Rapport Projet Chat C#

Fabien Beaujean – Adrien Poupa

Efrei Software Engineering Promotion 2018

Sommaire

I. Choix techniques

a. Structure du projet

Dans un premier temps, nous avons analysé puis cherché à respecter scrupuleusement le sujet en utilisant ses diagrammes définissant les classes à utiliser : interfaces, packages, etc.

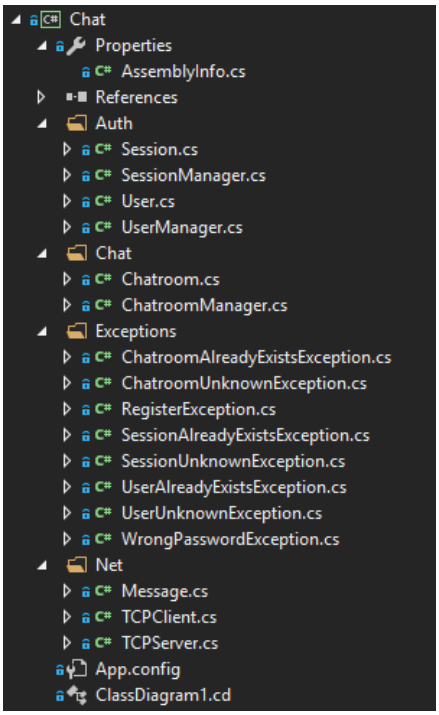
Cependant, alors que notre implémentation était finie, il ne nous est pas apparu cohérent de garder la structure telle quelle. Nous avons donc supprimé les interfaces dont ne nous voyions pas l'intérêt de ne les implémenter qu'une seule fois au détriment de classes plus simples et de classes abstraites.

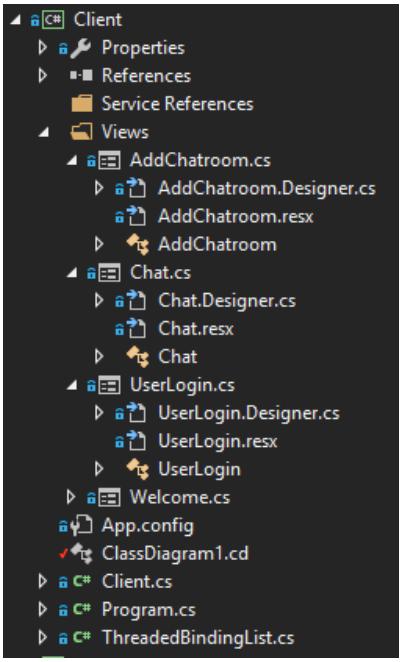
Nous avons cependant gardé la philosophie du projet originel : les fonctionnalités sont identiques, le fonctionnement général est resté le même.

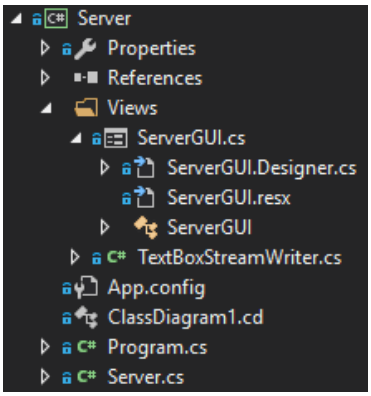
Nous avons commencé par développer un chat fonctionnant en ligne de commande.

Très tôt, nous avons compris qu'il nous faudrait séparer clairement le serveur et ses différents clients. C'est dans cet esprit que notre programme est séparé en trois projets distincts : le chat en lui-même qui ne peut être lancé, le Client qui contient l'interface utilisateur ainsi qu'une classe faisant le lien avec le projet Chat et un dernier projet similaire pour le Serveur. Les trois projets se situent dans la même solution, ce qui nous permet d'utiliser le code du projet Chat dans le Client et le Server. Pour lancer le programme, nous effectuons un démarrage multiple : un Server est lancé, ainsi qu'un Client.

Chaque projet contient différents dossiers, « packages » en Java. Ceux du projet Chat sont conformes au sujet, le dossier Views des autres projets permet de différencier le GUI.







b. Persistance des données

Nous avons choisi de stocker les données à conserver (liste des chatrooms et utilisateurs) dans un fichier binaire statique. Nous utilisons un BinaryFormatter pour enregistrer l'état actuel des classes UserManager et ChatroomManager dans un fichier. Nous aurions pu nous servir de mécanismes plus élaborés comme une base de données SQLite, mais nous pensions que cela serait plus difficile à ouvrir pour une personne n'ayant pas développé le projet sur nos machines (nécessité d'avoir installé un module spécifique).

La sauvegarde se fait comme ceci :

```
/// <summary>
/// Save the current list to a static file.
/// </summary>
/// <param name="path">Path to the file</param>
public void save(string path)
{
    try
    {
        using (Stream stream = File.Open(path, FileMode.Create))
        {
            BinaryFormatter bin = new BinaryFormatter();
            bin.Serialize(stream, UserList);
        }
    }
    catch (IOException e)
    {
        Console.WriteLine(e.Message);
    }
}
```

La récupération des fichiers est similaire :

```
/// <summary>
/// Load the users from a static file
/// </summary>
/// <param name="path">Path to the file</param>
public void load(string path)
{
    try
    {
        using (Stream stream = File.Open(path, FileMode.Open))
        {
            BinaryFormatter bin = new BinaryFormatter();
            List<User> users = (List<User>)bin.Deserialize(stream);
            userList = users;
        }
    }
    catch (IOException e)
    {
        Console.WriteLine(e.Message);
    }
}
```

c. Gestion des exceptions

Nous avons choisi de gérer les exceptions comme indiqué dans le sujet, adapté au C#. Nous avons donc créé une classe unique par exception à jeter.

Chaque classe, très simple, hérite de la classe de base System.Exception.

```
namespace Chat.Exceptions
{
    public class WrongPasswordException : System.Exception
    {
        public WrongPasswordException(string message) : base(message)
        {
        }

        public WrongPasswordException(string message, System.Exception innerException)
: base(message, innerException)
        {
        }
    }
}
```

d. Classe Session

Chaque utilisateur est identifié de façon unique grâce à sa Session associée. Chaque session contient le TcpClient de l'utilisateur pour accéder à son socket, son utilisateur ainsi qu'un identifiant unique permettant de le retrouver (Guid).

```
Guid token;
User user;
TcpClient client;
```

e. Classe Message

Conformément au sujet, le client et le serveur communiquent en s'échangeant des objets de classe Message. Cette classe contient un header, choisi dans une liste prédéfinie :

```
public enum Header { REGISTER, JOIN, QUIT, JOIN_CR, QUIT_CR, CREATE_CR, LIST_CR, POST,
LIST_USERS }
```

```
private Header head;
```

En plus du header, elle contient une liste de strings, qui contiennent le message et/ou les informations demandées par le serveur et le client.

```
private List<string> messageList;
```

f. Classe TCPClient

La classe TCPClient permet au Client, du projet Client, de se connecter au serveur.

Elle crée le TcpClient utilisé (attention à ne pas confondre TCPClient, notre classe, avec le type TcpClient natif en C#), obtenu à la connexion au serveur en utilisant son IP et son port.

Enfin, elle permet d'envoyer et de recevoir un message. Pour ce faire, on récupère le stream du TcpClient et on serialize ou deserialize l'objet Message que l'on veut envoyer ou recevoir.

```
        /// <summary>
        /// Get a message from the server.
        /// Deserialize the object message received and return it
        /// </summary>
        /// <returns>Message object received</returns>
        public Message getMessage()
        {
            try
            {
                NetworkStream strm = tcpClient.GetStream();
                IFormatter formatter = new BinaryFormatter();
                Message message = (Message)formatter.Deserialize(strm);
                Console.WriteLine("## TCPClient Receiving a message: " +
message.Head);
                return message;
            }
            catch (Exception e)
            {
                Console.WriteLine("TCPClient getMessage exception: " + e.Message);
                Console.WriteLine(e.Message);
            }

            return null;
        }

        /// <summary>
        /// Send a message to the server
        /// </summary>
        /// <param name="message">Message to send</param>
        public void sendMessage(Message message)
        {
            Console.WriteLine("## TCPClient Sending a message: " + message.Head);

            try
            {
                IFormatter formatter = new BinaryFormatter();
                NetworkStream strm = tcpClient.GetStream();
                formatter.Serialize(strm, message);
            }
            catch (SerializationException e)
            {
                Console.WriteLine("TCPClient sendMessage exception: "+e.Message);
            }
        }
```

g. Classe TCPServer

Cette classe est similaire à TCPClient, bien qu'un peu plus complexe. Ainsi, le TCPClient est remplacé par un TcpListener et trois threads font leur apparition ; nous en parlerons dans la classe Server.

On peut démarrer, arrêter le serveur mais aussi, comme précédemment, envoyer et recevoir un message.

```
/// <summary>
/// Get a message object from a given client
/// </summary>
/// <param name="socket">Client to listen to</param>
/// <returns>Deserialized Message object</returns>
public Message getMessage(Socket socket)
{
    Console.WriteLine("## TCPServer Receiving a message");

    try
    {
        NetworkStream strm = new NetworkStream(socket);
        IFormatter formatter = new BinaryFormatter();
        Message message = (Message)formatter.Deserialize(strm);
        Console.WriteLine("- message header: " + message.Head);
        return message;
    }
    catch(Exception e)
    {
        Console.WriteLine("TCPServer getMessage exception: " + e.Message);
        Console.WriteLine(e.Message);
    }

    return null;
}

/// <summary>
/// Send a message to a given client
/// </summary>
/// <param name="message">Message to send</param>
/// <param name="socket">Client to send the message to</param>
public void sendMessage(Message message, Socket socket)
{
    Console.WriteLine("## TCPServer Sending a message: " + message.Head);

    try
    {
        IFormatter formatter = new BinaryFormatter();
        NetworkStream strm = new NetworkStream(socket);
        formatter.Serialize(strm, message);
    }
    catch (Exception e)
    {
        Console.WriteLine("TCPServer sendMessage exception: " + e.Message);
        Console.WriteLine(e.Message);
    }
}
```

Bien que les fonctions soient similaires, il ne nous a pas semblé cohérent de créer une interface commune à TCPClient et TCPServer puisque leurs signatures diffèrent. Le serveur doit savoir à quel socket (client) envoyer le message, ce qui n'est pas nécessaire côté client où on parle au serveur.

h. Classe Client

Cette classe hérite de TCPClient, expliquée plus haut. Elle contient toute la logique du client, et est utilisée dans les WinForms du client.

Son rôle principal est de lancer deux threads, checkConnection et checkQuit qui, comme leurs noms l'indiquent, s'occupent de vérifier l'arrivée de nouveaux messages depuis le serveur et de vérifier si le serveur est toujours en ligne. Tant que le client a sa fenêtre ouverte et que le serveur est en ligne, ils continuent à tourner.

```
        /// <summary>
        /// Launch what we need for the client
        /// First a thread to check the connection and process the data sent by the
server
        /// Then a thread to quit if needed
        /// </summary>
        public void run()
        {
            Thread checkConnection = new Thread(new ThreadStart(this.checkData));
            checkConnection.Start();

            Thread checkQuit = new Thread(new ThreadStart(this.checkQuit));
            checkQuit.Start();
        }

        /// <summary>
        /// Check if we have messages incoming from the server
        /// </summary>
        public void checkData()
        {
            while (!Quit)
            {
                try
                {
                    if (tcpClient.GetStream().DataAvailable)
                    {
                        Thread.Sleep(25);
                        Message message = getMessage();

                        if (message != null)
                        {
                            // We have a message: call to processData
                            Thread processData = new Thread(() =>
this.processData(message));
                            processData.Start();
                        }
                    }
                }
                catch (InvalidOperationException e)
                {
                    Console.WriteLine(e.Message);
                }

                Thread.Sleep(5);
            }
        }
```

```
/// <summary>
/// Do what needs to be done if server is disconnected
/// </summary>
private void checkQuit()
{
    while (!Quit)
    {
        Socket socket = tcpClient.Client;

        if (socket.Poll(10, SelectMode.SelectRead) && socket.Available == 0)
        {
            Quit = true;
            Console.WriteLine("- Client checkQuit: Server disconnected");
        }

        Thread.Sleep(5);
    }
}
```

Pour vérifier si un message est disponible, on utilise getMessage définie dans la classe mère, TCPClient. Si on a bel et bien un message, une autre fonction de la classe, processData, s'occupe du traitement approprié.

```
/// <summary>
/// Deal with the message received
/// </summary>
/// <param name="message"></param>
private void processData(Message message)
{
    switch (message.Head)
    {
        case Message.Header.REGISTER:
            if(message.MessageList[0] == "success")
            {
                Console.WriteLine("- Registration success: " + User.Login);
            }
            else
            {
                Console.WriteLine("- Registration failed: " + User.Login);
            }
            break;
            // etc…
```

i. Classe ThreadedBindingList

Pour afficher des listes, le Client utilise des ThreadedBindingList. Il s'agit d'une classe implémentant des BindingLists pouvant être utilisées dans un thread.

Nous avons choisi de travailler avec des BindingLists car il nous semblait indispensable d'utiliser des listes permettant le « 2-way binding », où une mise à jour de la liste depuis la classe Client serait instantanément vue par le WinForm sans avoir besoin d'une boucle dans celui-ci qui effacerait la liste actuelle toutes les X secondes avant de la remplacer par une nouvelle, ce qui nous semblait être un non-sens quand des composants adaptés existent.

Dans le client, ces listes sont utilisées pour stocker les utilisateurs connectés à la chatroom actuelle, les chatrooms disponibles, ainsi que les messages reçus du serveur.

j. Classe Chat

Il s'agit de la vue principale du client. D'autres WinForms existent (connexion, identification) mais ils ne présentent pas un grand intérêt.

Trois threads sont utilisés pour :

- Récupérer périodiquement la liste de chatrooms disponibles en demandant au client d'envoyer un message LIST_CR
- Récupérer périodiquement la liste d'utilisateurs connectés à la chatroom actuelle en demandant au client d'envoyer un message LIST_USERS
- Vérifier périodiquement que le serveur est toujours connecté avec le booléen client.Quit

Le reste des fonctions sert à coder les comportements des boutons : clic sur le bouton pour ajouter une chatroom, gestion de la combobox pour choisir la chatroom, clic d'envoi de message.

Les threads ont un timeout de 2 secondes dans leur boucle pour éviter de surcharger le serveur inutilement, s'agissant de données qu'il n'est pas absolument indispensable d'avoir en temps réel.

Comme dans tous les WinForms du projet, il a fallu faire attention d'invoquer le thread du WinForm principal pour modifier ses éléments puisque ces derniers ne sont pas modifiables depuis un thread externe.

k. Classe Server

Son fonctionnement est similaire à la classe Client expliquée ci-dessus. Elle permet de gérer le serveur en héritant de TCPServer.

Tout d'abord, dans son constructeur, on instancie tous les Manager (User, Chatroom et Session) et on récupère les données stockées dans le fichier statique puis on lance trois threads pour :

- Écouter si on a des nouvelles connexions pour créer leur socket avec AcceptTcpClient et leur session
- Vérifier périodiquement si de nouveaux messages arrivent de la part de clients
- Vérifier périodiquement si des clients ont quitté le serveur

```
        /// <summary>
        /// Running the server launches 3 threads:
        /// Check for the messages sent by users
        /// Check if a client has left
        /// Create a new TcpClient object if a new client joins
        /// </summary>
        public void run()
        {
            checkDataThread = new Thread(new ThreadStart(this.checkData));
            checkDataThread.Start();

            checkQuitThread = new Thread(new ThreadStart(this.checkQuit));
            checkQuitThread.Start();

            listenerThread = new Thread(new ThreadStart(this.listen));
            listenerThread.Start();
        }

        /// <summary>
        /// Listen for new clients and create a new Session for each new client with
its own TcpClient instance
        /// Since AcceptTcpClient is blocking, this is run in a thread
        /// </summary>
        private void listen()
        {
            while (this.Running)
            {
                try
                {
                    Console.WriteLine("Waiting for a new connection...");
                    TcpClient client = this.tcpListener.AcceptTcpClient();
                    Session session = new Session();
                    session.Client = client;
                    SessionManager.addSession(session);

                    Console.WriteLine("New client: " + session.Token);
                }
                catch (SocketException)
                {
                    // Here we catch a WSACancelBlockingCall exception because
this.tcpListener is probably closed
                    Console.WriteLine("Listener thread closed");
                }
            }
        }

        /// <summary>
        /// Check data coming from clients
        /// </summary>
        private void checkData()
        {
            while (this.Running)
            {
                try
                {
                    lock (readLock)
                    {
                        if (SessionManager.SessionList.Count > 0)
                        {
                            foreach (Session session in
SessionManager.SessionList.ToList())
                            {
```

```
                                if (session != null &&
session.Client.GetStream().DataAvailable)
                                {
                                    Thread.Sleep(25);
                                    Message message =
getMessage(session.Client.Client);

                                    if (message != null)
                                    {
                                        // We have data to process: call to the
appropriate function
                                        Thread processData = new Thread(() =>
this.processData(session, message));
                                        processData.Start();
                                    }
                                }
                            }
                        }
                    }
                    catch(InvalidOperationException e)
                    {
                        Console.WriteLine(e.Message);
                    }

                    Thread.Sleep(5);
                }
            }

            /// <summary>
            /// Check if a client has left
            /// </summary>
            private void checkQuit()
            {
                while (this.Running)
                {
                    for (int i = 0; i < SessionManager.SessionList.Count; i++)
                    {
                        Socket socket = SessionManager.SessionList[i].Client.Client;

                        if (socket.Poll(10, SelectMode.SelectRead) && socket.Available ==
0)
                        {
                            Console.WriteLine("- User logged out : " +
SessionManager.SessionList[i].Token);

                            lock(readLock)
                            {
                                if (SessionManager.SessionList[i].User != null &&
                                    SessionManager.SessionList[i].User.Chatroom != null)
                                {
                                    // Tell the other users that he left
                                    broadcastToChatRoom(SessionManager.SessionList[i],
"left the chatroom \"" +
                                        SessionManager.SessionList[i].User.Chatroom.Name +
"\"");
                                }

                                SessionManager.SessionList[i].Client.Close();

sessionManager.removeSession(SessionManager.SessionList[i].Token);
                            }
```

```
                    }
                }

                Thread.Sleep(5);
            }
        }

        /// <summary>
        /// Check if a client has left a chatroom
        /// </summary>
        /// <param name="session"></param>
        /// <param name="message"></param>
        private void quitCr(Session session, Message message)
        {
            try
            {
                if (session.User.Chatroom != null)
                {
                    // Warn the user he left the chatroom
                    Message messageSuccess = new Message(Message.Header.QUIT_CR);
                    messageSuccess.addData("success");
                    messageSuccess.addData(session.User.Chatroom.Name);
                    sendMessage(messageSuccess, session.Client.Client);

                    // Warn the other users that this one left
                    broadcastToChatRoom(session, "left the chatroom \"" +
session.User.Chatroom.Name + "\"");

                    Console.WriteLine("- " + session.User.Login + " left the chatroom:
" + session.User.Chatroom.Name);

                    session.User.Chatroom = null;
                }
            }
            catch (ChatroomUnknownException e)
            {
                // Warn the user the chatroom does not exist
                Message messageError = new Message(Message.Header.QUIT_CR);
                messageError.addData("error");
                messageError.addData(message.MessageList[0]);
                sendMessage(messageError, session.Client.Client);

                messageError.addData("Chatroom " + e.Message + " does not exist");
            }
        }

        /// <summary>
        /// Process data sent by clients
        /// </summary>
        /// <param name="session">Session from where the message comes from</param>
        /// <param name="message">Message received</param>
        private void processData(Session session, Message message)
        {
            if (session.User != null)
            {
                switch (message.Head)
                {
                    case Message.Header.QUIT:
                        {
                            // Warn the user he has been disconnected
                            Message messageSuccess = new Message(Message.Header.QUIT);
                            messageSuccess.addData("success");
```

```
                            sendMessage(messageSuccess, session.Client.Client);

                            if(session.User.Chatroom != null)
                            {
                                // Warn the other users that he left
                                broadcastToChatRoom(session, "left the chatroom \"" +
session.User.Chatroom.Name + "\"");
                            }

                            session.Client.Close();
                            sessionManager.removeSession(session.Token);

                            Console.WriteLine("- User logout: " + session.Token);
                        }
                        break;
                        // etc
```

La fonction processData qui répond aux clients leur envoie directement des messages avec le sendMessage de TCPServer ou envoie un message à tous les clients connectés à une même chatroom grâce à la fonction broadcastToChatroom :

```
        /// <summary>
        /// Function used to send a message to all users in a chatroom
        /// </summary>
        /// <param name="session"></param>
        /// <param name="message"></param>
        private void broadcastToChatRoom(Session session, string message)
        {
            Chatroom chatroom = session.User.Chatroom;

            if(chatroom != null && message != "")
            {
                Message messageJoin = new Message(Message.Header.POST);
                messageJoin.addData(session.User.Login);
                messageJoin.addData(session.User.Login+": " + message);

                foreach(Session sessionUser in SessionManager.SessionList.ToList())
                {
                    if(sessionUser.User.Chatroom != null &&
                        sessionUser.User.Chatroom.Name == chatroom.Name)
                    {
                        sendMessage(messageJoin, sessionUser.Client.Client);
                    }
                }

                Console.WriteLine("- " + session.User.Login + "'s message broadcast");
            }
            else
            {
                Console.WriteLine("- User is not connected to any chatroom: " +
session.User.Login);
            }
        }
```

On utilise donc les sessions stockées dans le SessionManager pour déterminer à quels utilisateurs on doit appliquer un sendMessage.

l. Classes ServerGUI et TextBoxStreamWriter

Enfin, le projet Server est le plus simple du chat. Il ne s'agit de que paramétrer le port sur lequel on veut démarrer le serveur, et de lancer ou d'arrêter le serveur avec le même bouton. Les messages du serveur sont affichés dans une textbox. Nous avons utilisé la classe TextBoxStreamWriter pour rediriger le texte envoyé vers la console dans un WinForm. Il fallait faire attention à ce que ce texte soit envoyé correctement, puisqu'il pouvait être envoyé depuis différents threads.

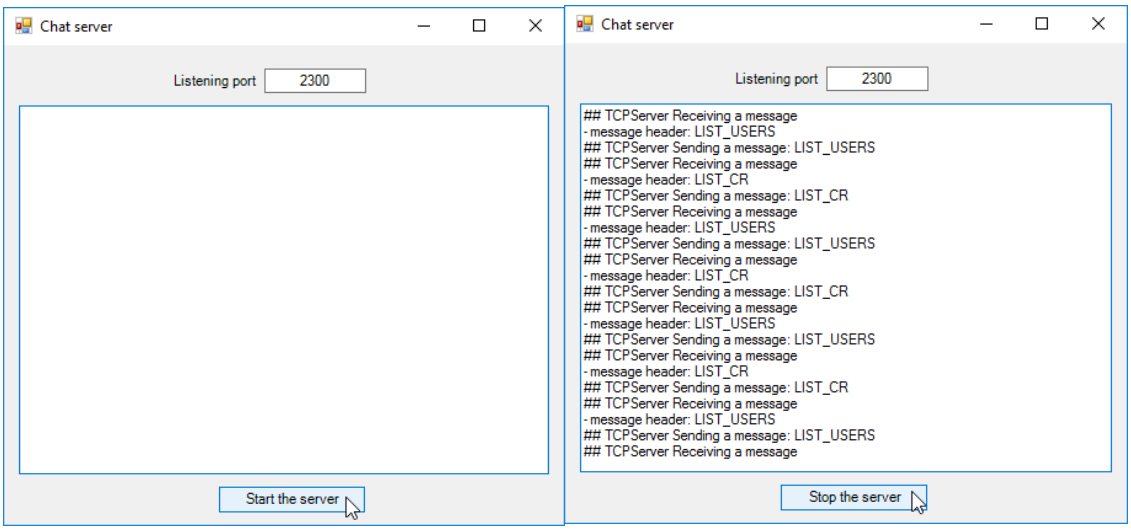
m. Thread-safe

Nous avons fait de notre mieux pour que notre programme soit thread-safe, en utilisant des lock et des mots-clé volatile lorsque c'était nécessaire.

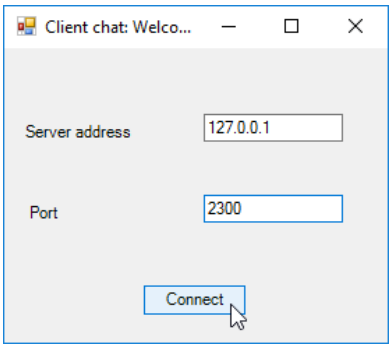
```
public volatile Object readLock;

readLock = new Object();

private void checkData()
{
   while (this.Running)
   {
      try
      {
         lock (readLock)
         {
```

II. Utilisation et impressions d'écran

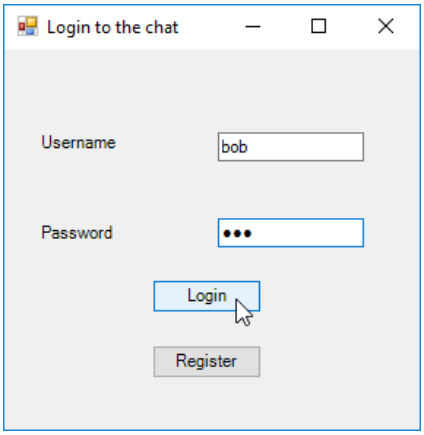
Lancement du serveur :



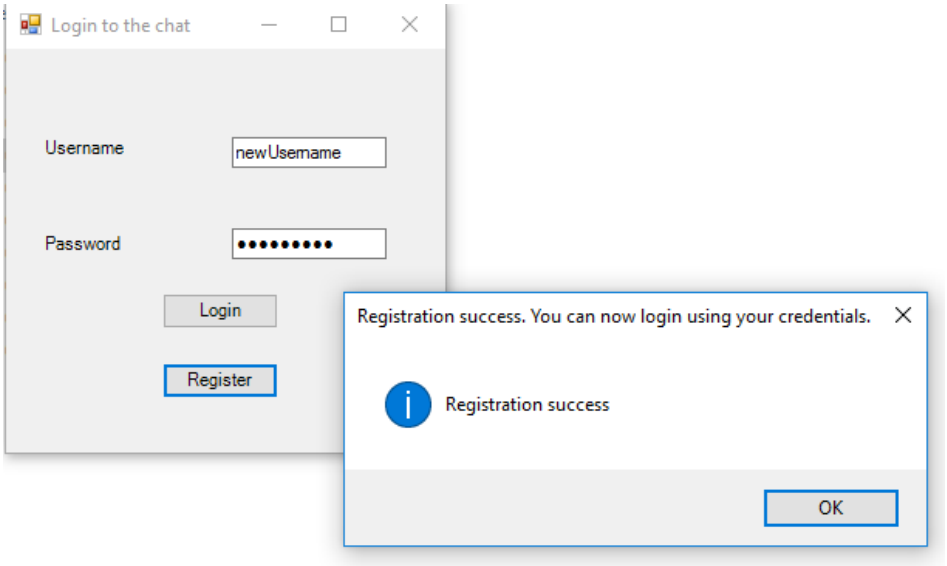
Lancement du client :



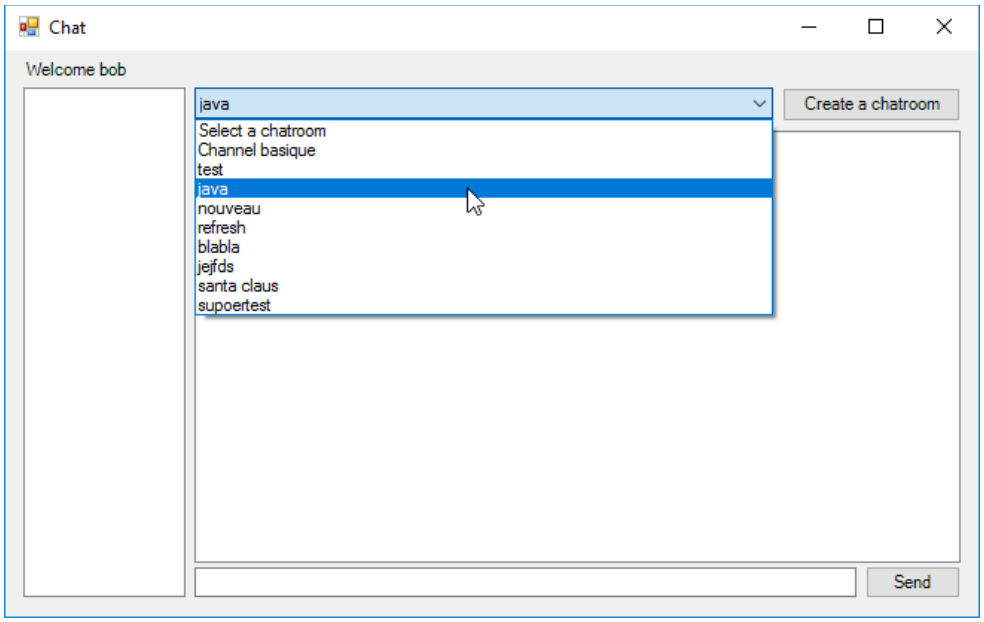
Connexion avec l'utilisateur « bob », mot de passe « 123 » :

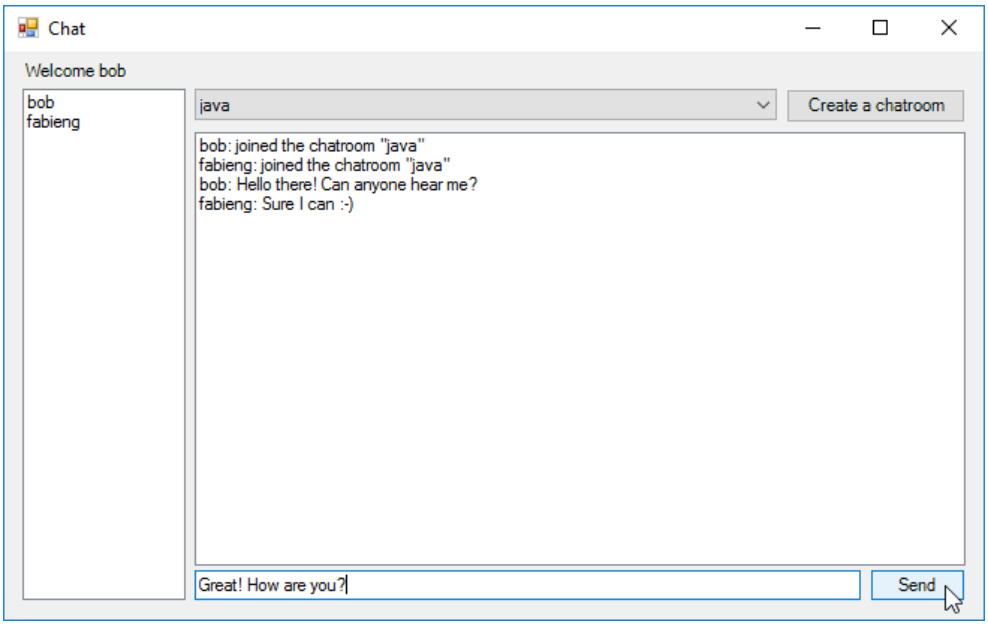


Création d'un nouvel utilisateur :

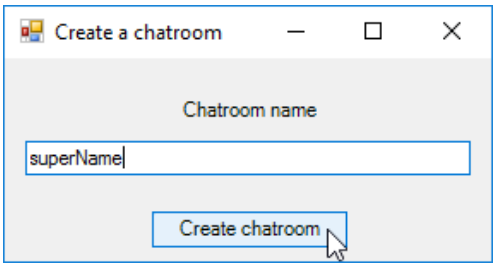


Sélection d'une chatroom :



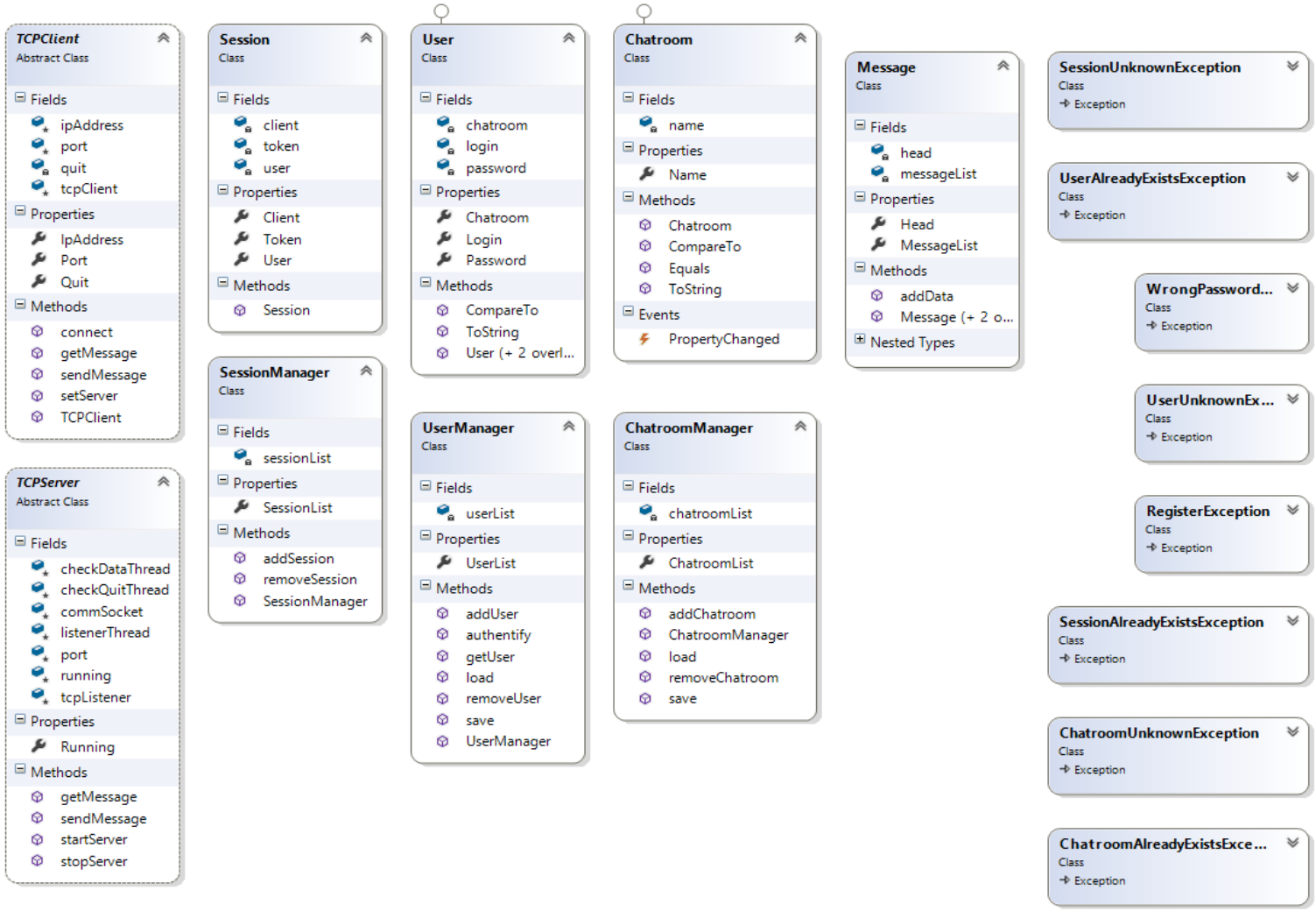


Création d'une chatroom :

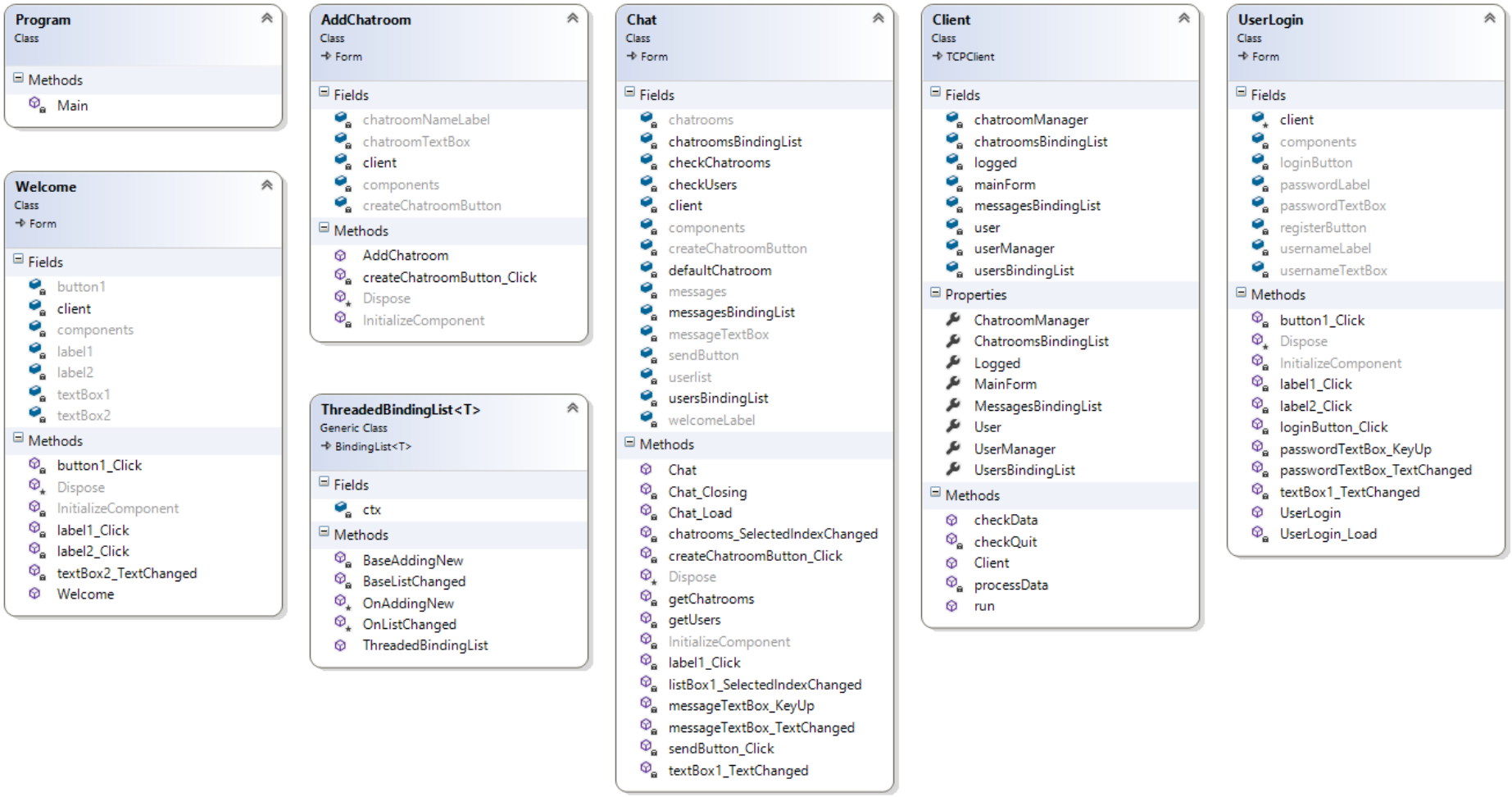


III. Diagrammes de classes

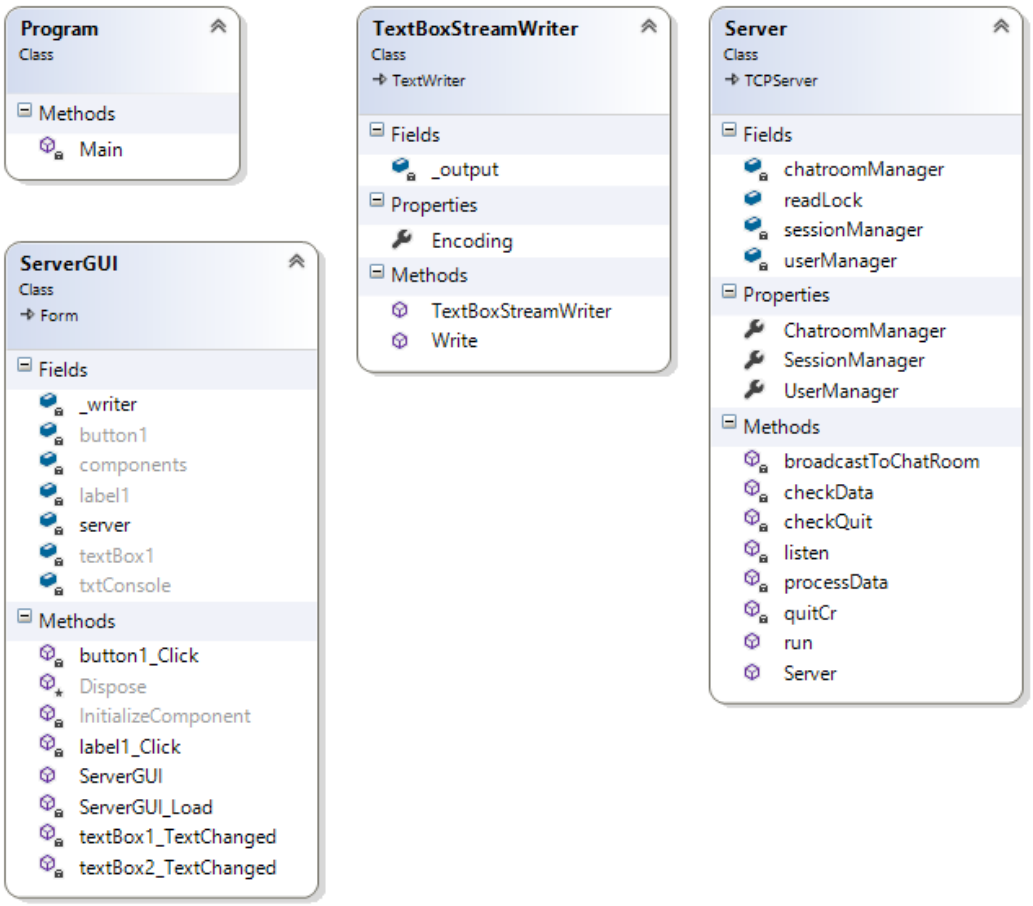
a. Projet « Chat »



b. Projet « Client »

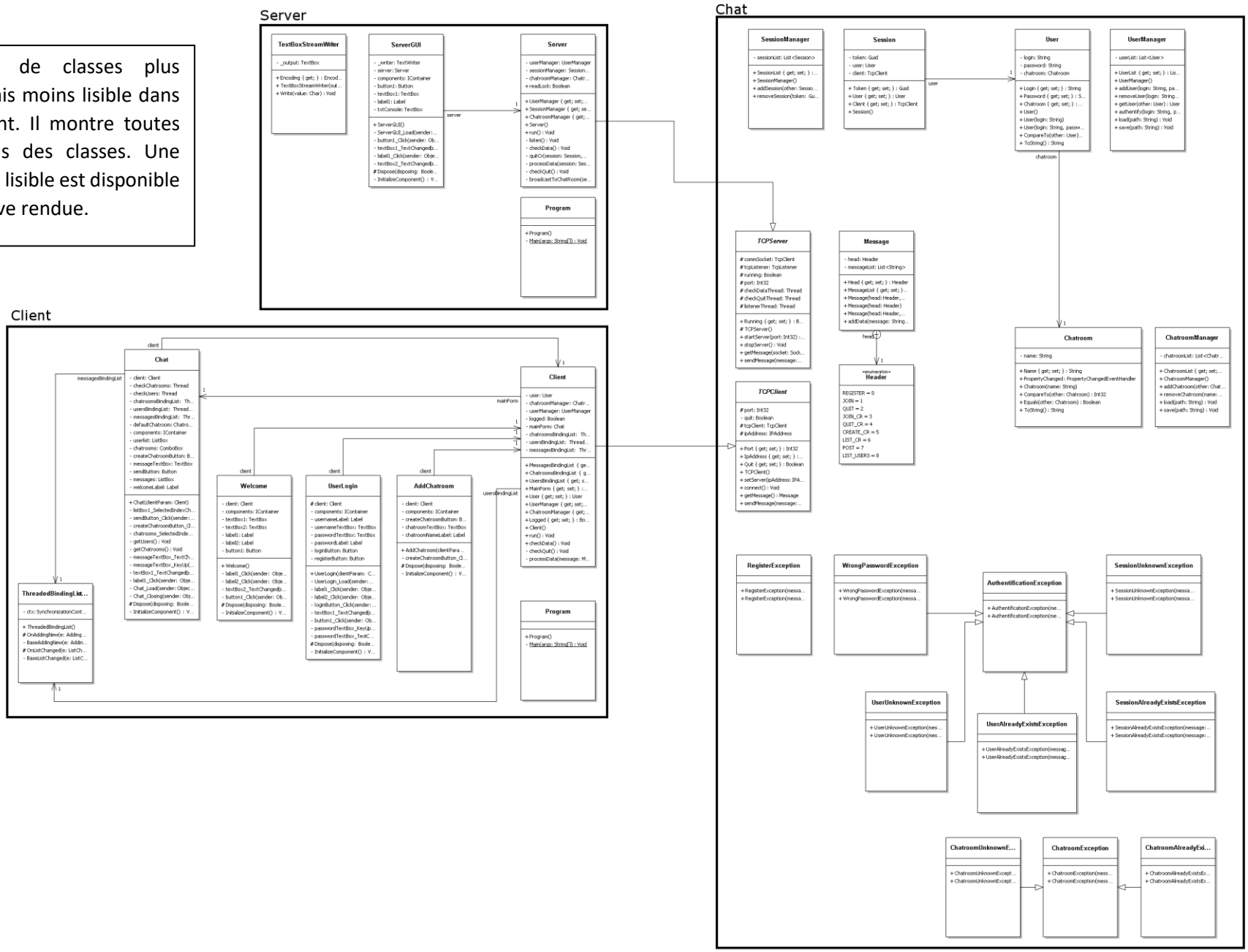


c. Projet « Server »



d. Diagramme de classes global

Diagramme de classes plus complet mais moins lisible dans ce document. Il montre toutes les relations des classes. Une version plus lisible est disponible dans l'archive rendue.



IV. Conclusion

Nous avons apprécié de travailler sur ce projet. Nous nous sommes toutefois permis de prendre quelques libertés par rapport au sujet, ce dernier ne nous semblant pas totalement adapté à notre projet : trop d'interfaces dont nous ne voyions pas l'utilité et de code pensé pour Java. À titre d'exemple, il est impossible de cloner une session d'une classe qui a un socket pour attribut. Nous avons aussi dû passer par des TcpClient et TcpListener au lieu des simples sockets préconisés. Nous avons rencontré des problèmes pour envoyer directement un objet sur le réseau en le sérialisant mais nous y sommes finalement parvenus.

Contrairement à ce que nous pensions au démarrage du projet, la création d'une interface graphique somme toute rudimentaire utilisant les WinForms a pris autant de temps que le développement du chat en ligne de commande. En effet, des problématiques spécifiques aux WinForms nous ont bloqué durant plus de temps que nous ne le pensions : invocation du thread de l'interface pour manipuler ses éléments, recherche d'un composant adapté pour une mise à jour aisée en 2-way binding.

Si notre projet nous semble tout à fait fonctionnel, il reste bien évidemment des pistes d'amélioration : nous pourrions stocker les mots de passe de façon sécurisée dans une vraie base de données, créer un espace de gestion des utilisateurs (ajout, suppression, modification) pour des utilisateurs administrateurs, créer une vraie gestion des chatrooms, ou encore créer un système de notification chaque fois que notre nom est cité, ainsi que la possibilité de se connecter à plusieurs chatrooms… on s'approcherait ainsi d'un système IRC.