Rapport de Projet:
ChatSystem

Timothee MacGarry - Merlin Poitou

I. Présentation du fonctionnement général et architecture

Dans cette première partie nous présenterons le fonctionnement général de notre application, et les choix architecturaux qui y sont liés.

A. Connexion

L'utilisateur, après avoir lancé l'application, se trouve sur une première interface où il lui est demandé de choisir son Pseudo. C'est ce pseudo qui servira comme moyen d'identification des autres usagers. Aux yeux des usagers, un utilisateur est uniquement reconnaissable via son pseudo. Cependant, nous y reviendrons plus tard, les discussions étant stockées par rapport à l'Hostname de l'utilisateur avec qui l'on discute, on ne découvre qui se cache derrière un pseudo qu'en arrivant sur l'interface de discussion - où les anciens messages sont affichés.

De façon sous-jacente, dès le lancement de l'application, les serveurs UDP et TCP sont démarrés. Toutes les classes sont instanciées, et une fois les observers mis en place, on envoie un message - un booléen, "true" - sur le canal broadcast UDP. Les serveurs UDP, toujours en écoute, des autres usagers, détectent le booléen, et s'ils ont déjà choisi leur pseudo, l'envoient en unicast à ce nouvel utilisateur. Sinon, ils restent passifs. La réception de ces pseudos par le nouvel utilisateur mène à l'ajout à la liste des utilisateurs actifs.

Sur l'interface graphique, l'utilisateur peut choisir son pseudo. S'il est nul, égal à "true", "false", trop long (20 caractères) ou égal à un pseudo déjà choisi par un utilisateur de la liste active, il est rejeté et un message d'erreur apparaît. Sinon, il est validé et envoyé en broadcast.

C'est la fin de la phase de connexion.

B. Choix de discussion

Une fois le pseudo choisi, l'interface change pour l'interface de choix de discussion. S'il n'y a pas d'utilisateurs connectés, rien n'apparaît. Sinon, il est affiché sous la forme de boutons les pseudos des utilisateurs connectés. Cette liste de boutons est mise à jour en direct, via des observateurs, à la connexion/déconnexion d'utilisateurs ou un changement de pseudo.

Cliquer sur un bouton mène au changement d'interface pour celle de discussion.

C. Interface de discussion

L'interface de discussion, comme les autres interfaces, est très simpliste. Une grande zone de texte qui affiche la discussion, avec en dessous une barre pour écrire les messages, et un bouton d'envoi (presser la touche entrée fonctionne aussi).

À l'arrivée sur l'interface de chat, s'il existe une discussion avec cette utilisateur dans la base de données, - par hostname et non pas par pseudo, car les pseudos peuvent changer - elle est affichée. Sinon, la zone de texte reste vide. La discussion est lancée dès que l'un ou l'autre des utilisateurs envoie un message - pour recevoir un message il n'est pas nécessaire d'être sur la zone de discussion, il sera ajouté à la base de données.

D'un point de vue TCP, les serveurs TCP tournent en boucle, acceptent toute nouvelle conversation d'un nouvel hostname, et créent un thread de réception TCPClient à chaque nouvelle connexion, ajouté à la map des conversations active du ThreadManager (map (utilisateur - thread TCPClient)).

Pour l'envoi de message, s'il n'existe pas de socket actif pour cette conversation - pas de conversation active pour cet utilisateur dans le ThreadManager - un nouveau thread TCPClient est créé, et le socket est utilisé pour envoyer le message. Sinon le socket est récupéré de la map, et de la même façon utilisé pour l'envoi.

D. Menu d'options

Quelle que soit l'interface, une barre d'options - menu - est disponible sur le haut de l'interface. Celle-ci présente des options différentes selon l'interface:

- Disconnection: présente partout, option qui ferme toutes les conversations actives, ferme et arrête l'interface graphique, envoie déconnexion : un booléen "false" en broadcast, puis réalise un System.exit(0). La réception d'un booléen "false" en UDP mène à la fermeture de l'interface de chat si la conversation était active avec cet utilisateur, la suppression du bouton correspondant à l'utilisateur si sur l'interface de choix de discussion, et la suppression de la liste des utilisateurs actifs.
- Change pseudo: disponible partout sauf sur l'interface de choix de pseudo, permet de changer de pseudo, et d'en notifier les autres utilisateurs.
- Change discussion: permet de changer de discussion si sur l'interface de chat.
- Clear conversation: vide la fenêtre de conversation de tous les messages, et supprime la conversation (les messages) de la base de données.

E. Choix architecturaux

1. Interface Graphique - Java Swing

Pour l'interface graphique, nous avons suivi les recommandations du cours et avons réalisé celle-ci en Java Swing. Nous avons opté pour une interface simpliste mais fonctionnelle.

Nous avons voulu la coder manuellement, sans logiciel créant des interfaces automatiques, ce qui explique également pourquoi elle reste visuellement basique.

Cependant, nous estimons qu'elle est fonctionnelle et pratique, donc elle convient parfaitement à notre usage.

Pour ce qui est de son architecture, nous avons opté pour une classe Interface de type JFrame, contenant trois sous classe de type Container représentant nos trois types d'interfaces. Cela permet de pouvoir attacher les observers directement à la classe Interface, et gérer les fonctions à appeler selon les observations reçues grâce à une variable "state" correspondant à l'interface active.

2. Database - JDBC: SQLite

Pour la base de données, nous avons choisi un modèle de bases de données décentralisées, où chaque utilisateur possède sa propre liste d'anciennes conversations. Nous avons premièrement installé le driver JDBC, ce qui nous a permis d'utiliser SQLite pour coder la base de données grâce aux librairies fournies par le driver. Pour créer la base de données, il faut se connecter grâce aux fonctions fournies par JDBC, ce qui génère un fichier si on se connecte pour la première fois. Après cela, nous avons simplement rajouté des fonctions pour créer les tables, insérer des lignes, enlever des lignes, accéder aux lignes indiquées, puis vérifier si les tables existent bien, le tout en se servant SQLite. Nous avons choisi de concevoir cette base de données afin de stocker les messages selon les conversations qu'un utilisateur a avec d'autres utilisateurs. Pour cela, nous créons deux tables, une appelée 'Conversations' stockant les conversations, et une autre appelée 'Messages' stockant les messages. La table 'Messages' contient toutes les informations pertinentes pour garder la trace d'un message (idMessage, utilisateur émetteur du message, utilisateur receveur, date, et le message lui-même), ainsi qu'une clé étrangère qui s'associe à la conversation correspondante avec un idConv.

3. Modèles

Nous avons deux principaux modèles dans notre architecture:

- Les User, définis par un hostname et un pseudo, qui représente les utilisateurs. L'égalité de deux User est réalisée via la comparaison des hostname, puisque le pseudo peut changer.
- Les Messages, qui représentent les messages envoyés lors d'une conversation. Un message est constitué d'un envoyeur, un récepteur (des User), un message (String), et une date d'envoi (qui sert principalement à différencier deux messages qui auraient le même contenu).

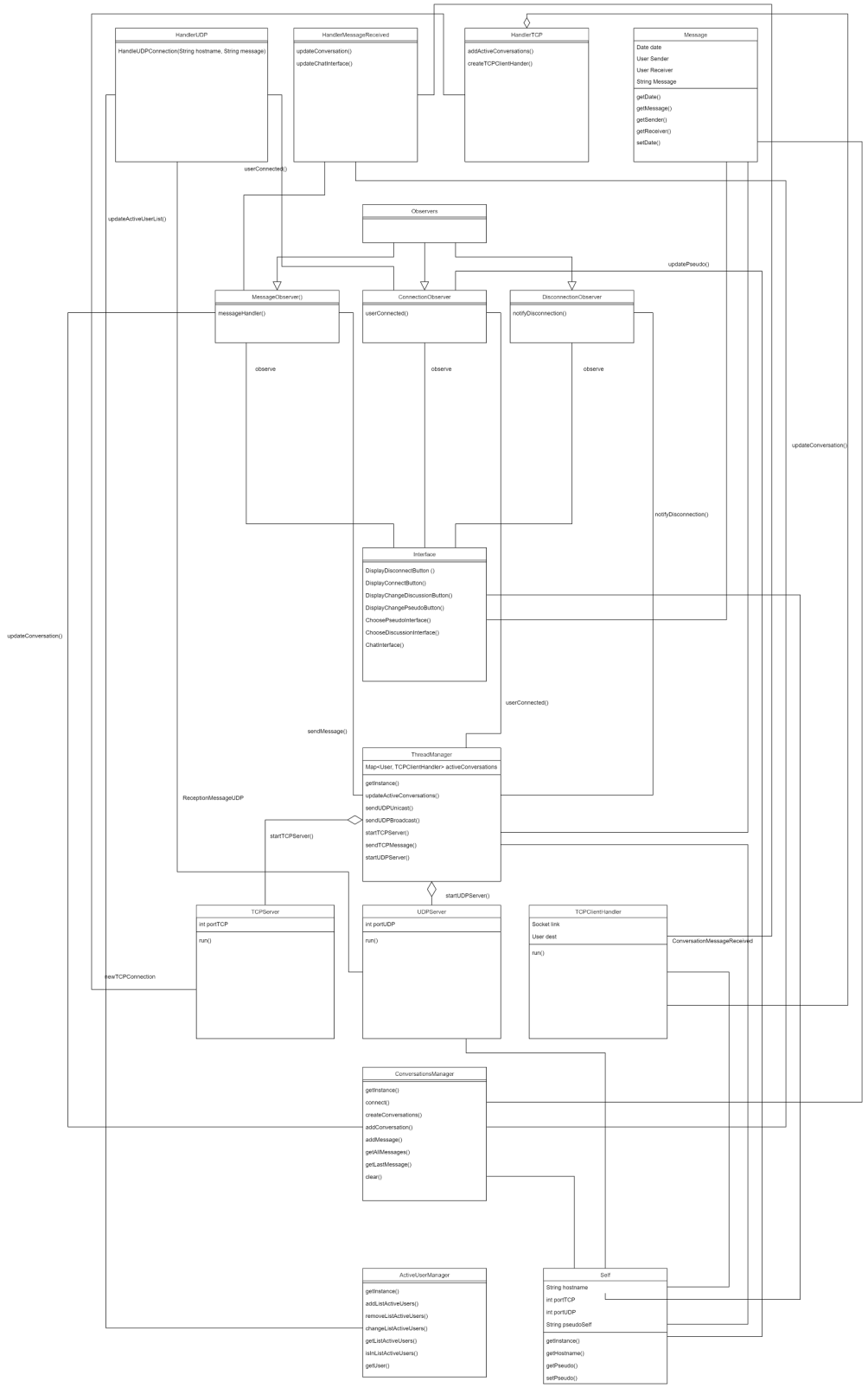
L'on définit également des handlers, et des observers. Les observers sont mis en place pour notifier les classes des autres packages dans trois principales situations:

- Une connexion, auquel cas on transmet le nouvel User, que ce soit une nouvelle connexion extérieure - UDPServer notifie ses observers - ou un choix de pseudo interne - Interface notifie ses observers.
- Une déconnexion, auquel cas on transmet le User concerné, que ce soit une déconnexion externe - UDPServer notifie ses observers - ou l'utilisateur qui se déconnecte - Interface notifie ses observers.
- Une gestion de message: envoi, depuis l'interface qui notifie ses observers, ou une réception, depuis un thread TCPClient qui notifie ses observers.
  Les handlers, définis dans le main, servent lorsqu'une classe appelle des fonctions de classes d'autre package, ou lorsque l'on veut notifier des observers, les classes des autres package étant invisibles depuis l'intérieur - à défaut de Self (qui représente nous même) et ActiveUserList (la liste des utilisateurs actifs).

II. Conceptions et diagrammes

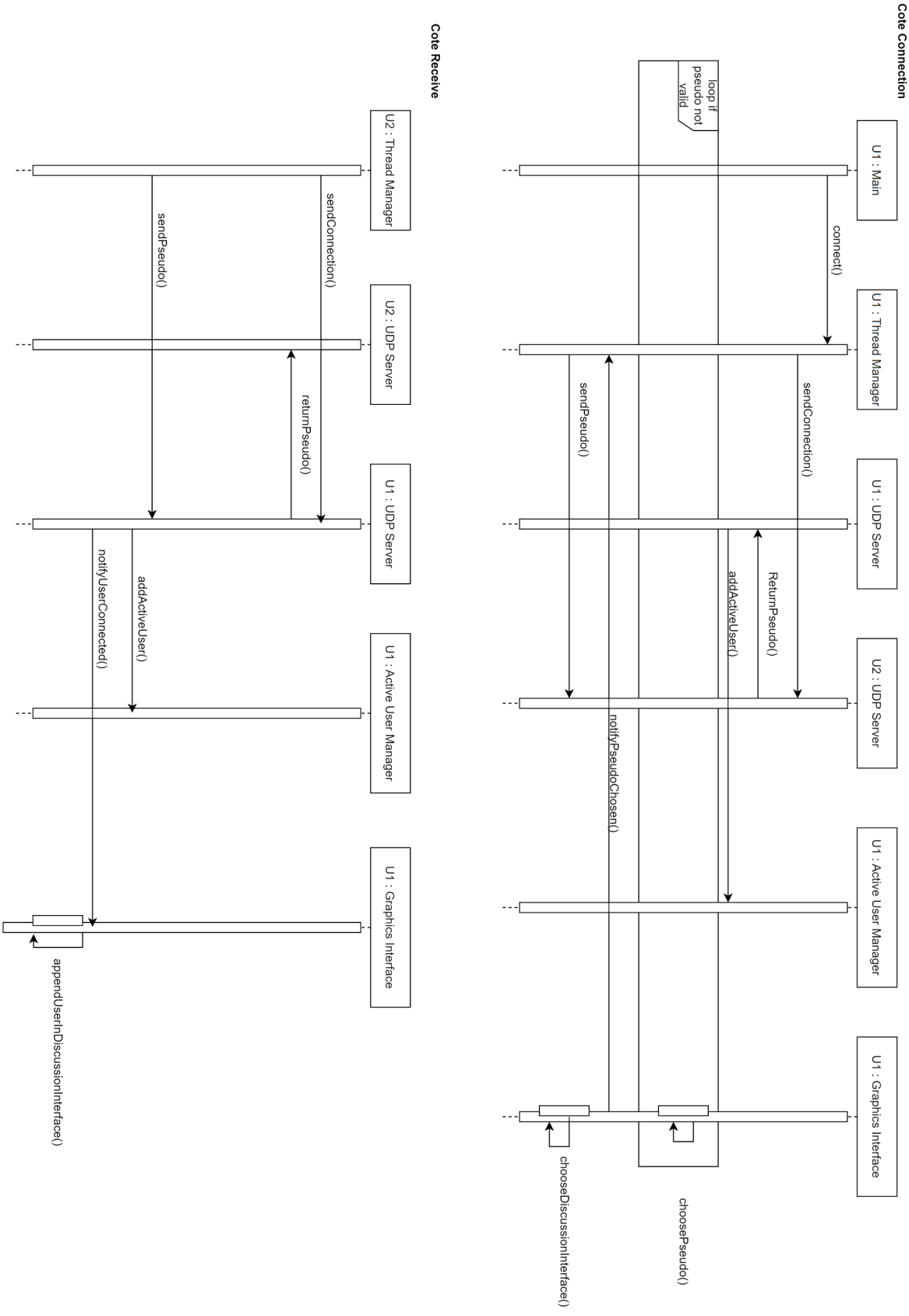
Les diagrammes, peu visibles sur ce rapport, sont également sur le dépôt Git.

A. Diagramme de classe



B. Diagrammes de séquence

1. Connexion



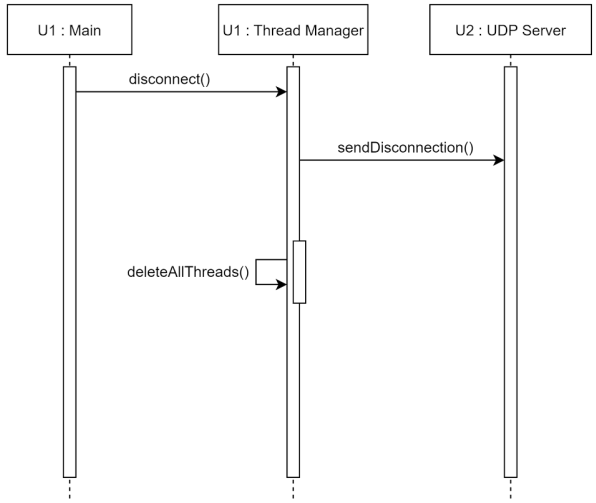
+ Pseudo: String

+ ListActiveUser: String Tab

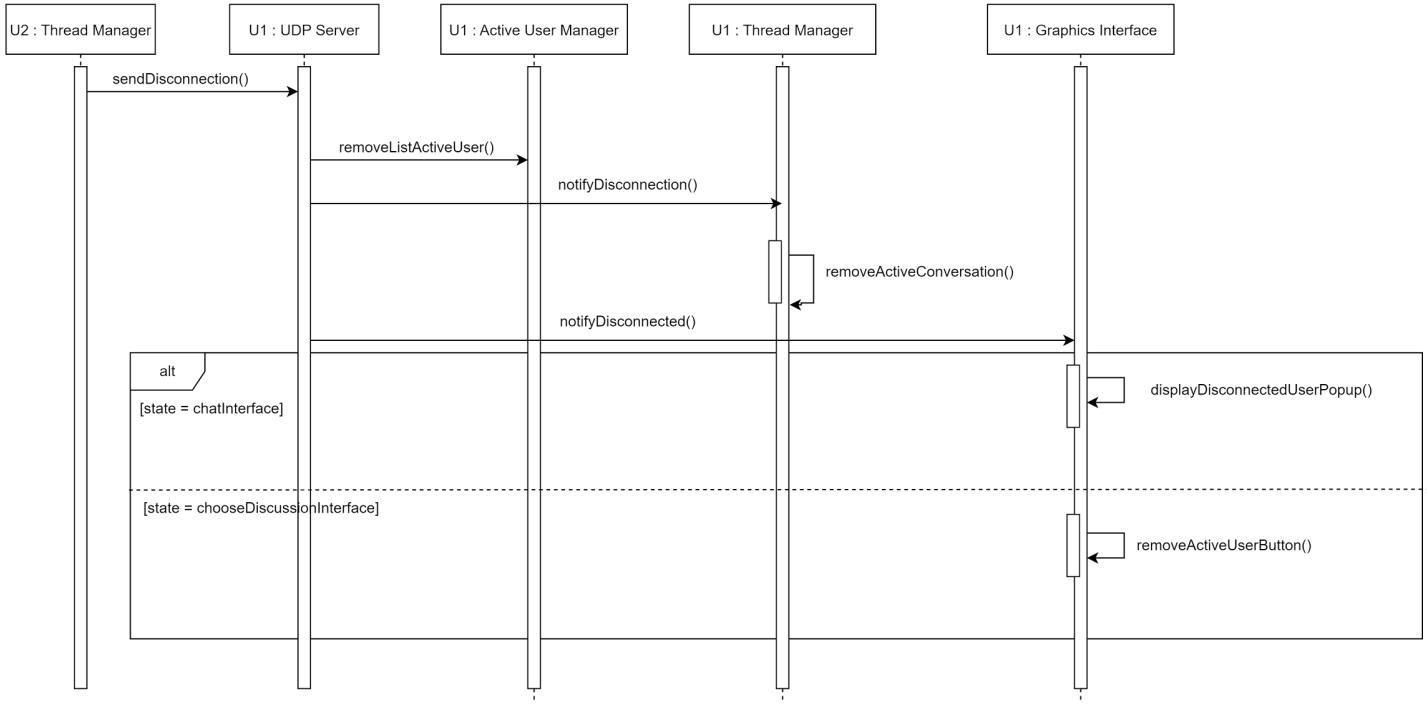
+ Valid: Boolean = False

2. Déconnexion

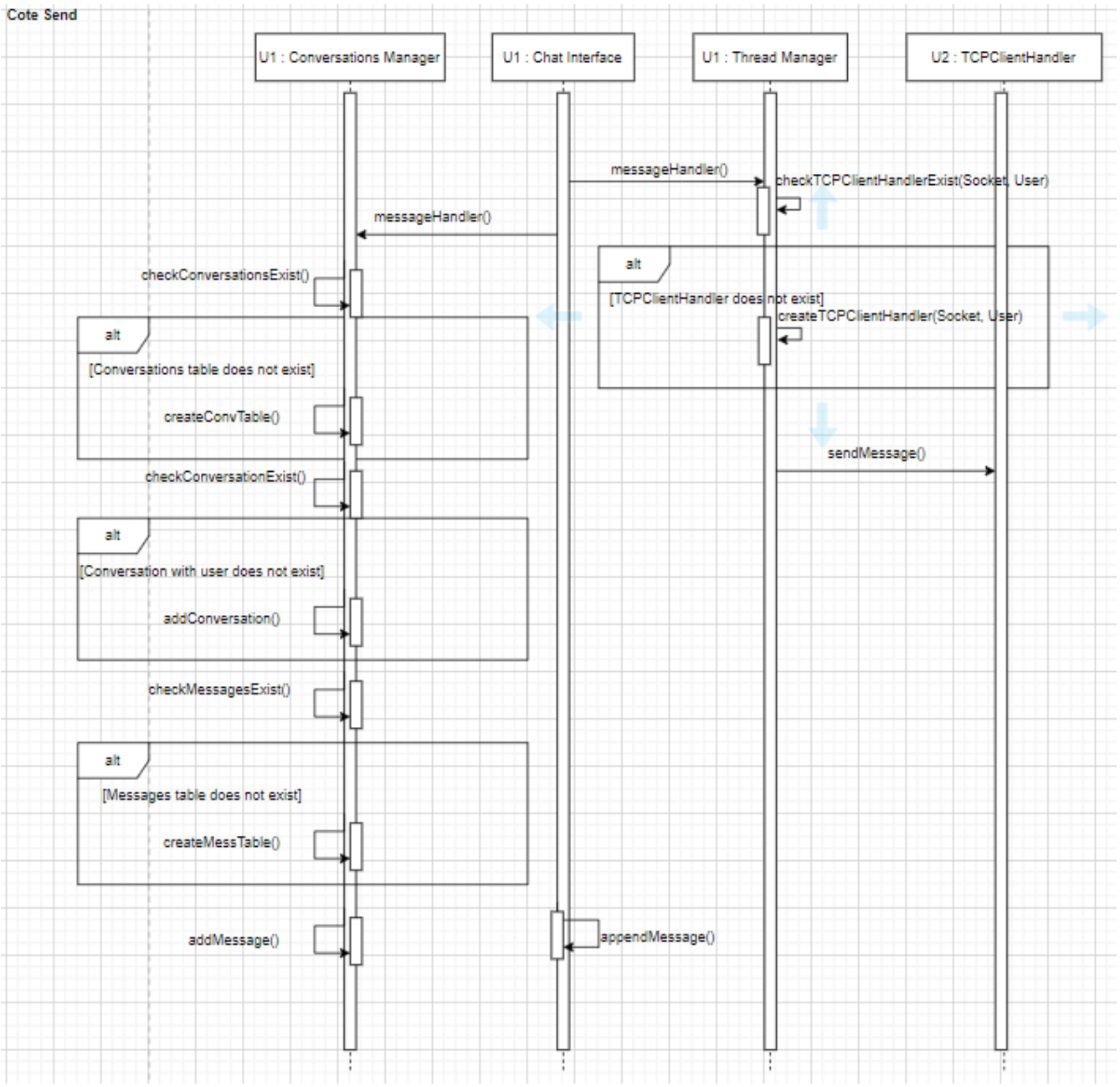
Cote Send

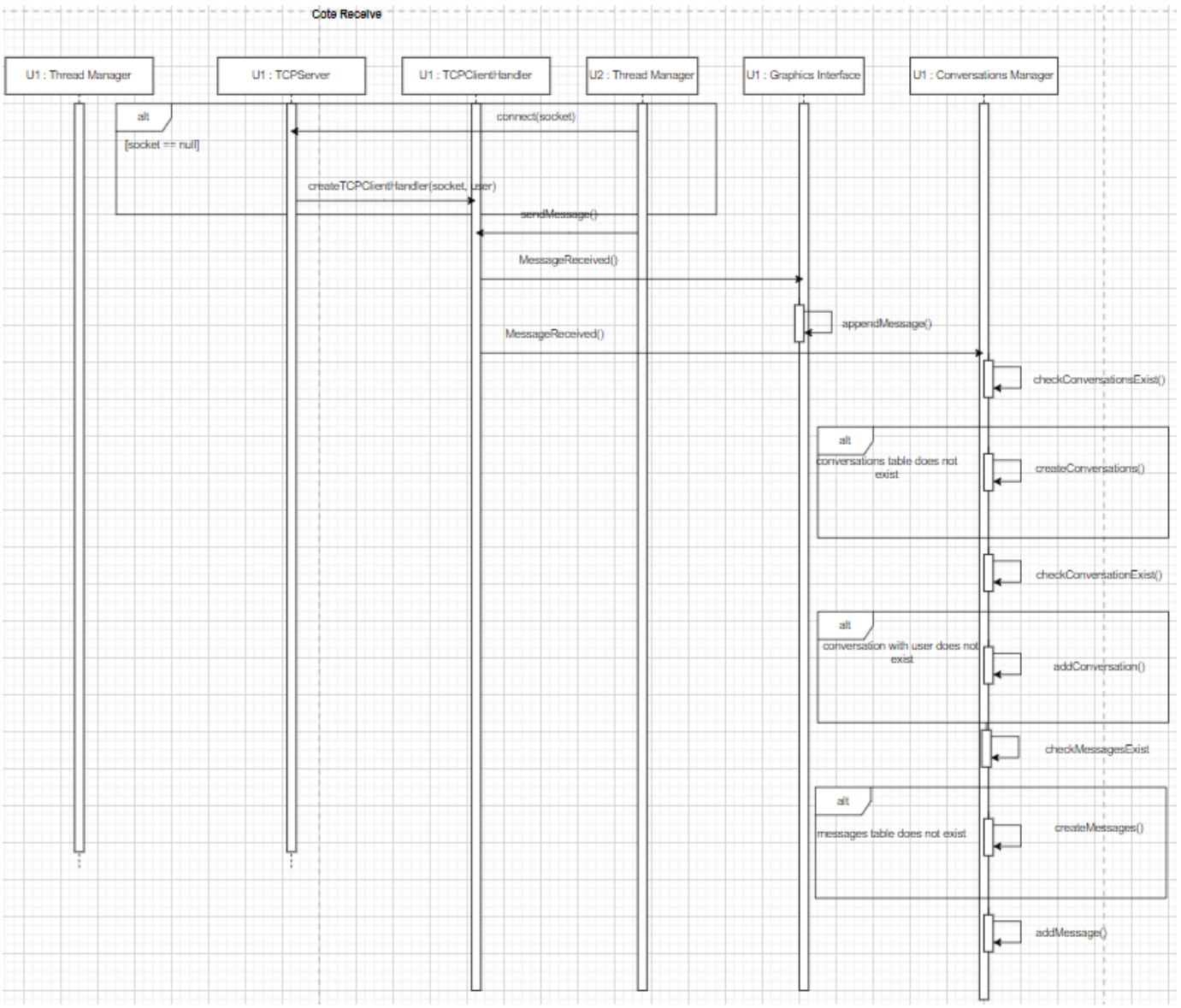


Cote Receive

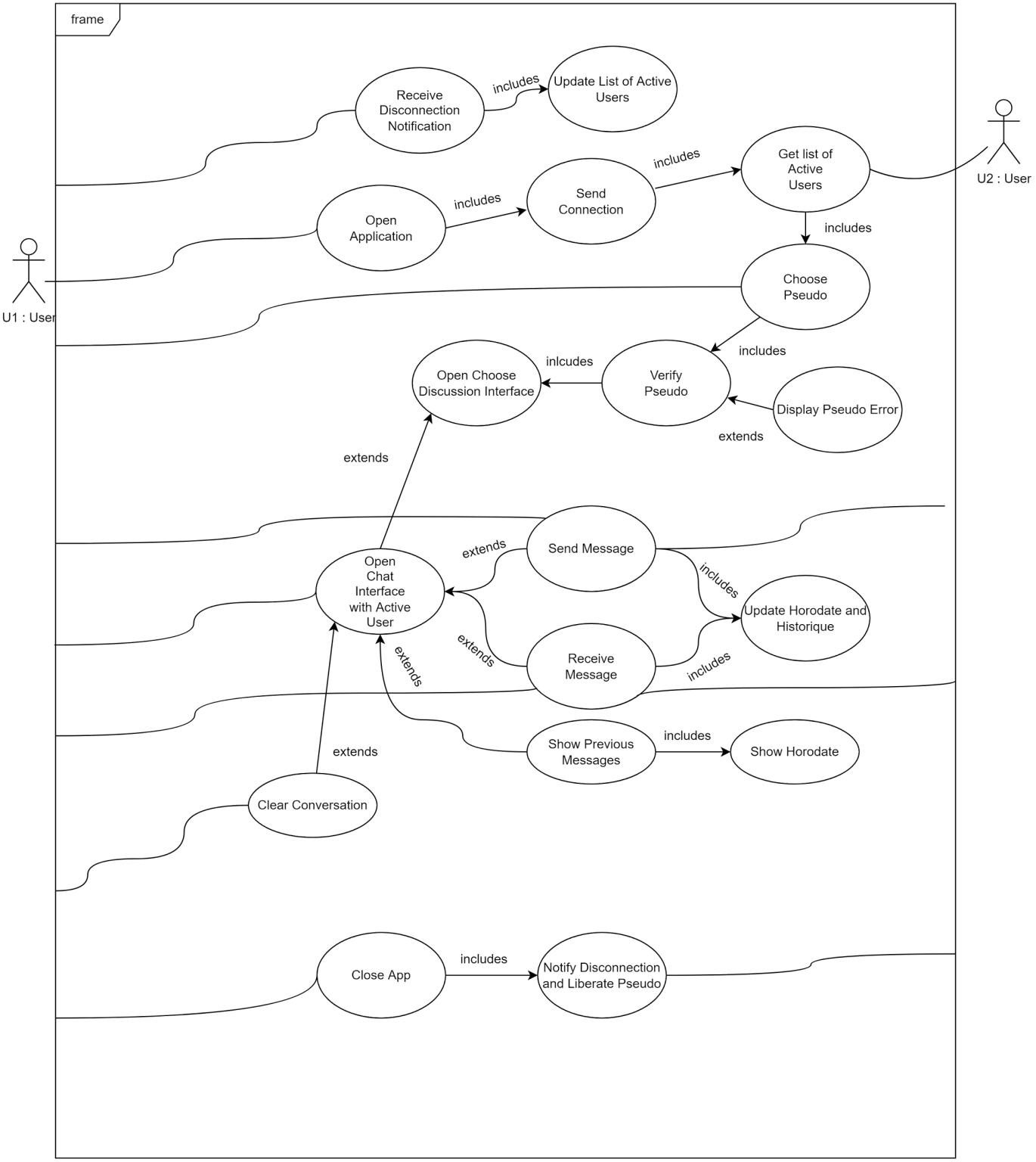


3. Discussion

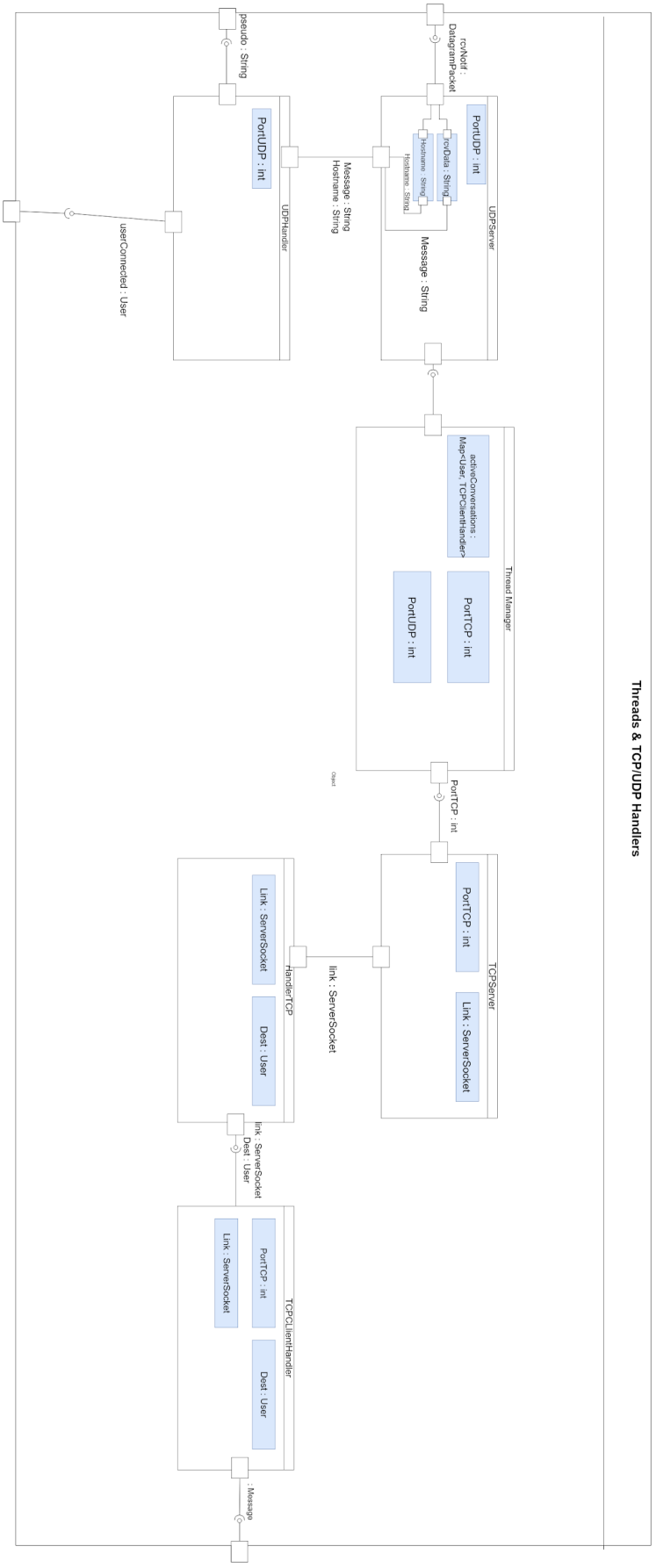



Cote Receive
U1 : Thread Manager
U1 : TCPServer
U1 : TCPClientHandler
U2 : Thread Manager
U1 : Graphics Interface
U1 : Conversations Manager
alt
[socket == null]
connect(socket)
createTCPClientHandler(socket, user)
sendMessage()
MessageReceived()
appendMessage()
MessageReceived()
checkConversationsExist()
alt
conversations table does not exist
createConversations()
checkConversationExist()
alt
conversation with user does not exist
addConversation()
checkMessagesExist
alt
messages table does not exist
createMessages()
addMessage()

C. Use Case


frame
Receive Disconnection Notification
includes
Update List of Active Users
Open Application
includes
Send Connection
includes
Get list of Active Users
U1 : User
U2 : User
includes
Choose Pseudo
includes
Verify Pseudo
inlcudes
Open Choose Discussion Interface
Display Pseudo Error
extends
extends
Send Message
extends
Open Chat Interface with Active User
includes
Update Horodate and Historique
extends
Receive Message
includes
extends
Show Previous Messages
includes
Show Horodate
extends
Clear Conversation
Close App
includes
Notify Disconnection and Liberate Pseudo

D. Diagramme de structure composite



III. Tests

Pour ce qui est des tests, nous avons réalisé quatre principaux fichiers tests - hors les tests manuels d'utilisation réalisés entre deux ordinateurs.

Test des modèles: User et Message. Le test est principalement sur le bon fonctionnement des fonctions equals(), qui nous a créé des bugs à un moment. Ces tests sont très basiques.

Les tests de la base de données étaient effectués en série d'étapes. Premièrement, il fallait supprimer la base de données afin de s'assurer qu'il n'y ait pas de tables, ou de lignes, non désirées. Ensuite, on se connecte à la base de données de test en utilisant la fonction connectDB(String dataBaseName). Une fois effectué, nous créons la table 'Conversations' stockant les conversations d'abords, en ajoutant une conversation simulée dans cette table. Puis nous créons la table 'Messages' qui stock les messages, en insérant un message simulé dans cette table. Il suffit ensuite d'appeler les fonctions qui traitent ces tables afin de vérifier si la base de données fonctionne entièrement.

Test des fonctions UDP. En re-définissant les handlers, nous avons testé le bon fonctionnement de nos fonctions d'envoi et de notre classe UDPServer. Nous avons testé tous les cas ayant lieu pendant le fonctionnement de l'application, à savoir un envoi de booléen "false","true"; ou un envoi de pseudo.

Test des fonctions TCP. En re-définissant les handlers, nous avons testé le bon fonctionnement des fonctions d'envoi et de réception de message TCP ainsi que le fonctionnement du TCPServer. Une fois le serveur lancé, on se connecte à celui-ci, et on lance un thread TCPClient côté client. Le serveur une fois la connexion acceptée et un thread TCPClient lancé, envoi un premier message au client. Le client envoie par la suite un second message au serveur.

Pour ce qui est des tests réalisés manuellement, nous avons testé visuellement le rendu de l'interface graphique, ainsi que le bon fonctionnement de ses différents boutons, ainsi que le fonctionnement de l'app dans son ensemble une fois les handlers et observers mis en place.

Tous les test ont été concluants et les fonctionnalités nous semblent correctes, à ceci prêt, que l'on peut également trouver dans le README:

- Quand un utilisateur1 se déconnecte lors de sa discussion avec l'utilisateur2, et que l'utilisateur1 se reconnecte, il n'apparaît pas dans la liste des utilisateurs actifs/discussions possibles tant qu' utilisateur 2 n'a pas changé également de pseudo. Est-ce que l'utilisateur2 ne prend pas en compte la re-connexion de l'utilisateur1?

- Pour ce qui est de la séparation des classes inter-package, Self et ActiveUserList sont utilisés un peu partout. Plus embêtant, ConversationsManager est utilisé dans l'interface lors de l'affichage des anciens messages. Nous n'avons pas réussi à mettre en place un handler qui récupérerait la liste des anciens messages, cela ne fonctionnait pas, d'où cette transgression des règles de POO.

IV. Manuel d'utilisation

Pour ce qui est de comment utiliser notre application, après clonage du dépôt Git il suffit de suivre les instructions Maven indiquées dans le README.

```
# compile
mvn compile
# run tests
mvn test
# Run main program
mvn exec:java -Dexec.mainClass="Main"
# Clear database
mvn exec:java -Dexec.mainClass="ClearDatabase"
```

Sans maven, il suffira de compiler puis lancer avec java le main:
/src/main/java/Main.java