**2020 - 2021**

Le lien du dépôt Git : <https://github.com/Rayanem/ChatSystem/?fbclid=IwAR10Rgd9muN2UfBgzuKAtNxtfDia8BnwYS1GuxNjpMr1XWiOKiTPvw1cqHA>

**Projet encadré par :**

* Sami Yangui
* Marcel Mateescu

**Projet réalisé par :**

* Rayane MAKRI
* Issam HIHI

**Programmation/Conception orientée objet :**

**Chat system Project**

# **Table des matières**

[**Table des matières** 1](#_Toc64323170)

[**1) Introduction :** 2](#_Toc64323171)

[**2) Guide d’utilisation :** 3](#_Toc64323172)

[**3) Choix d'implémentation :** 6](#_Toc64323173)

[**4) Projet UML :** 10](#_Toc64323174)

[**5) Conclusion :** 15](#_Toc64323175)

# **1) Introduction :**

Ce rapport présente en profondeur l'ensemble des choix derrières notre conception et aboutissant à l'implémentation de notre système.

Notre système va permettre aux utilisateurs de choisir un pseudo unique (un nom qui va permettre de les identifier de façon unique), et leur permettre de communiquer en s’envoyant et réceptionnant des messages lorsqu’ils seront connectés.

Les messages seront textuels mais pourront aussi se composer d’images ou bien de fichiers.

Le système, ainsi créé, va permettre, à chaque utilisateur :

* De choisir un pseudonyme unique avec lequel il sera reconnu dans ses interactions avec le système.
* D’identifier l’ensemble des utilisateurs actifs,
* De démarrer une conversation avec un utilisateur connecté (ou plusieurs utilisateurs connectés) et d’y mettre fin.
* D’informer un utilisateur connecté de la réception d’un message par un autre utilisateur connecté.
* De garantir un historique horodaté des conversations et de les afficher.
* D’informer l’ensemble des utilisateurs connectés du changement de pseudonyme d’un des utilisateurs.

# **2) Guide d’utilisation :**

1- Lancer l'exécutable avec un double clique :

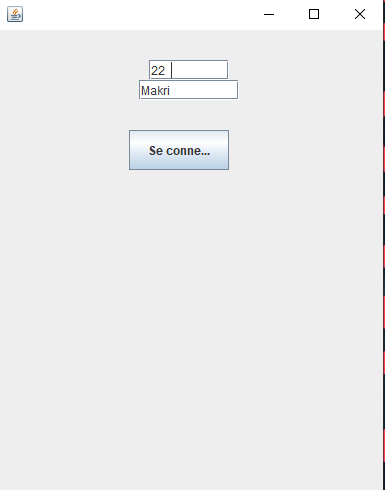


2- Dans le premier cadre vous choisissez un chiffre compris entre 1 et 50.

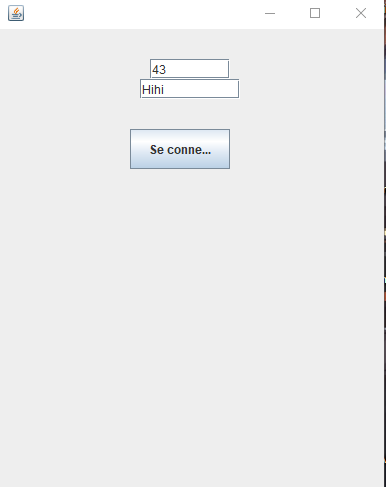
Le deuxième cadre est dédié à votre pseudo, qui va permettre de vous identifier.

Et enfin vous cliquez sur “Se connecter”.

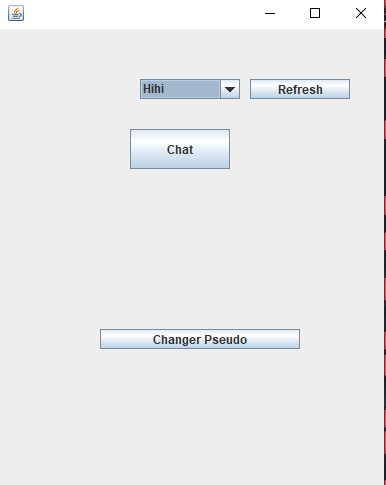
exemple :



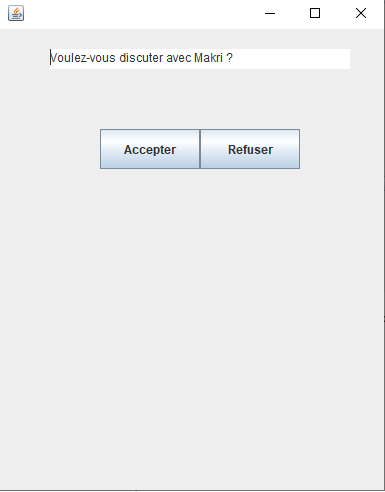
3- Relancer une nouvelle fois l’application pour le test : vous refaites l’étape 2 évidemment !

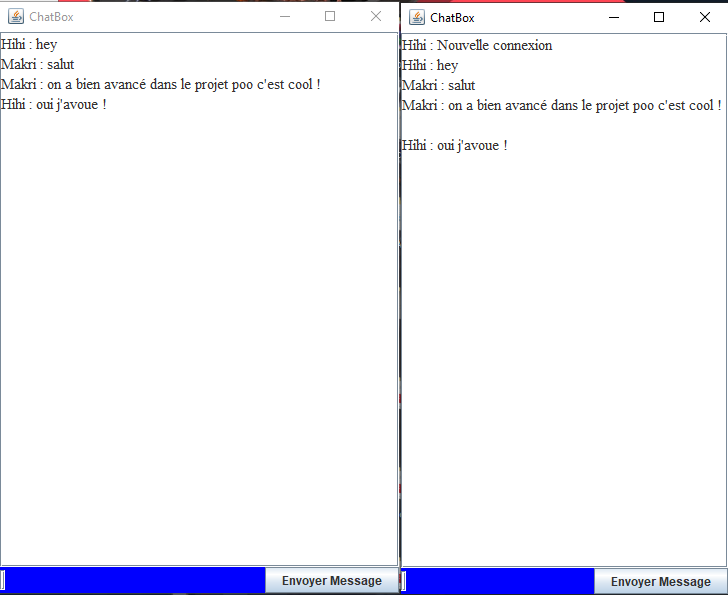


4- Vous pouvez à présent voir les membres connectés, lancer une session de chat, ou bien changer votre pseudo !



5- Quand vous décidé de commencer une session de chat, le destinataire reçoit une demander tout d’abord :



6- exemple de conversation : 

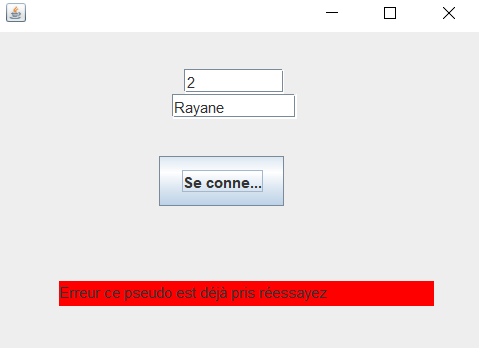
# **3) Choix d'implémentation :**

1. **Communication :**

Dans ce projet, la décision a été prise de réaliser la communication en utilisant les ports à la place des adresses IP. En effet, n’ayant pas la possibilité de tester notre application sur différentes machines en même temps, nous avons simulé ces différentes machines en utilisant les ports d’une seule machine. Ainsi une personne a la possibilité d’ouvrir 4 sessions de chat à la fois et 50 personnes peuvent utiliser l’application en même temps. Ce fonctionnement est identique à l’utilisation des adresses IP mais a été privilégié pour pouvoir tester notre application dans le contexte actuel.

1. **Connexion :**

Pour se connecter, l’utilisateur doit saisir son ID (entre 1 et 50) ainsi que le pseudo qu’il a choisi. L’ID est alors mis en correspondance avec un port de la machine et une demande de pseudo est envoyé en UDP sur tous les ports qui peuvent être utilisés. Cette demande comme toutes les communications est associée à un DemandID. 127 pour les demandes de pseudo. Cet ID est associé au datagram qui va ensuite être lu par un autre client à l’aide d’un thread Receiver qui reçoit les demandes. Ce Thread vérifie que le pseudo n’est pas celui qui lui est associé et répond ensuite. Du côté du demandeur, toutes les réponses sont compilées et si une seule réponse est négative, la demande de pseudo est rejeté et l’utilisateur doit saisir un autre pseudo.



Si la demande est acceptée, le pseudo est inscrit dans la classe Client et un Thread Receiver est créé pour recevoir les futures demandes.

De plus, lorsque le Thread Receiver reçoit une demande de pseudo et y répond favorablement, il va également joindre son pseudo et le port associé. Ainsi, le demandeur aura à la fin de sa demande reçut tous les pseudo et les ports associés qu’il va mettre dans une table de correspondance pour être utilisé plus tard.

Cette table est immédiatement utilisé pour répondre à tous les clients afin de les notifier qu’un nouveau client est connecté et donne les infos qui sont le pseudo et le port du client.

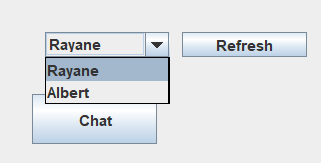
Le client est alors connecté et peut accéder au cœur de l’application. Le choix d’envoyer une demande initiale en UDP puis une réponse en TCP a été faite dans le but que la demande initiale n’a pas forcément besoin d’une réponse alors que la réponse doit absolument parvenir à l’utilisateur initial si elle est reçue. Les DemandID ont été implémentées afin d’avoir plusieurs demandes possibles qui utilisent le même code. Ainsi, l’ajout de fonctionnalité a été plus rapide car il suffisait simplement d’ajouter un ID et le traiter derrière. Le traitement de la correspondance port/pseudo a été fait afin d’éviter le maximum d’échange inutile tout en gardant un fonctionnement fiable. En effet, le dernier message de confirmation est nécessaire car dans le cas où le pseudo est déjà utiliser, il faut notifier tous les autres utilisateurs que la demande à laquelle ils ont donné leur accord n’est finalement pas valable.

1. **Changer de pseudo :**

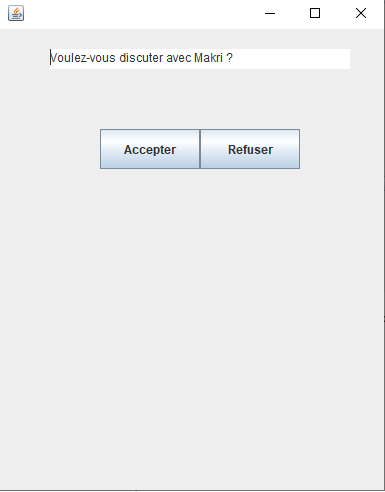
Une fois connecté, la fenêtre principale de l’application s’ouvre et laisse plusieurs choix à l’utilisateur. Attardons-nous sur le changement de pseudo. Dans le cas où l’utilisateur souhaite changer de pseudo, il faut de nouveau vérifier que le pseudo est disponible. On refait alors quasiment les mêmes démarches que pour la demande de pseudo initiale à la différence que le Thread Receiver va vérifier si le port est déjà utilisé. Si c'est le cas, on change seulement le pseudo associé au port afin d’éviter d’avoir deux fois le même client.

1. **Initier une demande de chat :**

L’utilisateur peut alors démarrer une demande de chat en sélectionnant l'utilisateur dans la liste des utilisateurs de l’application



Lorsque l’utilisateur est choisi, en cliquant sur le bouton chat, une demande est envoyée à l’utilisateur sélectionné. Cette demande reçue par le Thread Receiver est traitée et une fenêtre de confirmation est alors affichée.



Cette confirmation est nécessaire car elle permet d’accepter la demande de connexion qui est en attente de l’autre côté (link.accept()). Le client accède alors à la base de données (ce point sera détaillé plus tard) puis démarre un thread de discussion. Ce thread a deux rôles. Tout d’abord démarrer le thread nécessaire à la réception des messages (BufferedReader) et s’occuper de l’envoi des messages (PrintWriter)

Du côté de l’autre client attendant la demande de connexion, une fois la demande acceptée, la connexion est lancée et le thread est débloqué permettant alors de commencer la communication. Le traitement est sensiblement le même des deux côtés mais à dû être fait dans deux classes différentes afin de pallier aux différences de traitement entre la personne qui engage la conversation et celle qui reçoit la demande (notamment l’acceptation de la communication).

Lorsque l’on reçoit un message, on le garde en mémoire puis on l'affiche dans l’interface. Le stockage en mémoire sera nécessaire plus tard pour la base de données. Quand on envoie un message, l’interface appelle le thread nécessaire à l’envoie des messages, le message est affiché puis envoyé à l’aide de du thread.

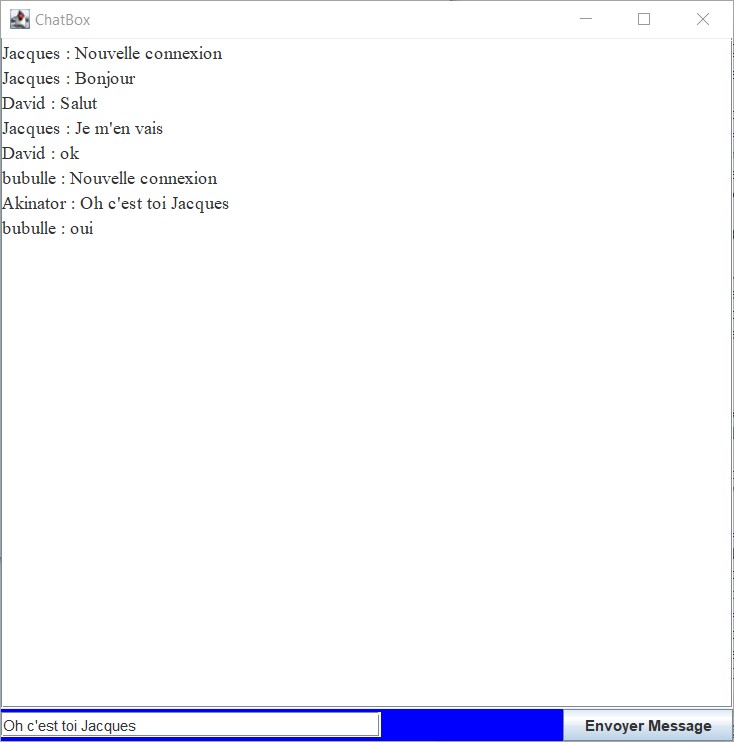
Pour mettre fin à la conversation, une des deux personnes peut envoyer quit sur le chat. Ceci aura pour effet de fermer la communication et d'enregistrer la conversation dans la base de données.

1. **Base de données et stockage de l’historique des conversations :**

Pour garder en mémoire les messages, nous avons utilisé une base de données sqlite qui stock l’historique dans un fichier local.

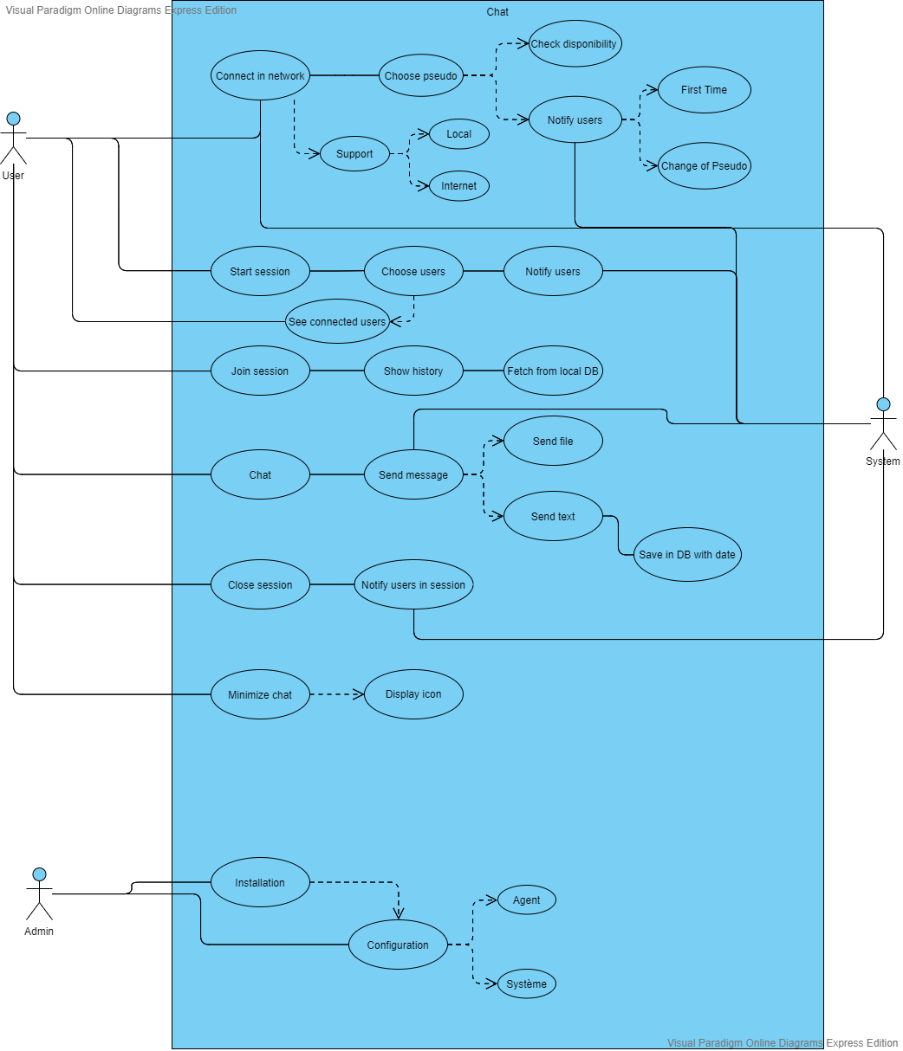
En effet, partant du principe que chaque utilisateur reste sur la même machine, il n’y a pas besoin de sauvegarder les communications auxquelles il n’a pas pris part. Ainsi, au démarrage d’une conversation, on appelle la classe responsable du fonctionnement de la base de données afin qu’elle soit créée si besoin. On récupère alors l’historique grâce à une succession de get qui nous rend la liste des messages. On peut alors les afficher sur l’interface.

A la fin de la communication, on rappelle la base de données avec la liste des messages reçues et envoyées ainsi que le pseudo associé. On peut alors tous les insérer un par un pour les utiliser plus tard.

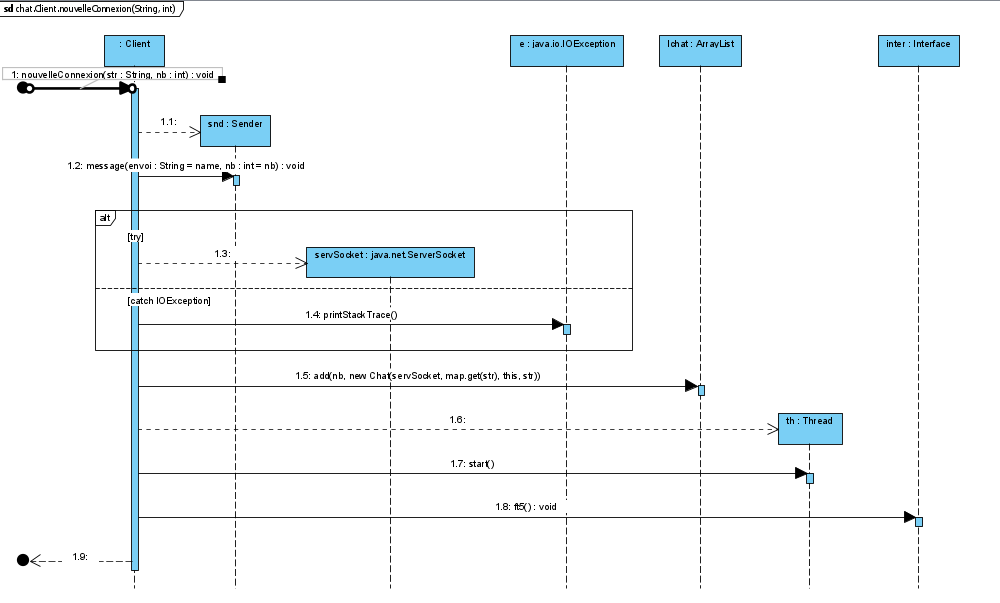


# **4) Projet UML :**

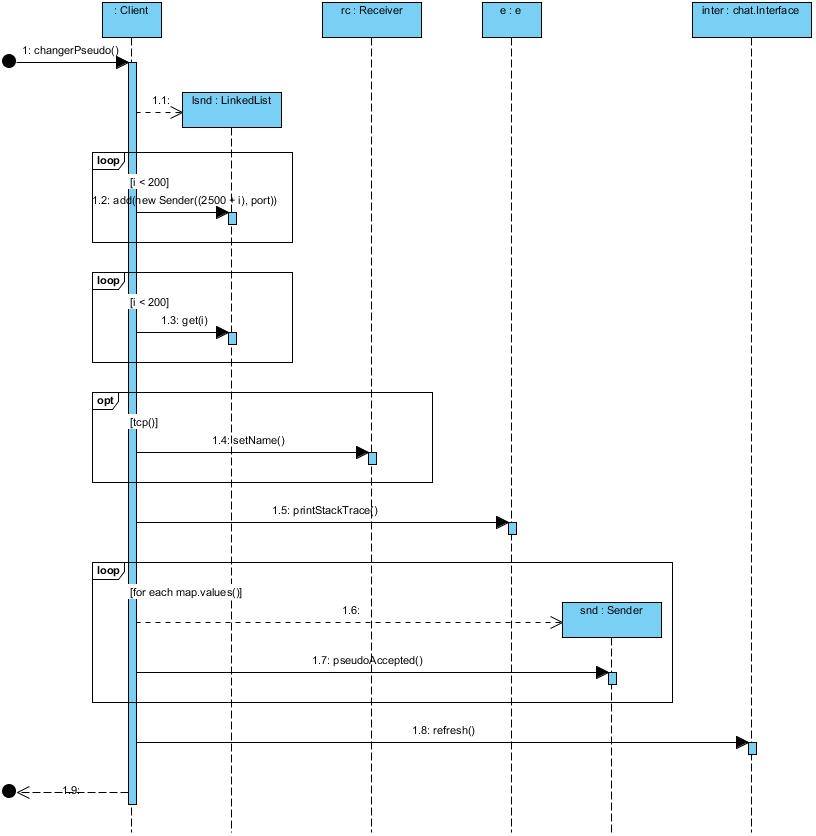
1. **Use case :**



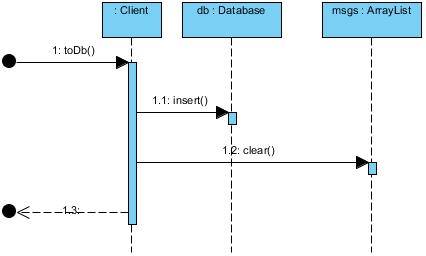
1. **Diagramme de séquences :** 
   * 1. **Connexion :**



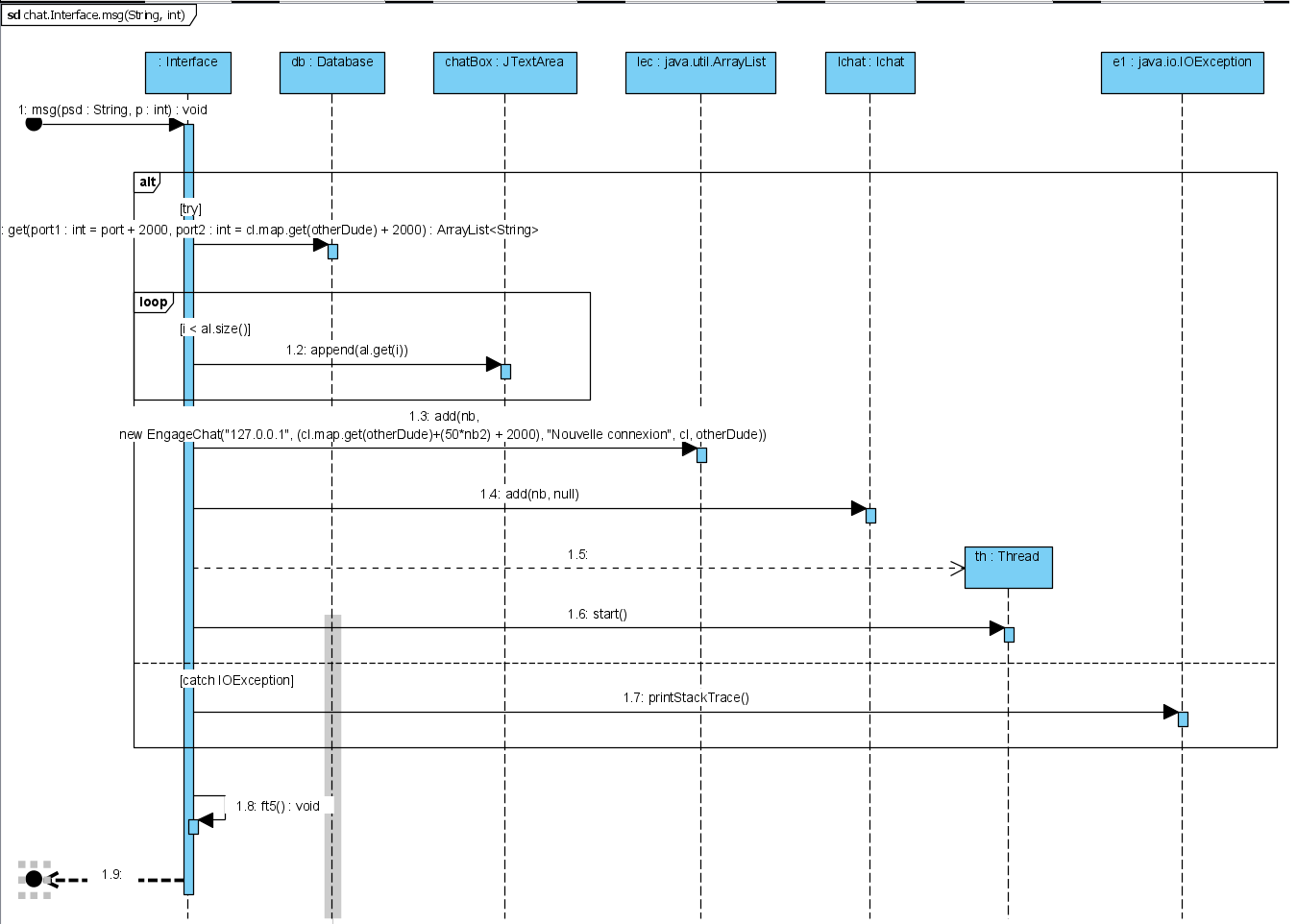
* + 1. **Changement de Pseudo :**



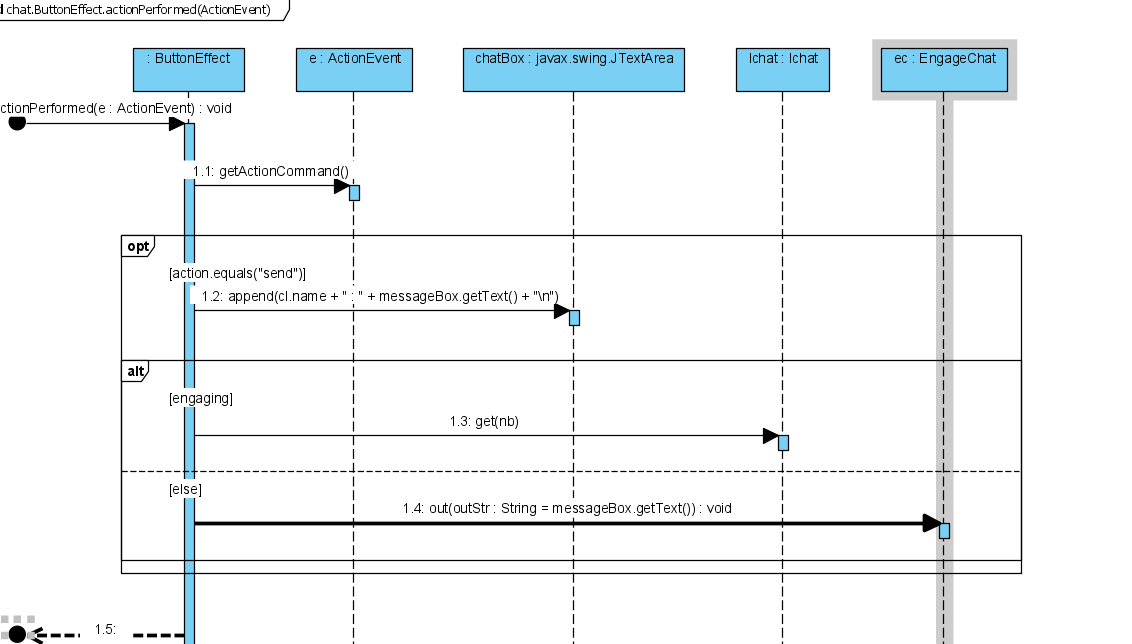
* + 1. **Sauvegarder dans la data base :**



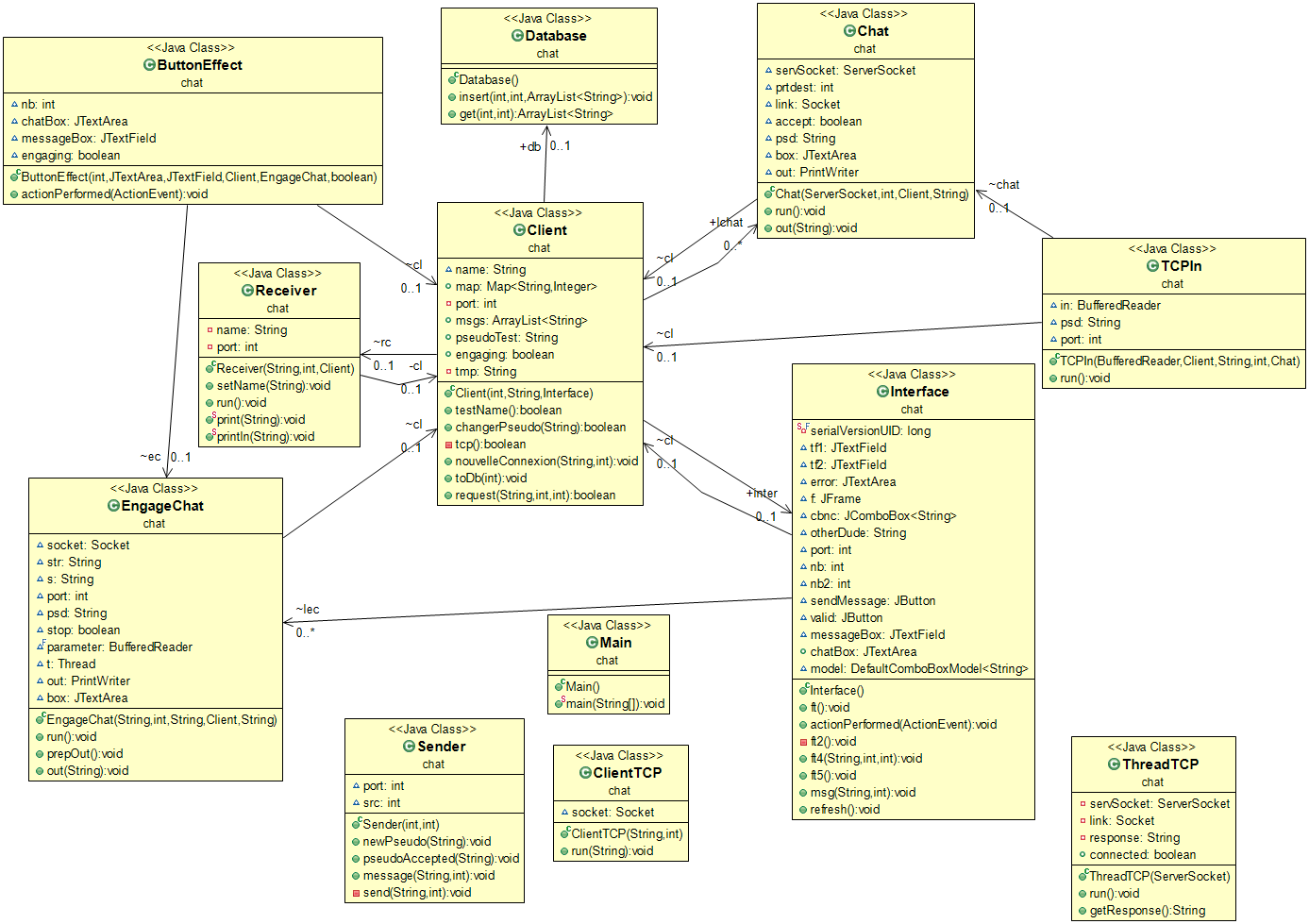
* + 1. **Demande de nouveau chat :**



* + 1. **Envoie d’un message :**



1. **Diagramme de classe :**



# **5) Conclusion :**

A la fin de ce projet, nous avons vraiment réussi à mettre en œuvre tout notre savoir-faire en Java, et les sujets et outils importants abordés cette année : programmation réseaux, Swing, Sqlite et également la manipulation de Git, ce dernier nous a permis de travailler à distance et de toujours mettre à jour notre code malgré cette période difficile.

La situation actuelle nous a beaucoup freiné par rapport aux tests qu’on pouvait faire (avoir accès à différentes machines des salles de tp par exemple), mais on a essayé de faire au mieux pour traiter les différents points possibles.