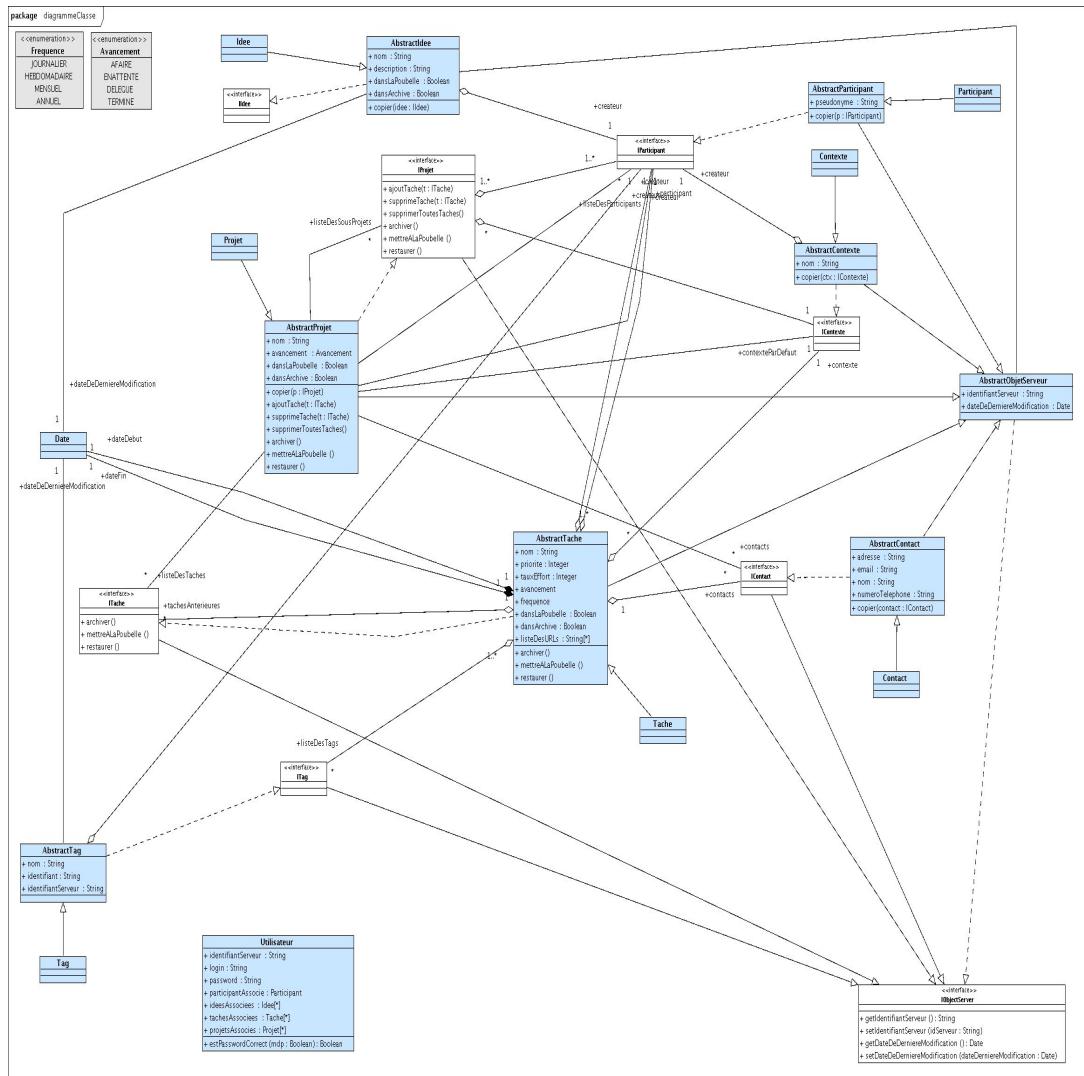


FIGURE 5.1 – Diagramme de sequence des requetes

5.3 Description détaillée de chaque composant

5.3.1 Composant DataManager : les entités du système et la logique métier

Le rôle que nous avions donné à ce composant était de "stocker les données GTD" (livrable 3). Lors de l'implémentation, nous nous sommes aperçus qu'il ne doit pas être une simple base de données : en effet, ce composant doit également proposer toutes les fonctionnalités assurant une gestion complète des données (CRUD, recherche avancée...).

Pour une description complète de la conception et de l'implémentation de ce composant, veuillez vous référer au livrable traitant de la persistance des données (en annexe).

On citera cependant quelques points clés vis-a-vis des objectifs que nous nous étions fixés.

Gestion des identifiants pour les entités du système

La **Sécurité** de notre application est en partie assurée par le chiffrage des identifiants serveurs attribués à chaque entité. En effet, ces identifiants doivent être peu évident à décoder, pour des raisons de

sécurité.

Prenons un exemple concret : si nous avions utilisé des identifiants séquentiels, il aurait été très facile pour un client d'accéder aux données d'un client B. Par exemple, si A possède 2 tâches dont les identifiants valent 3 et 5, il lui est facile de supposer qu'il existe une tâche d'identifiant 4 et d'essayer d'y accéder. Globalement, nous voulions que le client n'ait aucun moyen de deviner la structure de la base de données.

Nous avons donc utilisé une stratégie de génération d'identifiants un peu plus complexe, à savoir l'utilisation d'UUID (Universally Unique Identifier). Ces identifiants uniques sont codés sur 128 bits et sont produits en utilisant des composantes pseudo-aléatoires ainsi que les caractéristiques du serveur sur lequel il a été généré (numéro de disque dur, adresse MAC...). Il est facile d'indiquer à Hibernate une stratégie de génération d'identifiant différente de celle par défaut :

```
@Id  
@GeneratedValue(generator = "system-uuid")  
@GenericGenerator(name = "system-uuid", strategy = "uuid")  
public String getIdentifiantServeur() { }
```

Gestion de la cohérence des données

La cohérence des données stockées sur le serveur est bien évidemment primordiale. Étant amené à régler des problèmes de synchronisation clients/serveur, il nous a fallu spécifier un moyen de déterminer quelle version d'une entité choisir en cas de conflit (un client voulant par exemple remplacer une tâche sur le serveur).

Pour ce faire, notre serveur propose deux modes de mise à jour (UpdateMode), les applications clientes pouvant choisir librement le mode utilisé :

1. Mode "FORCE" : l'utilisateur est prioritaire par rapport aux données du serveur et peut les écraser à tout moment.
2. Mode "WARNING" : si la date de dernière modification de l'entité du client est supérieure à celle de la version du serveur, alors elle est correctement remplacée. Sinon, on utilise le callBack pour prévenir l'application cliente. Elle pourra alors choisir de forcer la mise à jour (en changeant l'updateMode) ou de se mettre à jour.

Dans nos spécifications, nous avions proposé un troisième type d'updateMode : "PASSIVE", dans lequel le serveur utiliserait une politique de gestion de conflits pour prendre lui-même une décision. Nous avions cependant précisé qu'un tel mode de mise à jour, bien qu'intéressant, représentait un grand temps de développement sans être fondamental. Il pourra cependant faire l'objet d'une prochaine mise à jour, et le code ayant été écrit afin de faciliter son incorporation.

5.3.2 Reactor

Il s'agit du seul point d'entrée du serveur. Par rapport à la définition effectuée dans le livrable 3, nous avons distingué deux types de reactor : un reactor chargé de recevoir les appels RMI (reactorRMI) et un reactor chargé de recevoir les appels Corba, ces deux types d'appel permettant aux applications clientes de communiquer avec le serveur GTD.

L'Acceptor : un contrôleur d'accès performant

Comme nous l'avions spécifié, les reactor possède un acceptor chargé de gérer l'identification des utilisateurs et de contrôler la validité de toutes les connexions. Lorsqu'un utilisateur souhaite se connecter (via la méthode "login" du reactor), on demande à l'acceptor de générer un jeton d'identification et on le retourne au client. Pour des raisons de sécurité, nous avons utilisé un mécanisme de chiffrage de ce jeton, afin d'empêcher une application "malveillante" de tenter de générer elle-même ses propres jetons. Le jeton (correspondant à une chaîne de caractères) est généré à partir de l'identifiant de l'utilisateur

souhaitant se logger. Cet identifiant correspond à un UUID, très difficile à générer soi-même, puisque prenant en compte de nombreux paramètres comme l'adresse MAC du serveur. Afin de complexifier le jeton, on applique l'algorithme de chiffrage SHA-512 (méthode "encode"). Le jeton ainsi généré est ajouté à la liste des jetons actifs, puis retourné à l'utilisateur.

Maintenant que l'utilisateur est authentifié, il peut faire appel aux différentes fonctionnalités proposées par le serveur, en passant systématiquement le jeton de connexion en paramètre. Avant de réaliser la fonctionnalité souhaitée, le reactor demandera à l'acceptor de vérifier si ce jeton correspond bien à un jeton actif, et donc une connexion valide (méthode "accept" renvoyant un booléen). Pour ce faire, il suffit de vérifier qu'il appartient bien à la liste des jetons actifs.

Lorsque l'utilisateur souhaite se déconnecter (méthode "disconnect" de l'acceptor), le réacteur fait appel à la méthode "retire" de l'acceptor, qui retire alors le jeton de la liste des jetons actifs.

Il est à noter qu'il est important que la liste des jetons actifs supporte des interrogations concurrentes, puisqu'elle doit pouvoir être interrogée simultanément par différentes instances de Reactor (JBOSS créant plusieurs instances de reactor puisqu'il s'agit d'un EJB Stateless). Il existe en Java des structures supportant une telle concurrence :

```
this.jetonActif = new ConcurrentHashMap<String, String>();
```

ReactorRMI

Il s'agit d'un bean session déployé sous JBOSS, afin de bénéficier du registre RMI interne à ce serveur d'application. Il est muni d'un Acceptor et d'une fabrique de commande, et implémente l'interface serveurRMI, listant l'ensemble des actions possibles pour une application cliente.

Lorsque ce reactor reçoit un appel de méthode, le reactor fait appel à l'acceptor pour déterminer si l'utilisateur est correctement authentifié. Si ce n'est pas le cas, il utilise le mécanisme de callback pour lever une exception (méthode "onFailure" : voir livrable relatif à la persistance - partie RMI/CallBack). On garantie ainsi le critère de **sécurité** que nous nous étions fixé : seul un utilisateur authentifié pourra intéragir avec notre serveur.

Dans le cas où l'utilisateur est correctement identifié, le reactor fait appel à la fabrique de commande pour générer la commande correspondant au type d'action souhaité par l'utilisateur. La commande ainsi générée est placée dans un nouveau Thread, permettant ainsi un fonctionnement asynchrone pour le serveur (le Thread "casse" la pile d'exécution). Si nous en avions eu le temps, nous aurions aimé concevoir un pool de Thread, afin de contrôler et d'optimiser le nombre de threads actifs sur le serveur : en effet, créer un Thread par appel reçu peut entraîner un ralentissement du serveur (trop de Threads actifs en même temps). En se basant sur le comportement du serveur d'application TomCat, par exemple, nous aurions pu créer un pool de 5 Threads, chaque commande étant mise en attente jusqu'à ce qu'un de ces threads puissent l'accueillir.

```
public void creerTag(final ITag tag, final String identification, final CallBack<ITag> callback) throws RemoteException {
    if (this.acceptor.accept(identification)) {
        final ThreadDeCommande tdc = new ThreadDeCommande(this.commandeFactory.getCommandeCreerTag(tag, callback));
        final Thread t = new Thread(tdc);
        t.start();
    } else {
        callback.onFailure(new Exception(this.identificationInvalide));
    }
}
```

ReactorCorba

Ce reactor implémente l'interface ServeurCorba, générée automatiquement à partir de la définition des interfaces IDL (voir rapport CORBA, en annexe). Le fonctionnement est globalement identique au reactor RMI, à une subtilité près : afin de ne pas contraindre trop fortement les clients au niveau des données GTD (Projet, Tache...), nous n'avons pas souhaité d'interdire l'utilisation de structures complexes (listes, ensembles...). Il nous a donc fallu définir des convertisseurs permettant de traduire les classes java générées par IDL (utilisant uniquement des énumérations et des tableaux) en classes conformes à notre modèle, et vice-et-versa.

On conserve strictement le même fonctionnement, excepté que la première chose à faire est de convertir les données passées en paramètres (respectant l'interface IDL) en données respectant notre spécification.

5.3.3 Le composant CommandeFactory

Ce composant regroupe la fabrique de commandes et l'ensemble des commandes du serveur GTD.

La fabrique de commandes

Il s'agit d'une simple implémentation du design-pattern factory : munie d'une méthode par type de commande, elle crée la commande correspondant au type d'action souhaité.

```
/**  
 * Cree la commande de suppression de tache.  
 * @param tache La tache a supprimer.  
 * @param callback Le callback.  
 * @return La commande de suppression de tache.  
 */  
public Commande getCommandeSupprimerTache( final ITache tache , final CallBack<String>  
callback ) {  
    return new CommandeSupprimerTache( tache , callback );  
}
```

Les Commandes : des contrôleurs de cas d'utilisation faisant appel aux services fondamentaux

Situées dans la couche des contrôleurs de cas d'utilisation du modèle de programmation JEE, les commandes ont pour but de faire appel à la logique métier (beans sessions TacheService, Utilisateur-Service...) tout en effectuant divers contrôles. Prenons l'exemple de la commande chargée d'assurer la création d'une idée.

On instancie une commande de ce type à chaque fois qu'un utilisateur fait appel à la méthode creeIdée du serveurRMI ou serveurCorba. Le reacteur fait alors appel à la fabrique de commandes pour générer la commande correspondant au type d'action souhaité.

Nous avons muni chaque commande d'attributs dans lesquels elle pourra stocker les différents paramètres d'appels aux services fondamentaux qu'elle est chargée d'utiliser. Par exemple, pour la commande CommandeCreerIdée, on stocke l'identifiant de l'utilisateur souhaitant créer cette idée, l'idée à créer (devant implémenter l'interface commune IIdee) ainsi que le CallBack du client (afin de pouvoir lui répondre lorsque la commande sera exécutée).

On **localisera** également les services auxquels la commande devra faire appel lors de son exécution, en stockant ces services en tant qu'attributs. Voici un exemple de construction d'un commande :

```

public CommandeCreerIdee(final String i, final IIdee id, final CallBack<IIdee> c) {
    super();
    this.identification = i;
    this.idee = id;
    this.callback = c;

    final Properties env = new Properties();
    env.setProperty("java.naming.factory.initial", "org.jnp.interfaces.NamingContextFactory"
    );
    env.setProperty("java.naming.provider.url", "localhost:1099");
    env.setProperty("java.naming.factory.url.pkgs", "org.jboss.naming");

    try {
        final Context context = new InitialContext(env);
        this.ideeServiceRemote = (IIdeeServiceRemote) context.lookup("IdeeService/local");
        this.utilisateurServiceRemote = (IUtilisateurServiceRemote) context.lookup("UtilisateurService/local");
    } catch (NamingException e) {
        e.printStackTrace();
    }
}

```

Pour chaque type de commande, on décrit le comportement à suivre dans la méthode "execute". Ce comportement correspond en général à une séquence d'appels à différents services de la logique métier (beans sessions), avec des vérifications sur les appels et les résultats obtenus. Si l'exécution de cette commande ne permet pas d'obtenir de résultat correct, on appelle la méthode "onFailure" du callBack en indiquant la nature de l'erreur rencontrée. Sinon, on fait appel à la méthode "onSuccess" en retournant le résultat attendu.

A titre d'exemple, considérons le comportement associé à la création d'une idée (CommandeCree-Idee).

```

public void execute() {
    final IIdee ide = this.ideeServiceRemote.getIdéeById(this.idee.
        getIdentifiantServeur());
    final Utilisateur uti = this.utilisateurServiceRemote.getUtilisateurById
        (this.idee.getCreateur().getIdentifiantServeur());

    // On s'assure :
    // - que l'idée n'existe pas déjà sur le serveur
    // - que l'utilisateur souhaitant la créer existe
    // - que l'utilisateur souhaitant la créer correspond bien à l'
        utilisateur authentifié
    if ((ide == null) && (uti != null) && (this.identification.
        equalsIgnoreCase(this.idee.getCreateur().getIdentifiantServeur())))
    {
        this.callback.onSuccess(this.ideeServiceRemote.creerIdée(this.
            idee));
    } else {
        if (ide != null) {
            this.callback.onFailure(new Exception("Idée déjà à
                existante"));
        } else {
            this.callback.onFailure(new Exception("Identifiant de
                l'utilisateur invalide"));
        }
    }
}

```

5.3.4 La CommandeQueue : implémentation du composant MessageQueing

Une fois la commande construite, elle sera stockée dans une file de commandes (classe CommandeQueue). Cette file de commande est unique dans le système (Design Pattern Singleton), et permet ainsi de

séquencialiser l'exécution des commandes. Le Reacteur crée un Thread chargé d'exécuter la commande (créé par le Reacteur) en instanciant la classe ThreadDeCommande. Lorsque le Reactor exécute ce thread (via la méthode "run"), il fait appel à la méthode "executionCommande" de la CommandeQueue, faisant elle-même appel à la méthode "execute" de la commande (présentée dans le paragraphe précédent). On remarque que le comportement est strictement identique à une file d'ordonnancement (les commandes s'exécutent selon l'ordre dans lequel leur exécution a été demandée par le ThreadDeCommande), bien qu'aucune file ne soit utilisée : on ordonne les appels concurrents à la méthode "executionCommande" de la CommandeQueue, en la préfixant du mot-clé "synchronized", n'autorisant qu'une seule exécution à la fois sur le serveur :

```
/*
 * Lance l 'execution de la commande.
 * @param c La commande a executer.
 */
public synchronized void executionCommande( final Commande c ) {
    c.execute();
}
```

5.4 Informations techniques utiles

Dans cette partie, nous regrouperons toutes les informations techniques nécessaires au lancement de notre serveur et à son déploiement. Ces informations pourront être utiles à l'équipe chargée de la maintenance ou de la reprise du serveur.

Bibliothèques nécessaires au développement du serveur

Les seules bibliothèques nécessaires au développement du serveur GTD sont les bibliothèques standards de JBoss et d'Hibernate/JPA.

Déployer le serveur GTD sous JBoss

Pour ce faire, il suffit de créer une archive JAR à partir de votre application serveur (sous Eclipse : Export->Jar). Placez le JAR obtenu dans le dossier /server/default/deploy depuis votre JBOSS.

Lancer JBOSS

Une fois le serveur GTD déployé, exécutez la commande "run.sh" (sous Linux et Mac) ou "run.bat" (sous Windows) pour lancer JBoss. Tous vos beans seront alors déployés et prêts à être appelés par les applications clientes.

Accéder à la base de données de l'application

Vous disposez d'une interface Web H2 pour accéder au contenu de la base de données :

1/ Télécharger l'archive H2 à l'adresse suivante : <http://repo1.maven.org/maven2/com/h2database/h2/1.2.124/h2-1.2.124.jar>

2/ Démarrer la console H2 :

```
/* emplacement du jar contenant le driver utilise */
java -cp /mnt/D2/JBoss/jboss-4.2.3.GA/server/default/lib/hsqldb.jar:
     /* jar utilise par la console */
     h2-1.2.124.jar
     /* Main class a lancer dans la console */
     org.h2.tools.Console
```

3/ Pour se connecter à la base de données du serveur :

- Quitter JBoss
- Choisir "HSQLDB"
- Champ JDBC URL : indiquez le chemin de la base de donnée (par exemple : jdbc:hsqldb:/mnt/D2/JBoss/jboss-4.2.3.GA/server/default/data/hypersonic/localDB)
- ne pas oublier de se déconnecter avant de lancer JBoss

5.5 Conclusion

Dans ce paragraphe, nous effectuerons un bref bilan des difficultés rencontrées tout au long du processus de développement du Serveur GTD, de l'analyse à l'implémentation.

5.5.1 Analyse et conception

Bien que nous n'ayons pas utilisé d'outil de génération de code à partir de notre travail d'analyse et de conception, nous avons essayé d'être le plus fidèle possible à nos spécifications lors de l'implémentation du serveur GTD.

C'est le premier projet dans lequel nous avons pu nous appuyer sur la conception de manière aussi poussée, tout en gardant un certain esprit critique pour ne pas traduire "bêtement" nos spécifications, s'il existe une meilleure solution via les technologies employées. Ce travail de spécification était d'autant plus important qu'il aurait dû être la base de travail de toutes les applications clientes, bien que ça n'ai pas été le cas (voir paragraphe suivante : Organisation interne et externe). Nous nous sommes rendu compte, lors de l'implémentation, que de nombreux problèmes et ambiguïtés avaient été levé par nos spécifications, qui se sont révélées relativement pertinentes.

Cependant, une meilleure connaissance des technologies utilisées auraient changé notre PSM¹ et notre processus de développement (il aura fallu commencer par écrire les IDL et définir nos entités serveur à partir des classes Java générées).

Notre travail de conception nous a permis de nous séparer efficacement le travail, d'éviter des problèmes d'implémentations, et de raisonner en termes formels sur les différents problèmes de conception. Cependant, nos impératifs de temps ne nous ont pas permis de tester la génération de code à partir de modèle, ce que nous regrettons mais que nous ne pouvions éviter : réaliser une telle démarche sans scripts pré-écrits aurait été bien trop long. En effet, nous pensons que l'approche M.D.A sur ce type de projet et avec ce type de délai ne peut être intéressante que si l'on dispose d'une série de stéréotypes (Entité Hibernate, module CORBA...) et de scripts associés importante.

5.5.2 Organisation interne et externe

En terme de conception, justement, un travail de coordination entre les différents membres du groupe Serveur ainsi qu'avec les applications clientes a été nécessaire. Chaque membre du groupe Serveur étant affecté à une tâche bien particulière, nous avons dû constamment échanger et débattre. Par exemple, la rédaction des commandes (voir livrable 5) a fait apparaître de nouveaux besoins pour les beans session. Cette coordination interne s'est finalement révélée plus enrichissante que contraignante, chaque membre ayant du recul sur la partie de l'autre et pouvant ainsi avancer une opinion différente.

En ce qui concerne la coordination avec les applications clientes, elle s'est révélée difficile au départ (jusqu'au troisième livrable, les clients ne devaient pas prendre en compte nos spécifications), chaotique par la suite (chaque client ayant une vision particulière du sujet ou des besoins particuliers en terme de fonctionnalité, les demandes de modifications ont été nombreuses et peu ordonnées), puis inexisteante (les applications clientes n'ayant pas eu le temps d'implémenter une communication effective avec le serveur, les modifications que nous avions pris le temps de réaliser étant donc inutiles).

Cet échec peut être expliqué par différents facteurs : premièrement, nous avons commencé notre projet en même temps que les applications clientes, alors que nos spécifications auraient dû être réalisées avant que les autres groupes ne commencent le sujet, afin de servir de base à une réflexion ordonnée. Deuxièmement, nous avons mis beaucoup de temps à implémenter le serveur (toujours pas terminé à deux jours de la date finale), à la fois pour des raisons de temps, de connaissances (nous devions attendre d'avoir fini les cours sur corba avant de faire les IDL, sur Hibernate avant de faire les beans...) et

1. Platform Specific Model, modèle spécifique aux technologies utilisées et pouvant être vu comme le modèle le plus proche du code possible

d'organisation. L'attribution d'un chef de projet ou d'un groupe fixant clairement les spécifications sans discussion possible et fixant un calendrier commun à tous les groupes aurait certainement pu améliorer cette organisation.

5.5.3 Implémentation du projet

Partie Communication Distante

Lors de ce projet, nous avons pu constater que RMI était un protocole très performant et facile à mettre en place dans le cadre du développement d'un système distribué. Malheureusement, pour certaines utilisations comme des services web ou plus généralement dans le cas où le client ne connaît pas forcément le serveur RMI n'est pas forcément le protocole le plus approprié. Dans le cadre de ce projet, notre serveur assume un rôle proche de celui d'un serveur web et nous avons donc du adapter le protocole RMI à nos besoins de communication asynchrone grâce au mécanisme de callback. Si nous avions eu le choix de notre protocole, nous nous serions probablement tourné vers JMS qui permet nativement la communication asynchrone et qui ne nécessite pas de stubs mais au final, l'utilisation de RMI se sera révélée être une bonne expérience avec ses avantages et ses inconvénients que nous avons néanmoins su surmonter.

Partie persistance

La partie persistance du projet nous a permis de manipuler la technologie Hibernate et les Entreprise Java Beans de manière poussée, et nous a donné l'occasion de nous frotter à plusieurs problèmes assez complexes (de notre point de vue du moins). Nous avons vite réalisé que les choix de conception que nous avions fait (notamment pour les données partagées avec les clients) n'était pas les plus adaptés pour faciliter la création d'entités (notamment à cause des List vs Set), confirmant le fait qu'un bon concepteur doit avoir de bonnes connaissances dans les technologies qu'il souhaite utiliser.

Face à des problèmes que nous ne pouvions résoudre, nous nous sommes rapidement tournés vers les communautés Hibernate et JBoss pour trouver des réponses, de manière plus prononcée qu'à l'accoutumée, ce qui là aussi nous a permis de développer nos compétences dans la recherche d'information (selon nous primordiale en informatique). La nature même des problèmes est assez différente de ceux auxquels nous sommes habituellement confrontés : configuration et technique, en plus de l'architecture et la conception (qui nous sont plus familiers).

Nous sommes globalement satisfaits du travail effectué sur cette partie, et avons dans tous les cas beaucoup appris avec ce projet, utilisant les technologies JEE/Hibernate sur un problème concret, avec des applications réelles. Avec le recul, nous aurions certainement pris des décisions différentes, mais nous respectons en grande partie nos spécifications et c'était l'objectif que nous nous étions fixés.

Chapitre 6

Annexes

Projet GTD - Objets Distribués R.M.I et Persistance des données

S. Begaudeau
B. Gosset
A. Lagarde
C. Renaudineau

Janvier 2010

Table des matières

1 RMI etCallBack	
section1.1 Bref rappel de la communication RMI avec le serveur	3
1.2 Le mécanisme de CallBack	3
2 Persistance des données : Hibernate et EJB	6
2.1 Conception des beans entités	6
2.1.1 Partie commune entre clients et serveur	6
2.1.2 persistance des entités : EJB Entity et Hibernate	7
2.2 Conception des beans sessions : la logique métier	9
2.2.1 Interfaces des beans sessions	9
2.2.2 Création et suppression d'une entité	9
2.2.3 Mise à jour d'une entité	10
2.2.4 Recherche d'entités	10
2.3 Appel aux beans sessions	10
2.3.1 Localisation du bean session	10
2.3.2 Appel aux fonctionnalités proposées	11
2.4 Configuration de la base de données	11
2.5 Tests unitaires	12
2.6 Bilan sur le travail effectué	12

Introduction

Dans ce rapport, nous reviendrons sur deux des principales technologies employées dans notre projet GTD, à savoir RMI et Hibernate. Plus globalement, nous détaillerons l'intégration de ces technologies à nos spécifications, en expliquant les problèmes rencontrés et les solutions apportées.

Dans un premier temps, intéressons-nous à RMI, un procédé de communication distante, et du système de CallBack, permettant de "casser" l'aspect synchrone de RMI pour un traitement des requêtes asynchrone (fondamental pour un serveur).

Chapitre 1

RMI et CallBack : une communication asynchrone

1.1 Bref rappel de la communication RMI avec le serveur

La communication avec le serveur (en RMI comme en CORBA) s'effectuera en 4 étapes clés :

1. La **récuperation du serveur** afin de pouvoir effectuer des appels de méthodes distants.
Cette récupération s'effectuera via l'interrogation d'un annuaire de services.
Exemple d'appel :

```
ServeurGTD serv = (ServeurGTD)Naming.lookup("rmi://" + host+ "/ServeurGTD_RMI");
```
2. **Identification** auprès du serveur : chaque utilisateur d'une application GTD possédera un compte, avec un login et un mot de passe, qu'il devra renseigner pour se connecter.
Suite à cette identification, le serveur renverra un jeton de connexion (chaîne de caractères cryptée), identifiant l'utilisateur de manière unique et sécurisée. Ce jeton devra être passé en paramètre de tous les futurs appels au serveur, jusqu'à la déconnexion du client.
Pour plus de détail, veuillez vous référer au livrable 3 de GLO.
3. **Appel aux fonctionnalités** du serveur, comme par exemple l'insertion d'une tâche. Comme nous le verrons, la réponse renvoyée par le serveur repose sur les mécanismes d'identification et de CallBack (voir partie suivante).
4. **Déconnexion** auprès du serveur : le jeton de connexion fourni devient inutilisable.
Pour pouvoir effectuer de nouveaux appels au serveur, l'application cliente devra refaire une demande d'identification.

1.2 Le mécanisme de CallBack

Issu du Design Pattern du même nom, le callback est un objet qui nous permettra de fournir plusieurs fonctionnalités simplement, pour cela regardons un court exemple.

Sans callback :

```

public class Main {
    public static void main(String[] args) {
        String port = args[0];
        MonObjet obj = (MonObjet) c.lookup("rmi://" + port +"MyObject");

        try {
            Integer result = obj.donneMoiMonResult("mon_nom");
            traiteur(result);
        } catch (RemoteException e) {
            traiteurException(e);
        } catch (ResultException e) {
            traiteurException(e);
        }
    }

    public static void traiteur(Integer i) {
        // traiter le resultat
    }

    public static void traiteurException(Exception ex) {
        // traiter l'exception
    }
}

public interface MonObjet {
    public Integer donneMoiMonResult(String nom) throws ResultException;
}

```

Sans mécanisme de callback, les clients seraient confrontés à des méthodes synchrones, et donc bloquantes, en utilisant CORBA comme RMI. De plus, le serveur serait aussi bloqué durant le traitement d'une requête, ou alors il devrait mettre en place un système d'attente active (while(true)...) pour garder la pile d'exécution de l'appel venant du client, jusqu'au moment où la réponse est prête à être transmise. Le serveur deviendrait ainsi pratiquement mono-utilisateur, et les clients devraient créer de nouveaux threads pour chacune de leurs requêtes, avec la gestion de la concurrence qui va avec.

Le mécanisme de callback peut être vu comme un pattern observeur amélioré. En effet, avec le pattern observeur, un objet observeur va s'enregistrer auprès d'un objet observé et il informe ce dernier (par son interface) que si un événement intéressant à lieu alors il doit le prévenir avec sa méthode notify. Avec le pattern callback, l'objet qui observe fourni un objet callback à l'objet qui est observé, lorsque ce dernier a une information intéressante à transmettre à l'observeur, alors il appelle la méthode de son choix sur l'objet callback.

Avec callback :

```

public class Main {
    public static void main(String[] args) {
        try {

            String port = args[0];
            MonObjet obj = (MonObjet) c.lookup("rmi://" + port +"MyObject");
            obj.donneMoiMonResultAsync("mon_nom", new CallBackClient());
        } catch (Exception e) {
            e.printStackTrace();
        }
    }

    public static void traiteur(Integer i) {
        // traiter le resultat
    }

    public static void traiteurException(Exception ex) {
        // traiter l'exception
    }
}

```

```

public class CallBackClient implements CallBack<Integer> {
    public CallBackClient() {
        UnicastRemoteObject.export(this);
    }

    public void onFailure(AsynchException ex) throws java.rmi.RemoteException {
        Main.traiterException(ex);
    }

    public void onSucces(Integer i) {
        Main.traiter(i);
    }
}

public interface MonObjet {
    public void donneMoiMonResultAsync(String nom, CallBack<Integer> callback);
}

public interface CallBack <T> extends java.rmi.Remote {
    public void onFailure(AsynchException ex) throws java.rmi.RemoteException;
    public void onSuccess(T i) throws java.rmi.RemoteException;
}

```

Dans le cas ci-dessus, il faut imaginer que le serveur reçoit l'appel de "donneMoiMonResultAsync", qu'il crée un nouveau thread avec le callback et le string, puis que la méthode donneMoiMonResultAsync se termine, coupant la pile d'exécution et redonnant la main au client. Il est à noter que dans la version avec callback, le bloc try catch n'est présent que pour les exceptions provenant de rmi (RemoteException), pas pour les exceptions provenant des méthodes du serveur qui elle serait transmise par l'appel à "onFailure".

```

public void donneMoiMonResultAsync(String nom, CallBack<T> callback) {
    ThreadDeCalcul tdc = new ThreadDeCalcul(nom, callback);
    tdc.start();
}

```

Le thread "ThreadDeCalcul" prend alors 25min pour effectuer l'opération côté serveur, puis à la fin de ce calcul, il appelle la méthode "onSuccess" de l'objet callback, celui ci déclenche alors l'exécution du code "traiter(i)" coté client. Ce code est déclenché coté client car le callback n'est pas sérialisable, donc seul une référence sur le callback est passée au serveur, l'objet callback reste toujours coté client. Le serveur peut réaliser l'appel sur le callback car lorsque le client utilise la méthode lookup pour demander au registre rmi du serveur une référence sur le serveur distant, il s'enregistre sur le registre rmi aussi, permettant ainsi à la méthode "UnicastRemoteObject.export(this)" du constructeur du callback d'exporter une référence du callback dans le registre rmi, permettant alors au serveur de rappeler le client ultérieurement. Le principe s'applique aussi dans le cadre de CORBA.

Chapitre 2

Persistante des données : Hibernate et EJB

Dans cette partie, nous détaillerons l'ensemble des EJB (Entreprise Java Beans) mis en place pour assurer le traitement de la logique métier. Nous reviendrons sur les choix effectués, ainsi que sur les différents problèmes rencontrés, avant d'effectuer un bilan du travail réalisé.

2.1 Conception des beans entités

Attardons-nous brièvement sur les différents beans entités que nous avons été amené à réaliser. Dans le cadre de ce projet, nous devions proposer un ensemble d'interfaces représentant les données GTD (Idees, Tâches, Projets, Contacts...), afin de fixer une norme commune à toutes les applications clientes, notamment pour garantir une bonne inter-opérabilité.

Au vu de cette problématique particulière, nos beans entités (représentant les entités du domaine, à savoir les Idees, Tâches, Projets...) devront implémenter ces interfaces. Amenés à écrire du code utilisable par les applications clientes au sein de ces entités (déclaration commentée des attributs, getters et setters...), nous avons séparé chaque bean entité en deux parties :

1. la partie commune entre clients et serveur ;
2. la partie persistance (annotations hibernate...), uniquement assurée par le serveur.

2.1.1 Partie commune entre clients et serveur

La partie commune entre client et serveur étant fidèle à nos spécifications et relativement triviale, nous ne reviendrons pas dessus en détail. Pour chaque entité du domaine (par exemple Projet), on proposera une interface (IProjet) présentant l'ensemble des getters, setters et méthodes indispensables à la bonne représentation de cette entité. Cette base minimale a été déterminée à partir de la spécification de la méthode GTD, ou du moins de l'interprétation que nous en avons faite. Toutes les entités envoyées par le client devront implémenter ces interfaces.

Chaque entité se verra également munie d'une classe abstraite (AbstractProjet), proposant un ensemble de fonctionnalités que nous pensons utiles aux clients (constructeur par recopie, setters, méthodes utiles...). Cependant, les clients pourront bien évidemment concevoir leur Projet sans se baser sur cette

classe abstraite.

Il est à noter que, puisque ces entités seront ammenées à circuler sur le réseau (via des protocoles de communication distante comme RMI ou CORBA), elles devront être sérialisables (et donc implémenter l'interface Serializable en Java).

2.1.2 persistance des entités : EJB Entity et Hibernate

Declaration d'un bean Entity sous Hibernate

Les entités du serveur seront représentées par un EJB entité, implémentant l'interface et héritant de la classe abstraite correspondante. La persistance des entités sera assurée par Hibernate, un mécanisme de persistance proposé par JBoss (bien que pouvant être utilisé indépendamment).

Pour déclarer un bean entité, on préfixe la déclaration de la classe par l'annotation javax.persistence.Entity (faisant partie de JPA). On peut également préciser le nom de la table que la base de données utilisera pour stocker tous les projets :

```
@Entity  
@Table(name = "PROJET")  
public final class Projet extends AbstractProjet {}
```

Traitement des identifiants serveurs

La gestion des identifiants attribués à chaque entité est selon nous primordiale : en effet, ces identifiants doivent être peu évident à décoder, pour des raisons de sécurité.

Prenons un exemple concret : si nous avions utilisé des identifiants séquentiels, il aurait été très facile pour un client d'accéder aux données d'un client B. Par exemple, si A possède 2 tâches dont les identifiants valent 3 et 5, il est facile de supposer qu'il existe une tâche d'identifiant 4 et d'essayer d'y accéder. Globalement, nous voulions que le client n'ait aucun moyen de deviner la structure de la base de données.

Nous avons donc utilisé une stratégie de génération d'identifiants un peu plus complexe, à savoir l'utilisation d'UUID (Universally Unique Identifier). Ces identifiants uniques sont codés sur 128 bits et sont produits en utilisant des composantes pseudo-aléatoires ainsi que les caractéristiques du serveur sur lequel il a été généré (numéro de disque dur, adresse MAC...). Il est facile d'indiquer à Hibernate une stratégie de génération d'identifiant différente de celle par défaut :

```
@Id  
@GeneratedValue(generator = "system-uuid")  
@GenericGenerator(name = "system-uuid", strategy = "uuid")  
public String getIdentifiantServeur() { }
```

Traitement des relations inter-entités

Nous avons rencontré de nombreux problèmes liés à la déclaration des relations entre deux entités du serveur. Nous reviendrons ici sur les principales difficultés auxquelles nous avons été confrontées, ainsi que sur les différentes solutions que nous avons apportées.

Premièrement, nous avons dû interdire le chargement paresseux (Lazy Loading), pour une raison simple : toutes nos entités devant pouvoir être récupérées puis traitées à distance, il est impossible par exemple de charger les tâches contenus dans un projet de manière paresseuse. On interdira ce type de chargement en forçant l'initialisation des attributs (fetchType = eager). On précisera également le type

de comportement que l'on souhaite qu'Hibernate adopte vis-à-vis des opérations en cascade. Comme en cours, on précisera des mappedBy pour lier les relations *..1 (ManyToOne) et 1..* (OneToMany).

Par exemple, un projet possède une liste de tâche (relation OneToMany), chaque tâche possédant un attribut projetConteneur stockant l'unique projet dans lequel elle est contenue. On préfixe le getter de cette liste de tâche dans la classe projet par l'annotation suivante :

```
@OneToOne(mappedBy = "projetConteneur", fetch = FetchType.EAGER, cascade = CascadeType.ALL)
    private Set<Tache> getListeDesTaches() { }

// Dans la classe Tache, on a l'annotation suivante :
@ManyToOne
    public Projet getProjetConteneur() {
        return this.projetConteneur;
    }
```

Il est à noter que nous avons rencontré plusieurs problèmes liés à l'utilisation de Listes plutôt que d'ensemble (Set). La communauté Hibernate préconise en effet d'utiliser des Set, soulignant le fait que les développeurs utilisent des listes à tors et à travers, alors que sémantiquement il s'agit d'ensembles. Cependant, nous avions déjà fixé les différentes interfaces de nos entités, et ne pouvions plus changer nos entités. Nous avons alors dû utiliser une astuce, en créant deux getters et setters pour les attributs que nous souhaitions transformer en Set :

1. un getter/setter public, destiné au client, et présentant ces attributs comme des listes : c'est celui que nous avons défini dans nos interfaces et classes abstraites ;
2. un getter/setter privé, défini uniquement dans l'entité serveur, permettant à Hibernate de récupérer les informations contenues dans l'entité (via le getter), ou d'affecter à une entité des données contenues dans la base de données (via le setter).

Voici un exemple de ces doubles-paires getters/setters, sur la classe Tache :

```
// Getter public, destine aux clients et non persiste.
@Transient
@Override
public List<IContact> getListeContacts() {
    return contacts;
}

// Getter privé, destine au serveur
@OneToMany(fetch = FetchType.EAGER)
private Set<Contact> getContacts() {
    HashSet<Contact> listeElem = new HashSet<Contact>();
    for(IContact iElem : contacts){
        listeElem.add((Contact)iElem);
    }
    return listeElem;
}

private void setContacts(Set<Contact> listedescontacts) {
    contacts.clear();
    for(Contact t : listedescontacts){
        contacts.add(t);
    }
}
```

Lorsque nous l'avons jugé pertinent, nous avons utilisé l'annotation OrderBy pour trier certaines relations (comme une liste de tâches selon leur priorité et taux d'effort). Nous avons par ailleurs utilisé les techniques vues en cours ou découvertes sur Internet ; nous vous invitons à vous référer au code, doc et commentaires pour de plus amples détails.

Nous avons rencontré de grosses difficultés pour faire fonctionner correctement le mapping Hibernate : bien qu'aucune exception ne soit levée ni aucun warning affiché, la liste des contacts pour une tâche (par exemple) n'est pas correctement persistée. Il s'agit évidemment d'un problème majeur, qui remet en cause

le bon fonctionnement de notre serveur. Nous avons tout essayé pour résoudre ce problème, en nous associant avec plusieurs groupes, et personne n'a trouvé de solution. Certains groupes ont proposé un mapping manuel, mais nous l'avons jugé beaucoup trop coûteux pour être satisfaisant. Nous n'arrivons pas à comprendre la nature de problème, puisque nous avons suivi toutes les bonnes pratiques Hibernate vues en cours.

2.2 Conception des beans sessions : la logique métier

Maintenant que nous avons défini les différentes entités GTD et que nous sommes en moyen d'en assurer la persistance, il va nous falloir proposer un ensemble de services permettant de gérer ces entités, et correspondant aux services fondamentaux du domaine.

2.2.1 Interfaces des beans sessions

Nous avons défini une interface pour chaque bean session déployé. Cet interface fera à la fois office d'interface locale et distante. Lors de la définition de chaque bean session (à l'aide de l'annotation Stateless), on précise les interfaces locales et distantes de ce bean.

```
@Stateless
@Local(value = ITacheServiceRemote.class)
@Remote(value = ITacheServiceRemote.class)
public final class TacheService implements ITacheServiceRemote { }
```

2.2.2 Création et suppression d'une entité

La création d'une entité par le bean session est assez intuitive : le client crée premièrement une tâche T1 (sans renseigner son identifiant). Il fait ensuite appel au service pour persister cette tâche sur le serveur (serviceTache.creerTache(T1)). On réalise alors une copie de T1 (d'où le constructeur de recopie dans l'entité Tâche). On fait ensuite appel à l'entityManager pour persister la tâche ainsi créée, qui lui attribuera un identifiant serveur. On retourne finalement la tâche persistée, afin que le client puisse récupérer l'id généré (T1 = serviceTache.creerTache(T1)).

```
public ITache creerTache(final ITache i) {
    final Tache Tache = new Tache(i);
    this.em.persist(Tache);
    return Tache;
}
```

En ce qui concerne la suppression, on proposera une méthode de suppression totale, supprimant toutes les entités d'un certain type, et une suppression basée sur l'identifiant (via la méthode "find" de l'entityManager).

```
// Suppression de toutes les taches
public void removeAll() {
    final String queryText = "delete Tache";
    final Query q = this.em.createQuery(queryText);
    q.executeUpdate();
}

// Suppression de la tache ayant pour id l'identifiantServeur passé en paramètre
public void removeTacheById(final String identifiantServeur) {
    Tache iToRemove = em.find(Tache.class, identifiantServeur);
    em.remove(iToRemove);
}
```

2.2.3 Mise à jour d'une entité

La mise à jour d'entités est assez intuitive, et s'appuie elle aussi sur le constructeur de recopie de l'entité : à l'aide de la méthode "find" de l'entity manager, on trouve l'entité que le client souhaite mettre à jour. On utilise ensuite la méthode de recopie "copier" (également appellée par le constructeur de recopie), permettant de mettre à jour tous les attributs de l'entité. On demande enfin à l'entityManager d'effectuer la mise à jour dans la base de données.

```
public ITache updateTache(final String identifiantServeur, final ITache tache) {
    Tache iToUpdate = em.find(Tache.class, identifiantServeur);
    iToUpdate.copier(tache);
    em.merge(iToUpdate);
    return iToUpdate;
}
```

2.2.4 Recherche d'entités

Les clients étant souvent amenés à effectuer différentes recherches d'entités en fonction de critères spécifiques à la méthodologie GTD (obtenir toutes les tâches créées par un certain Participant depuis une certaine date, obtenir un projet à partir de son nom...), nos services proposent plusieurs méthodes de recherche.

La recherche d'une entité à partir de son identifiant se réalise très simplement via la méthode "find" de l'entityManager :

```
public ITache getTacheById(final String identifiantServeur) {
    return em.find(Tache.class, identifiantServeur);
}
```

Les recherches plus complexes ont été implémentées en HQL, le langage de requêtage proposé par Hibernate :

```
// Retourne toutes les tâches pouvant être réalisées dans le contexte c
public List<IProjet> getTacheByCtx(final IContexte c) {
    final String queryText = "from Tache where contexte = :ctx";
    final Query q = this.em.createQuery(queryText);
    q.setParameter("ctx", c);
    final List<IProjet> resultList = (List<IProjet>) q.getResultList();
    return resultList;
}
```

Globalement, l'implémentation de méthodes de recherche n'a pas posé de difficulté particulière, puisqu'assez simples.

2.3 Appel aux beans sessions

Les beans sessions réalisés seront appelés par les différentes commandes, correspondant à une série d'actions que le client souhaite réaliser. Nous aurions pu mettre en place un système de transactions (proposé par JBoss), mais les actions spécifiées étant unitaires (mise à jour d'une tâche, création d'un projet...), nous n'avons pas jugé ce système nécessaire.

2.3.1 Localisation du bean session

Lors de la construction d'une commande, on localise les beans sessions auxquels elle aura besoin de faire appel, via la création d'un contexte. Pour récupérer le contexte de nos beans sessions, il nous a fallu préciser un ensemble de propriétés, notamment le port à écouter, puis construire un contexte à partir de ces propriétés :

```

final Properties env = new Properties();
env.setProperty("java.naming.factory.initial", "org.jnp.interfaces.
NamingContextFactory");
env.setProperty("java.naming.provider.url", "localhost:1099");
env.setProperty("java.naming.factory.url.pkgs", "org.jboss.naming");

final Context context = new InitialContext(env);

```

On aurait également pu préciser ces propriétés dans le fichier jndi.properties.

Une fois le contexte renseigné, on peut effectuer des "lookup" pour localiser les services recherchés. Puisque l'on est sur le serveur, on caste le service obtenu avec son interface locale :

```

try {
    this.ideeServiceRemote = (IIdeeServiceRemote) context.lookup("IdeeService/local");
    this.utilisateurServiceRemote = (IUtilisateurServiceRemote)
        context.lookup("UtilisateurService/local");
} catch (NamingException e) {
    e.printStackTrace();
}

```

2.3.2 Appel aux fonctionnalités proposées

Une fois les différents services localisés, la commande pourra y faire appel lors de l'exécution de sa méthode "execute" :

```

public void execute() {
    final IIdee ide = this.ideeServiceRemote.getIdéeById(this.idee.
        getIdentifiantServeur());
    final Utilisateur uti = this.utilisateurServiceRemote.getUtilisateurById
        (this.idee.getCreateur().getIdentifiantServeur());
    ...
}

```

2.4 Configuration de la base de données

JBoss proposant par défaut une base de données embarquée se recréant à chaque lancement, il paraît évident qu'un travail de configuration était nécessaire pour cadrer avec nos spécifications. La configuration de la base de données est très simple avec JBoss : il suffit d'intégrer un fichier persistence.xml au jar qui sera déployé sous JBoss.

Pour utiliser une base de données H2 (bien adaptée à JPA/Hibernate) non embarquée avec JBoss (i.e. que les données sont conservées même si le serveur redémarre), il suffit de déclarer l'unité de persistance suivante :

```

<persistence-unit name="EJB3Test">
    <jta-data-source>java:/DefaultDS</jta-data-source>
    <properties>
        <property name="hibernate.hbm2ddl.auto" value="update"/>
    </properties>
</persistence-unit>

```

On peut ensuite accéder à une interface Web pour parcourir la base de données (voir livrable 5 : Informations Techniques utiles).

Dans les spécifications, nous avions également fait part de notre volonté de réaliser automatiquement des sauvegardes de la base données, envoyées par la suite à une machine distante (par mail par exemple).

Ce système de réplication des données nous aurait permis de proposer le retour à une version antérieure de la base (en cas de crash par exemple), et ce même en cas de panne physique du serveur. Cependant, nous n'avons pas eu le temps de nous attarder sur cette problématique qui, bien que fondamentale pour un projet d'entreprise, ne nous a pas semblée primordiale en terme de fonctionnalités ou d'intérêt pédagogique.

2.5 Tests unitaires

Nous avions bien évidemment prévu de réaliser des tests unitaires pour chacune de nos entités et services. Nous nous sommes renseignés pour mettre en place JBoss TestSuite, une suite de tests destinés à être exécutés sur nos beans déployés sous JBoss. Cependant, le temps nous manquant, nous avons préféré favoriser le développement des fonctionnalités nécessaires au client plutôt que les tests. Dans le cadre d'un projet d'entreprise, nous aurions déployé des tests unitaires pour chaque bean entité et session.

Bien que nous n'ayons pas implémenté de tests unitaires pour chaque bean, nous avons réalisé plusieurs tests en écrivant du code client (localisation puis interrogation des services) afin de vérifier le bon fonctionnement de nos beans.

2.6 Bilan sur le travail effectué

Dans ce paragraphe, nous effectuerons un bref bilan des difficultés rencontrées tout au long des étapes évoquées précédemment.

Premièrement, cette partie du projet nous a permis de manipuler la technologie Hibernate et les Entreprise Java Beans de manière poussée, et nous a donné l'occasion de nous frotter à plusieurs problèmes assez complexes (de notre point de vue du moins). Nous avons vite réalisé que les choix d'implémentation que nous avions fait (notamment pour les données partagées avec les clients) n'était pas les plus adaptés pour faciliter la création d'entités (notamment à cause des List vs Set), confirmant le fait qu'un bon concepteur doit avoir de bonnes connaissances dans les technologies qu'il souhaite utiliser.

Face à des problèmes que nous ne pouvions résoudre, nous nous sommes rapidement tournés vers les communautés Hibernate et JBoss pour trouver des réponses, de manière plus prononcée qu'à l'accoutumée, ce qui là aussi nous a permis de développer nos compétences dans la recherche d'information (selon nous primordiale en informatique). La nature même des problèmes est assez différente de ceux auxquels nous sommes habituellement confrontés : configuration et technique, en plus de l'architecure et la conception (qui nous sont plus familiers).

En terme de conception, justement, un travail de coordination entre les différents membres du groupe Serveur ainsi qu'avec les applications clientes a été nécessaire. Chaque membre du groupe Serveur étant affecté à une tâche bien particulière, nous avons dû constamment échanger et débattre. Par exemple, la rédaction des commandes (voir livrable 5) a fait apparaître de nouveaux besoins pour les beans session. Cette coordination interne s'est finalement révélée plus enrichissante que contraignante, chaque membre ayant du recul sur la partie de l'autre et pouvant ainsi avancer une opinion différente.

En ce qui concerne la coordination avec les applications clientes, elle s'est révélée difficile au départ (jusqu'au troisième livrable, les clients ne devaient pas prendre en compte nos spécifications), chaotique par la suite (chaque client ayant une vision particulière du sujet ou des besoins particuliers en terme de fonctionnalité, les demandes de modifications ont été nombreuses et peu ordonnées), puis inexisteante (les applications clientes n'ayant pas eu le temps d'implémenter une communication effective avec le serveur, les modifications que nous avions pris le temps de réaliser étant donc inutiles).

Cet échec peut être expliqué par différents facteurs : premièrement, nous avons commencé en même temps que les applications clientes, alors que nos spécifications auraient dû être réalisées avant que les

autres groupes ne commencent le sujet, afin de servir de base à une réflexion ordonnée. Deuxièmement, nous avons mis beaucoup de temps à implémenter le serveur (toujours pas terminé à deux jours de la date finale), à la fois pour des raisons de temps, de connaissances (nous devions attendre d'avoir fini les cours sur corba avant de faire les IDL, sur Hibernate avant de faire les beans...) et d'organisation. L'attribution d'un chef de projet ou d'un groupe fixant clairement les spécifications sans discussion possible et fixant un calendrier commun à tous les groupes aurait certainement pu améliorer cette organisation.

Nous sommes globalement satisfaits du travail effectué, et avons dans tous les cas beaucoup appris avec ce projet, utilisant les technologies JEE/Hibernate sur un problème concret, avec des applications réelles. Avec le recul, nous aurions certainement pris des décisions différentes, mais nous respectons en grande partie nos spécifications et c'était l'objectif que nous nous étions fixé.



*12, rue de la Houssinière
44322 Nantes* _____

Objets Distribués

Rapport CORBA et implémentation d'interfaces en IDL

BEGAudeau Stephane, GOSSET Benjamin, LAGARDE Alex,
RENAUDINEAU Christophe

Master 2 - ALMA

2009-2010

Table des matières

1	Introduction a CORBA	2
2	Le langage IDL	3
3	Implémentation d'interfaces en IDL	4
4	Conclusion	10

Chapitre 1

Introduction a CORBA

En tant que responsable de l'implémentation de l'application serveur GTD, nous nous sommes intéressés de près à l'architecture logicielle CORBA servant au développement de composants et d'ORB (Object Request Broker permettant l'envoi et la réception de requêtes à distance). L'architecture CORBA (Common Object Request Broker Architecture) est un middleware qui intervient dans le cadre d'applications réparties où une communication est nécessaire entre des applications clientes et une application serveur. Chaque composant développé sera écrit dans le langage IDL (Interface Definition Language) sous forme d'interface. En effet les objets CORBA qui composent notre application seront rattachés à une interface IDL qui définira pour le client les opérations possibles sur ces objets. C'est donc dans ce contexte précis que nous avons eu recours aux IDL que nous avons étudiés en cours afin de décrire aux clients les interfaces objets nécessaires pour permettre une communication avec le serveur. En outre, l'avantage de l'utilisation des IDL dans le cadre d'applications CORBA est une promesse de services faite aux clients et une obligation de mis en oeuvre. De plus, les deux parties conservent ainsi un degré d'indépendance au niveau des langages de programmation utilisées pour l'implémentation des applications clientes et serveur.

Chapitre 2

Le langage IDL

Comme nous avions déjà pensé la partie interface de notre application, nous avons dû traduire celle-ci dans le langage IDL en prenant en compte les différentes contraintes et limites de ce langage. De manière générale, nous regroupons sous les termes de contraintes et limites les aspects tels que la non existence de certains types JAVA en IDL, ou l'inexistence de certains concepts comme par exemple la généréricité (pas de template). Dès lors, traduire nos besoins d'implémentation d'interfaces en IDL s'est avéré par moment compliqué puisque nous avions jusqu'alors pensé celle-ci que de manière locale avec les nombreuses possibilités d'implémentation offert par le langage de programmation que nous avions choisi (*Java*). Pour reprendre l'exemple énoncé précédemment, nous avions pensé utilisé le mécanisme des template pour définir des classes génériques pouvant s'adapter à nos besoins, or après de multiples recherches, nous avons pu constater qu'il était impossible de traduire un tel mécanisme en langage IDL. De même en ce qui concerne certains types, IDL ne prend pas en compte les différents types de collection (List, LinkedList, HashMap, ...) que nous souhaitions utiliser au sein de nos interfaces locales Java. Pour répondre efficacement à ce genre de problèmes, nous avons pris la décision d'implémenter de manière simple et opérante nos interfaces IDL et de prévoir un mécanisme au sein de notre programme Java pour matcher les types propres aux IDL et ceux présents uniquement en Java (ex : transformation d'un "tableau" retourné par une méthode définit en IDL en "List" dans notre programme Java). Afin de ne pas rentrer trop dans les détails dès maintenant nous allons présenté ci dessous, une partie de notre fichier IDL reflétant l'implémentation générale que nous avons établi.

Chapitre 3

Implémentation d'interfaces en IDL

Nous avons commencé par faire le choix de n'utiliser qu'un seul module (qui sera traduit par la suite en un package) pour contenir toutes nos interfaces. Nous aurions pu avoir recourt à plusieurs d'entre eux mais cela ne nous paraissait pas forcément pertinent, ni utile de séparer les différentes interfaces sous des modules différents.

Nous avons donc pour commencer la déclaration de notre module :

```
1 module donneespartagees{  
2     };  
3 }
```

Nous avons par la suite jugée bon de définir les types énumérations ensembles afin de les distinguées des interfaces. Ci dessous, un exemple commentée d'énumération présent au sein de notre fichier IDL :

```
1  /**
2   * Enumeration representant les etats d'avancement →
3   *    ↪ possibles d'un projet ou d'une tache .
4   * @author Stephane Begaudreau , Benjamin Gosset , Alex →
5   *    ↪ Lagarde , Christophe Renaudineau .
6   * @version 1.0.0
7   */
8 enum Avancement {  
9     /**
10      * Indique que la tache ou le projet est a realiser .  
11      */
```

```

11     AFAIRE,
12
13     /**
14      * Indique que la tache ou le projet est en attente.
15      */
16     ENATTENTE,
17
18     /**
19      * Indique que la tache ou le projet a ete delegue a →
20      ↪ un participant.
21      */
22     DELEGUE,
23
24     /**
25      * Indique que la tache ou le projet est termine.
26      */
27     TERMINE
28 };

```

Ensuite on retrouve les interfaces que nous évoquions précédemment tel que l'interface IObjetServeur et l'interface IProjet décrivant respectivement l'ensemble des comportements communs à tous les objets stockés sur le serveur et les différentes opérations associées au projet :

```

1 /**
2  * Interface regroupant l'ensemble des comportements →
3  ↪ communs a tous les objets stockes sur le serveur.
4  * @author Stephane Begaudeau, Benjamin Gosset, Alex →
5  ↪ Lagarde, Christophe Renaudineau.
6  * @version 1.0.0
7  */
8 interface IObjetServeur{
9
10    attribute string IdentifiantServeur;
11
12    attribute string DateDeDerniereModification;
13
14 };
15
16 /**
17  * Interface representant les projets.
18  * @author Stephane Begaudeau, Benjamin Gosset, Alex →
19  ↪ Lagarde, Christophe Renaudineau.
20  * @version 1.0.0
21  */
22 interface IProjet : IObjetServeur {

```

```

22     attribute string Nom;
23
24     attribute boolean EstDansPoubelle;
25
26     attribute Avancement avancement;
27
28     attribute IContexte ContexteParDefaut;
29
30     attribute SequenceTache ListeDeTaches;
31
32     attribute SequenceParticipant ListeDeParticipants;
33
34     attribute SequenceProjet ListeDeSousProjets;
35
36     attribute IParticipant Createur;
37
38     attribute SequenceContact ListeContacts;
39
40     attribute boolean DansArchive;
41
42
43 /**
44 * Ajoute une tache au projet.
45 * @param t La tache a ajouter.
46 */
47 void ajoutTache(in ITache t);
48
49 /**
50 * Supprime la tache du projet.
51 * @param t La tache a supprimer.
52 */
53 void supprimeTache(in ITache t);
54
55 /**
56 * Supprime toutes les taches du projet.
57 */
58 void supprimerToutesTaches();
59
60 /**
61 * Archive le projet.
62 */
63 void archiver();
64
65 /**
66 * Supprime le projet, en le mettant a la poubelle.
67 */
68 void mettreALaPoubelle();
69
70 /**

```

```

71     * Restaurer le projet depuis la poubelle .
72     */
73 void restaurer();
74
75 };

```

On remarque que l'interface IProjet hérite de l'interface IObjetServeur ce qui montre la présence de la notion d'héritage en IDL. L'héritage est utilisable, si ce n'est que celui-ci comporte certaines restrictions notamment quand à la redéfinition de noms ou la surcharge d'opérateurs qui sont clairement interdits en IDL. Au sein de la définition de l'interface IProjet, on y trouve des déclarations d'attributs consultables et modifiables à l'aide d'accesseurs tels que les "getters" et les "setters" (get<Name> et set<Name> en JAVA) ce qui nous est indiqué par le mot clé "attribute" (attribute <TypeAttribut> <NomAttribut>). Les types de ces attributs peuvent être des types de base comme par exemple 'string' ou 'boolean', des types énumérées (c'est le cas de 'Avancement'), des types structurées comme 'IContext' ou 'IParticipant' qui sont définis sous forme d'interface de la même manière que 'IProjet' ou encore des types séquencés tels que 'SequenceTache', 'SequenceParticipant', 'SequenceContact' et 'SequenceProjet'. En ce qui concerne la définition des types séquencés nous y reviendront par la suite. Une fois la définition des attributs faite, nous avons la définition des opérations dans ce cas toutes précédées par le mot clé 'void' indiquant qu'aucune valeur de retour n'est renvoyé. La définition des paramètres au sein de la déclaration des opérations comporte le mot clé "in" qui précise que ces paramètres sont des paramètres d'entrées uniquement.

Passons maintenant à la déclaration des types séquences vues auparavant :

```

1  typedef sequence<string> SequenceString;
2
3      typedef sequence<ITag> SequenceTag;
4
5      typedef sequence<ITache> SequenceTache;
6
7      typedef sequence<IContexte> SequenceContexte;
8
9          typedef sequence<IParticipant> SequenceParticipant;
10
11     typedef sequence<IProjet> SequenceProjet;
12
13     typedef sequence<IContact> SequenceContact;

```

Ces types étant utilisées dans différentes interfaces du module, nous avons

fait le choix de les définir indépendamment de celles ci, c'est à dire directement dans le module. On retrouve notamment des séquences de types issues des interfaces présentes au sein du module. Créer un type sequence propre à chaque séquences présentes dans les définitions des attributs des interfaces nous a semblé utile pour ne pas avoir à dupliqué certaines d'entre elles.

Ci dessous l'exemple d'une interface appelée "CallBack" faisant intervenir un type 'objet' en paramètre de sortie d'une opération et un opération ayant recours au soulèvement d'une erreur :

```

1  /**
2   * L'interface de CallBack permettant au client de →
3   *   ↪ recevoir sa reponse.
4   * @author Stephane Begaudeau, Benjamin Gosset, Alex →
5   *   ↪ Lagarde, Christophe Renaudineau.
6   * @version 1.0.0
7   * @param <T> Le type de reponse desire par le client →
8   *   ↪ en cas de succes.
9   */
10  interface Callback{
11
12      exception UneErreure { string message ; short ligne; →
13          ↪ };
14
15      /**
16       * Methode appelee apres la reussite du traitement →
17       *   ↪ sur le serveur.
18       * @param result Le resultat renvoye par le serveur.
19       */
20      void onSucess(out Object result);
21
22      /**
23       * Methode appelee apres l'echec du traitement sur le →
24       *   ↪ serveur.
25       * @param e L'exception renvoyee par le serveur.
26       */
27      void onFailure(inout string message) raises ( →
28          ↪ UneErreure);
29  };

```

L'opération "onFailure" prend en paramètre d'entrée/sortie une chaîne de caractère et lève une exception utilisateur particulière définie plus haut lors de la déclaration de "UneErreure".

Bien évidemment, notre fichier IDL comporte d'autres déclaration d'énumérations et d'interfaces mais les exemples présentés ci-dessus reprennent un

peu près tous les mécanismes auquels nous avons eu recours lors de l'implémentation de cette partie.

Un dernier point reste à ajouter, c'est celui de la présence d'un "Header" dans notre implémentation pour palier certaines erreurs d'exécution dues à la déclaration de types structurés avant même la définition de la structure et ce de manière croisée. L'avantage de notre fichier 'donneespartagees.h' est simple, puisqu'il contient la déclaration des interfaces, ce qui permet lors de la compilation d'éviter des erreurs de non connaissance d'un type structuré.

Une fois le fichier IDL terminé, à l'aide de la commande 'idlj donnees-partagees.idl', on génère les classes Stub (contenant le code permettant le lancement, côté client, de requêtes correspondant aux méthodes des interfaces), les classes Java, les classes Holders et les classs Helper.

Chapitre 4

Conclusion

En conclusion, nous avons réussi à partir de notre fichier IDL à générer les différentes classes (Stubs, Java, Holders, ...) que nous voulions. Malheureusement, en raison d'une organisation inter groupes défectueuse, nous ne sommes pas réellement en mesure de dire si la communication CORBA fonctionne correctement. Nous avons eu néanmoins quelques retours sur les classes générées par la compilation de notre fichier IDL qui ont donné lieu parfois à quelques corrections ou changements. De manière plus générale, l'implémentation d'interfaces en IDL nous a permis de mettre en pratique les connaissances que nous avions acquises en cours et de constater son intérêt au moment de la génération automatique des classes issues de la compilation du fichier IDL.