

Réalisation d'un utilitaire de chat

1. Introduction

Durant les TP de java, vous allez implémenter en Java un modèle objet décrit en UML. Ce modèle est celui d'un utilitaire de chat. Dans la suite de ce document, on décrit les modèles produits durant la phase d'analyse du problème. L'analyse a été décomposée en plusieurs étapes reproduites ci-dessous :

- analyse de l'application,
- analyse du domaine,
- confrontation des résultats.

1.1. Analyse de l'application

Le diagramme des cas d'utilisation suivant montre l'application à réaliser.

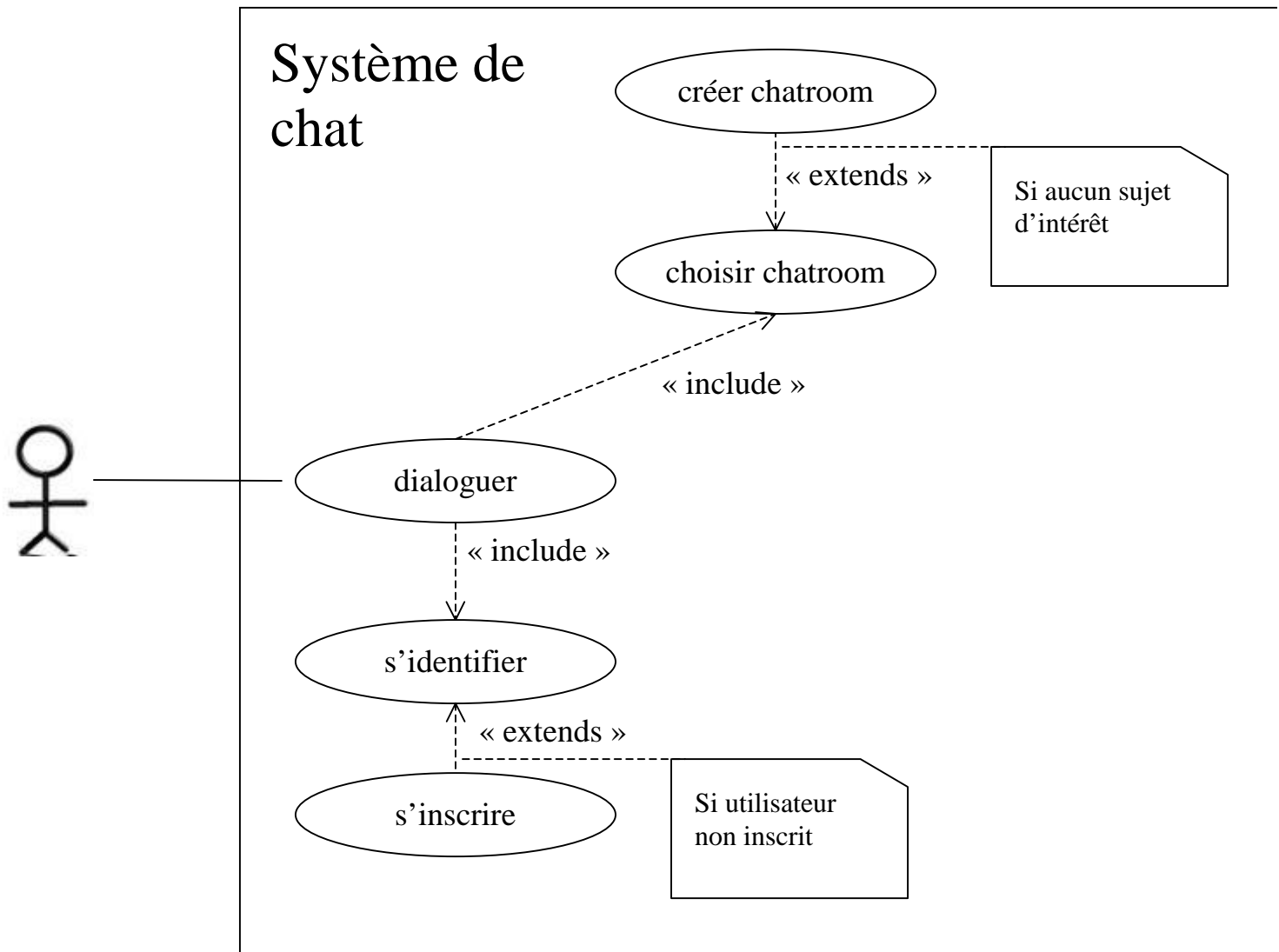


Figure 1 : diagramme des cas d'utilisation du système de Chat

L'application à réaliser est décrite par le diagramme d'activité suivant.

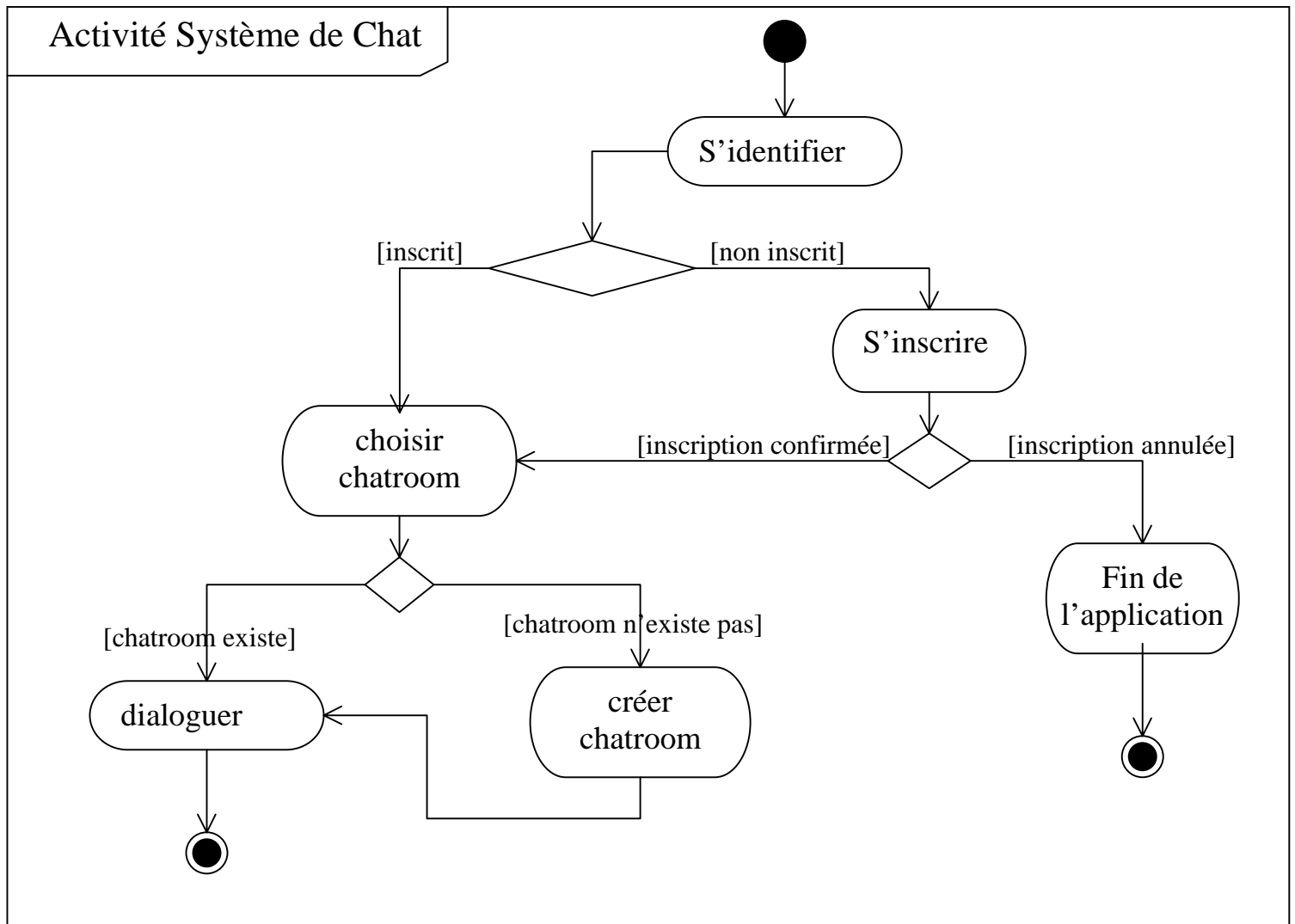


Figure 2 : diagramme d'activité du système de Chat

1.2. Analyse du domaine

Le diagramme des classes suivant décrit le domaine de l'application.

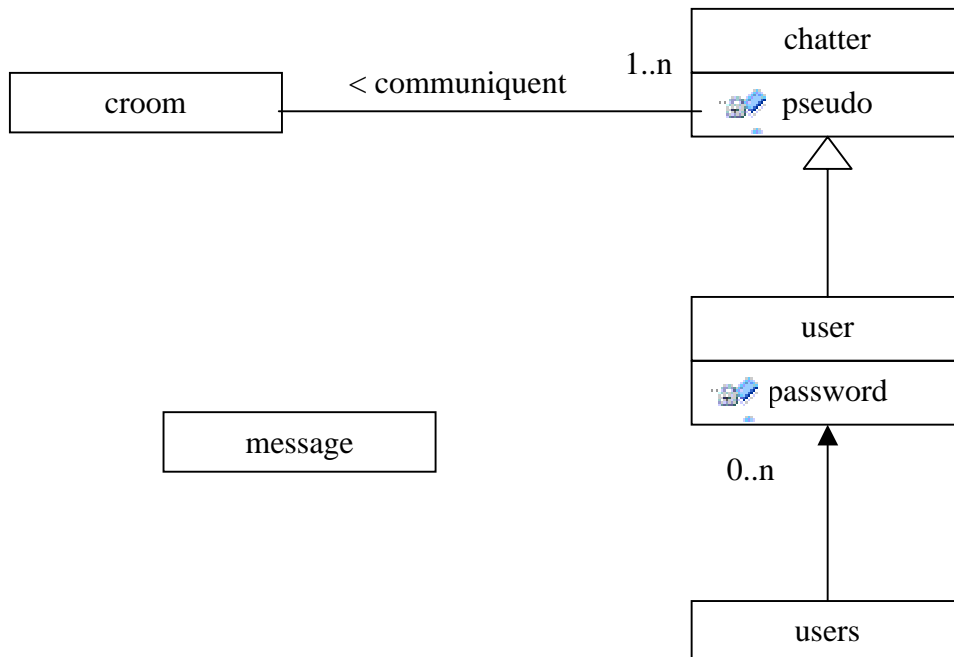


Figure 3 : diagramme des classes du domaine

1.3. Confrontation des résultats

Dans ce paragraphe, on montre comment les classes du domaine sont utilisées par l'application décrite au paragraphe 1.1. Le diagramme de séquence suivant reprend le diagramme d'activité de la figure 2, et sert de base de travail pour confronter les résultats.

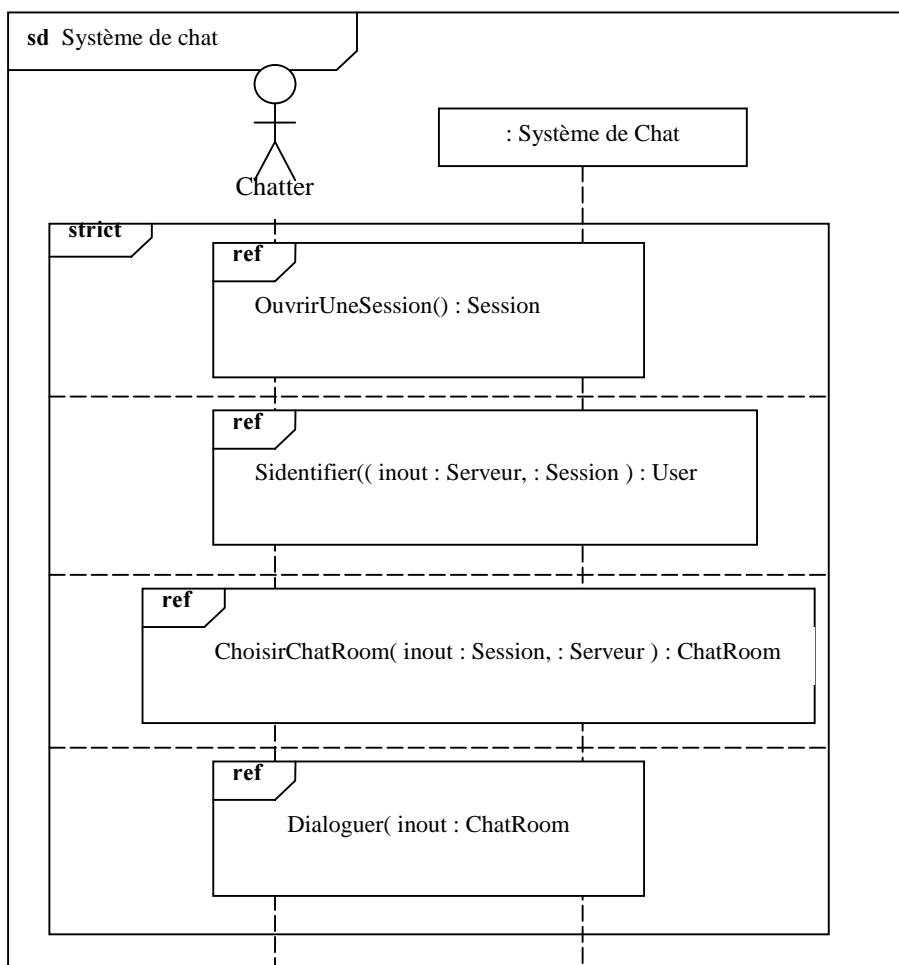


Figure 4 : diagramme de séquences illustrant le fonctionnement de l'application

Le lien entre l'analyse du domaine et l'analyse de l'application est réalisé par l'ajout de classes « contrôleurs », et ceci afin de respecter le modèle MVC (Model View Controler) largement utilisé en développement logiciel orienté objet.

Les diagrammes de séquences suivants détaillent les diagrammes qui sont référencés sur la figure 4.

La figure suivante détaille l'ouverture d'une session, c'est à dire le moment où un Chatter se connecte au système de chat. On y voit deux nouvelles classes « contrôleurs » (Server et Session) qui s'intègrent dans le diagramme des classes comme le montre la figure 6.

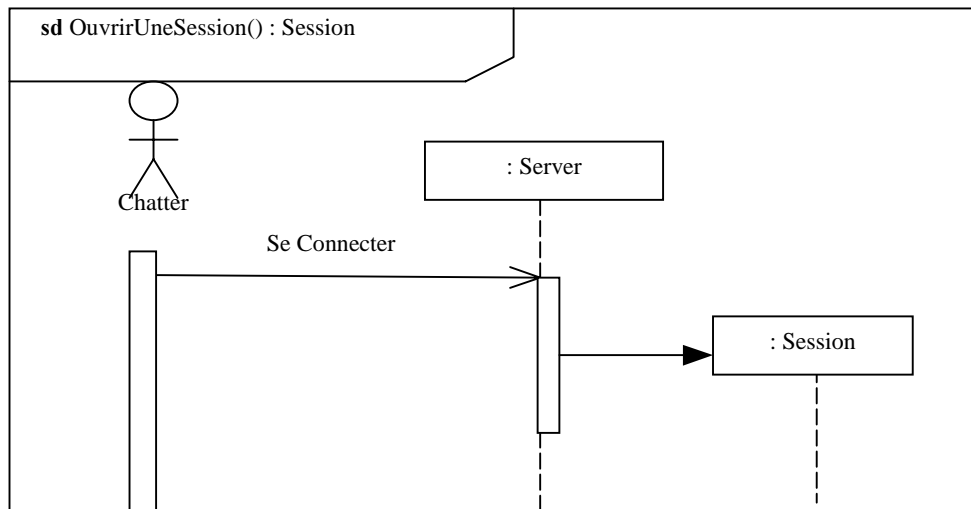


Figure 5 : l'ouverture d'une session

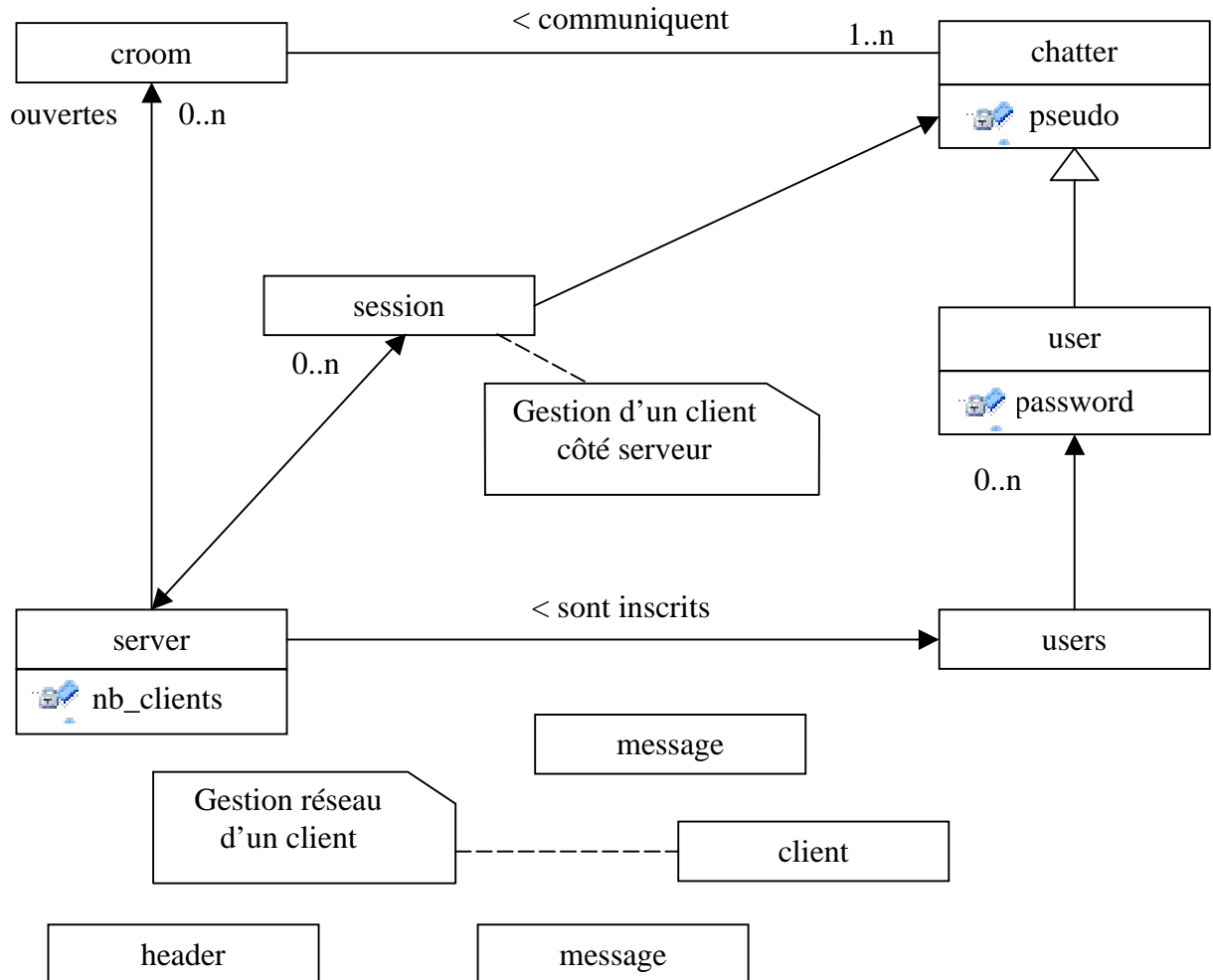


Figure 6 : diagramme des classes complété avec les classes contrôleurs

Les figures 7 et 8 décrivent de façon détaillées des diagrammes qui sont simplement référencés sur la figure 4.

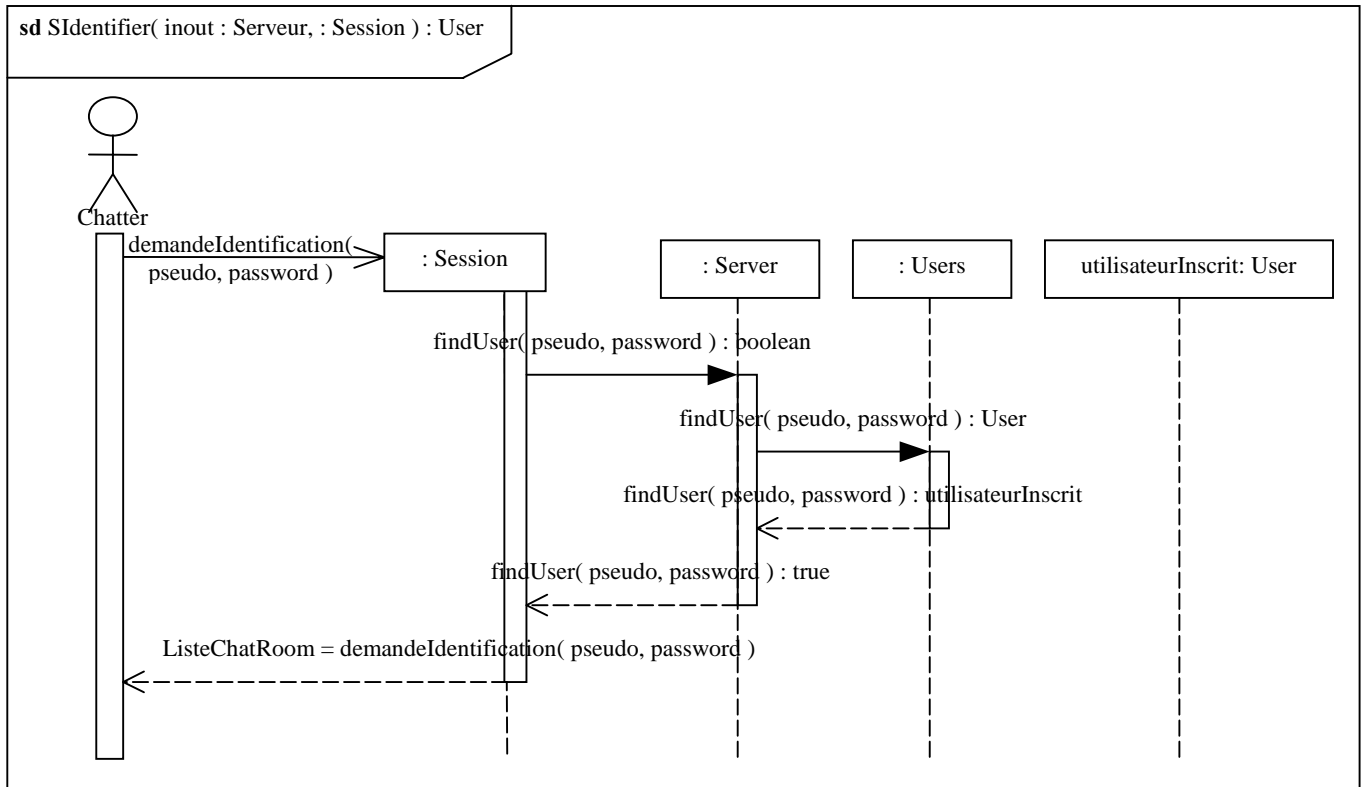


Figure 7 : l'identification du Chatter

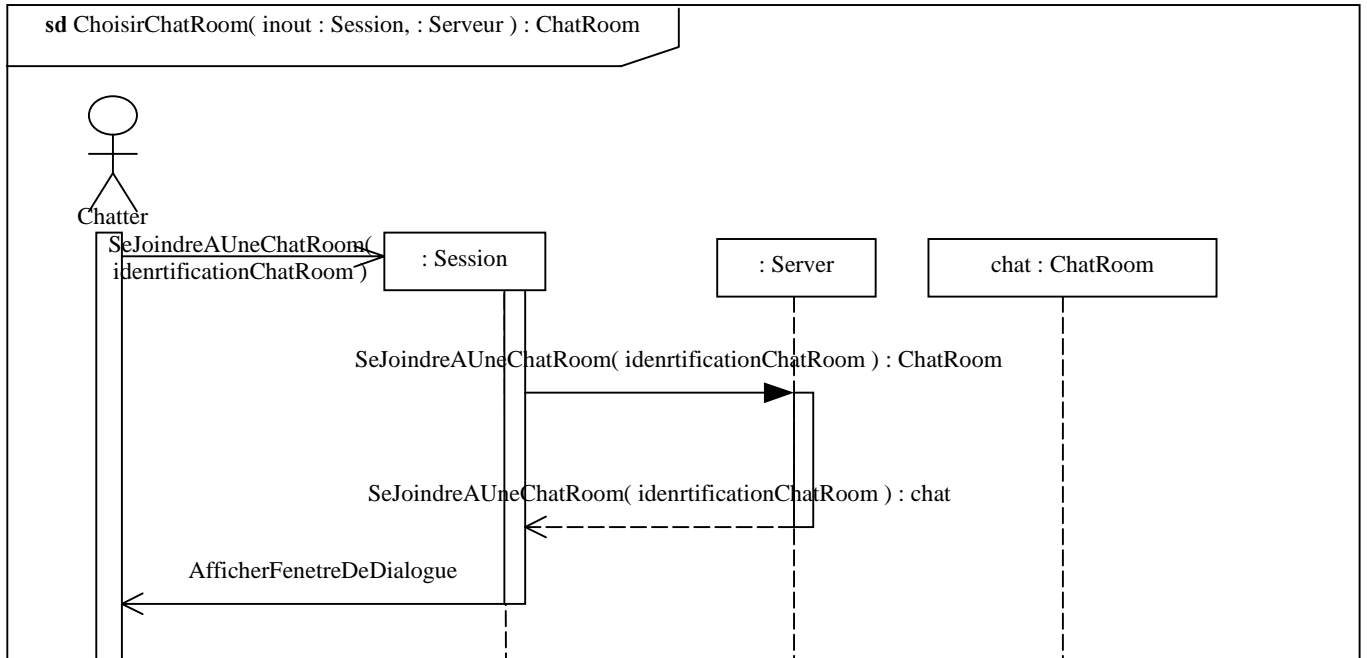


Figure 8 : le choix de la ChatRoom

Pour illustrer le diagramme référencé sous l'appellation « Dialoguer » à la figure 4, on utilise un diagramme d'objets (représenté ci-dessous). On y voit la transmission d'un message de données (du texte à transmettre) depuis un client vers sa session, répercuté vers le serveur. Le serveur recherche la chatroom où est inscrit l'émetteur du message, et redirige le texte vers tous les chatters inscrits à cette chatroom, l'émetteur y compris.

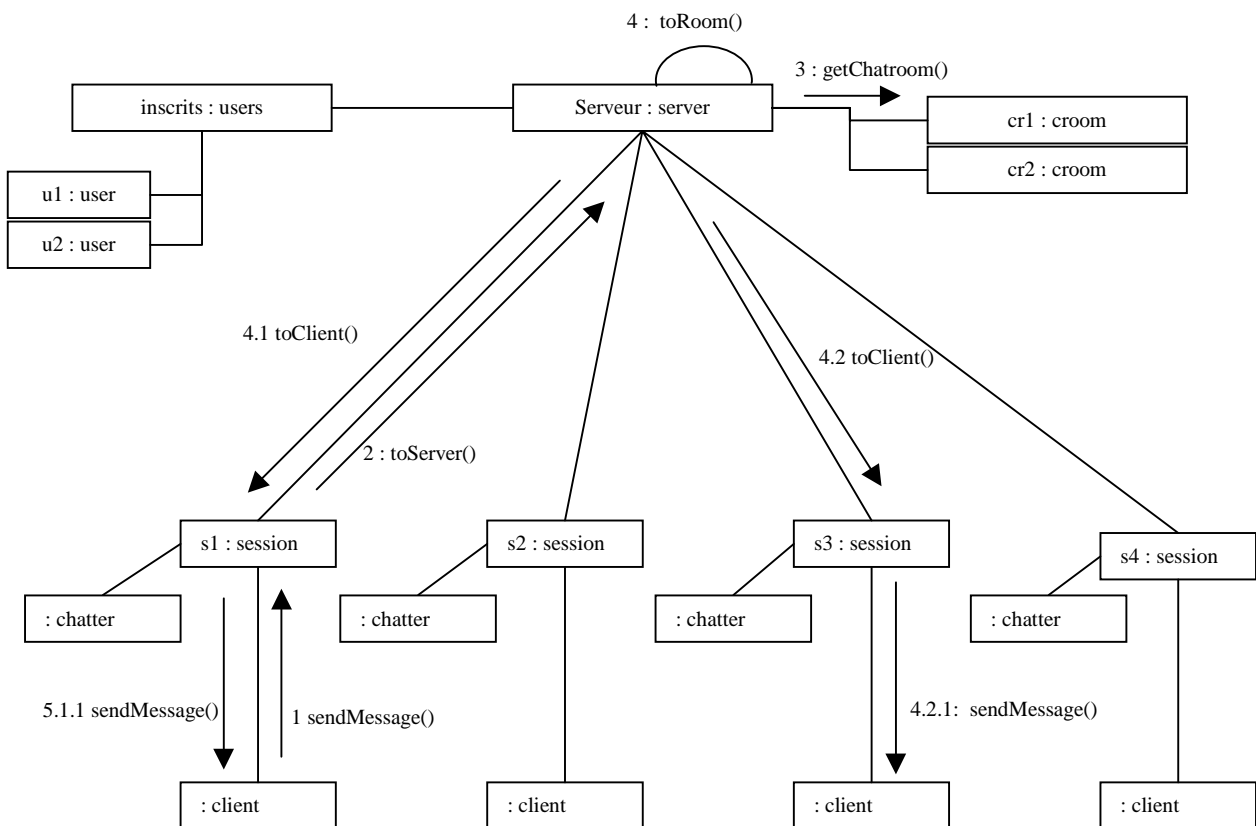


Figure 9 : un diagramme d'objets pour illustrer la phase de dialogue sur le Chat

2. Organisation des TP

Il y a 5 séances de TP. La répartition du travail dans les séances est prévue comme suit :

- TP 1 et 2 : développement des classes de bases et du protocole réseau,
- TP 3 et 4 : développement de la partie réseau incluant la programmation multi-thread et la sérialisation d'objets via un réseau,
- TP 5 : développement de l'interface utilisateur graphique côté client.

Votre travail sera évalué lors de la dernière séance (le TP 5) durant laquelle 2 heures seront consacrées à l'évaluation. En plus de l'évaluation, un examen de TP devrait être organisé ultérieurement.

Note concernant les différentes séances de TP : vous trouverez tous les renseignements nécessaires à l'utilisation des différentes classes de java à l'adresse suivante :

<http://java.sun.com/j2se/1.3/docs/api/>

où vous consulterez très probablement les packages :

- java.net
- java.util
- java.io
- java.awt
- java.awt.event
- javax.swing

Lors du TP 1, vous devrez implémenter le diagramme des classes du domaine. Vous trouverez en annexe de ce document des explications sur les techniques utilisées pour implémenter des associations. Vous trouverez de plus amples informations sur le sujet dans le livre UML 2 de B. Charroux, A. Osmani et Y.T. Mieg paru aux éditions Pearson Education.

TP 1 et 2

Notions abordées : les classes, les objets, les constructeurs, les attributs statiques, les classes conteneurs, les associations simples, l'héritage et le polymorphisme.

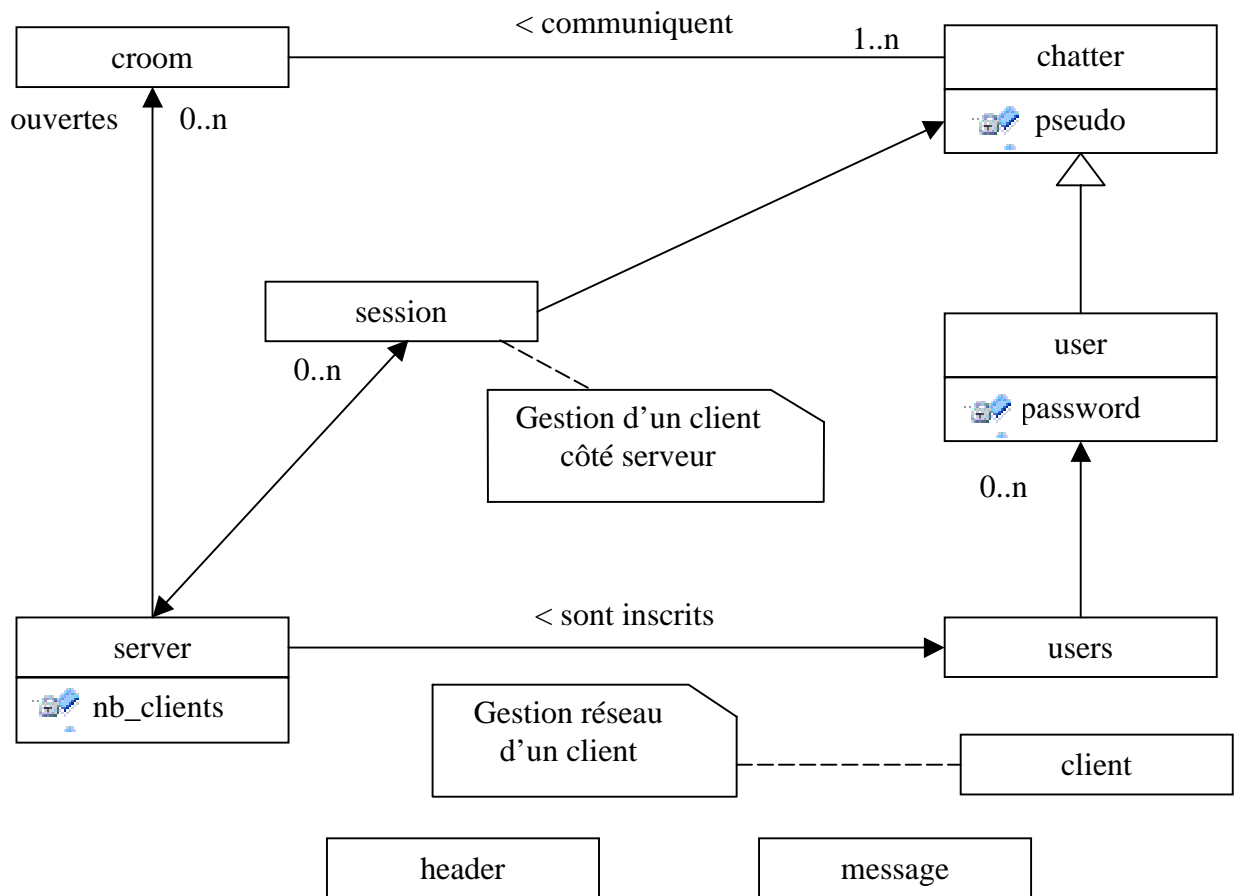


Figure 10 : diagramme des classes complété avec les classes contrôleurs

Le diagramme des classes ci-dessus modélise le système informatique simplifié d'un utilitaire de chat : des utilisateurs s'identifient auprès d'un serveur et choisissent un thème de discussion. Ils sont ensuite mis en relation avec d'autres utilisateurs.

Pour prévoir la réutilisabilité par partie de l'application les classes ont été placées dans des paquetages (figure suivante). Le paquetage `Client` contient la classe `Client`, le paquetage `Message` contient les classes `Message` et `Header`. Les classes restantes sont contenues dans le paquetage `Server`.

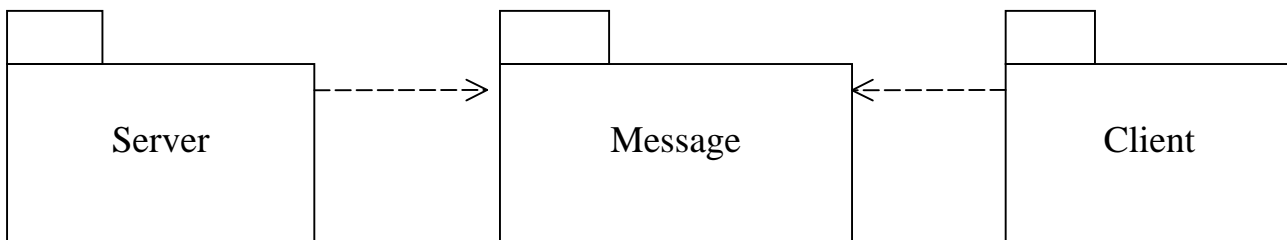


Figure 11 : les paquetages de l'application

classes chatter, user et users.

Vous constatez sur le diagramme des classes qu'un héritage existe entre les classes **chatter** et **user**. La classe **chatter** contient le pseudo du chatter, la classe **user** dérivée contient le mot de passe associé.

Ecrivez la classe **chatter** dans un fichier `chatter.java`, afin qu'elle puisse s'utiliser avec le programme suivant :

```
public static void main(String arg[])
{
    chatter toto = new chatter("gilbert");
    System.out.println(toto);    // doit afficher :
}
```

Ajouter un constructeur sans arguments qui crée le chatter dénommé "invité". Ajouter également les méthode `get` et `set` permettant d'accéder au membre **private** de cette classe.

Ecrivez la classe **user**, avec deux constructeurs, dont un sans arguments. Au moins un des deux constructeurs doit utiliser `super()`. N'oubliez pas d'implémenter la méthode **`equals()`**.

Ecrivez la classe **users**, qui permet de stocker tous les utilisateurs inscrits, et qui peut être utilisée avec le programme suivant :

```
public static void main(String arg[])
{
    user u = new user("gilbert","pass");

    users liste = new users();

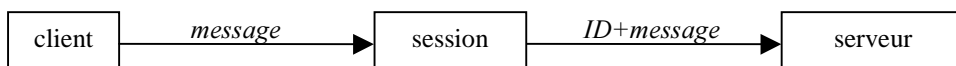
    liste.addUser(u);
    System.out.println(u+" est enregistre :"+liste.isRegistered(u)+"\n");
}
```

protocole réseau, classes headers et message

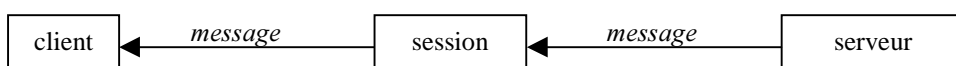
La communication entre clients, sessions et serveur est établie à l'aide de messages formatés dont l'analyse doit rester simple. L'architecture réseau (qui sera mise en place lors d'une prochaine séance) retenue est la suivante :

Un **serveur** unique gère les demandes de connexion entrantes de la part d'un ou de plusieurs **clients**. Lors de l'établissement d'une connexion, le serveur associe à chaque client une et une seule **session**. Cette session a pour rôle de relayer les messages entre client et serveur, et d'identifier le client auprès du serveur, que ce soit au niveau protocole (en rappelant son adresse IP ou tout autre identifiant), ou au niveau applicatif (en rappelant le nom du chatter associé au client qu'elle gère).

Ainsi, une communication entre client et serveur se fait de la manière suivante :



Une communication du serveur vers le client se fait de la manière suivante :



Comme pour toute communication réseau, le format d'un message tient compte du fait qu'il comporte des informations de protocole et des informations pour l'application, par exemple les textes qui sont transmis entre chatters.

La classe **message** a essentiellement pour but de rendre aisée l'analyse et le traitement des messages transmis entre les classes **client**, **session** et **server**.

A l'origine, les messages sont formés comme des **String** comportant un certain nombre de champs séparés par un délimiteur à choisir. Une instance de la classe **message** permet de découper cette String en String correspondant aux champs de la chaîne originale.

Une étude préalable du protocole réseau indique qu'un message doit comporter :

- Un champ pour l'identificateur de session;
- Un champ pour la nature du message;
- Un ou des champs pour les données.

Tous les champs ne sont pas systématiquement exploités, à l'exception du deuxième.

Exemple de message et de chaîne associée pour transmettre un texte au cours d'un chat

Le séparateur choisi pour la chaîne est, par exemple, le caractère #

Chaîne :

"#0#TEXT#comment ca va?#"

message :

champ identificateur : "0"

champ nature du message : "TEXTE"

champ de données : "comment ca va ?"

Vous êtes libres de choisir le ou les caractères délimiteurs, l'ordre des champs et le nombre de champs de données.

Ecrivez la classe message qui doit pouvoir être utilisée de la manière suivante :

```
public static void main(String arg[])
{
    message msg = new message("#0#TEXT#comment ca va?");

    String s= msg.toString();
    String donnees = msg.getField(2);

    int nb_fields = msg.getNbFields();

    System.out.println(nb_fields); // affiche 3
    System.out.println(s); // affiche #0#TEXT#comment ca va ?#
    System.out.println(donnees); // affiche : comment ca va ?
}
```

les messages seront également utilisés pour la gestion de tout le protocole réseau entre client, session et serveur, car des informations seront échangées entre des instances de ces classes avant même que du texte soit transmis entre chatters !

la classe Headers est une classe fournissant un ensemble de symboles définis comme des variables public final static.

Elle permet également de convertir un nom de commande (une String issue d'un objet **message**) en un entier.

Exemple : si vous choisissez TEXT comme nom de commande du protocole réseau, la classe Headers contiendra : `public final static int TEXT=0;` .Le numéro délivré dans cet exemple a été choisi arbitrairement.

Vous devrez également pouvoir convertir la String "TEXT" en ce numéro grâce à cette classe Headers.

Nature des messages émis :

L'application de chat devra être capable de :

- Transmettre des messages entre des chatters inscrits à la même chatroom (et donc ne pas les transmettre à ceux qui discutent dans une autre room)

- Transmettre des messages dits 'de service' (issus directement du serveur et non d'un client) à une chatroom ou à tous les chatters connectés (broadcast, pour un shutdown du serveur par exemple);
- Transmettre les messages nécessaires à toutes les étapes de l'identification et du choix d'une chatroom.

Les applications doivent se terminer proprement, c'est à dire qu'un arrêt "sauvage" d'un client doit être traité par le serveur.

c l a s s e c r o o m

L'application doit également gérer la répartition de tous les chatters au sein de chatrooms, qui sont des espaces de discussion ayant un thème particulier (nommé topic). Un client (ou une session) est affectée à une et une seule chatroom. Ainsi, si un utilisateur physique désire participer à plusieurs discussions, il devra utiliser une instance de client par discussion à laquelle il souhaite participer.

Ecrivez la classe **croom** dans un fichier nommé **croom.java**, faisant partie du package du serveur. Une instance de **croom** permet entre autres d'accéder à toutes les sessions des chatters y participant. En cas de déconnexion d'un **client**, la **session** correspondante doit être retirée de la **croom** où elle est référencée.

TP 3 et 4

Notions abordées : programmation réseau, programmation multi-thread et sérialisation d'objets

http://perso.efrei.fr/~charroux/cours/java/java_reseau.pdf

http://perso.efrei.fr/~charroux/cours/java/java_multi-thread.pdf

http://perso.efrei.fr/~charroux/cours/java/java_entrees_sorties.pdf

1. Un serveur basé sur les sockets et TCP/IP

Durant cette séance, vous allez déployer le serveur sur une machine et connecter plusieurs clients, sur cette machine et sur d'autres.

La classe serveur

Ecrivez le programme d'un tel serveur dans un fichier `server.java` à l'aide du squelette du programme donné ci-dessous :

```
public class server
{
    // membres privés

    public final static int DEFAULT_PORT = 8080;

    public server()
    {
```

Ouvrir une socket serveur pour écoute des connexions réseau

```

    le serveur est actif
    le serveur accepte encore des connexions
    attente de connexion
    création de la session associée à la connexion
}
```

```
}
```

```
public static void main(String arg[])  
{  
    server s=new server();  
}  
}
```

la classe session

Cette classe est le gestionnaire de session, et chacune de ses instances permet de faire communiquer le serveur et un client. Une instance permet donc d'"écouter" le client distant pour transmettre ses messages au serveur, et dans le même temps relayer les messages du serveur vers son client, ces deux opérations étant a priori asynchrones. Pour réaliser cela, il est nécessaire d'utiliser des Threads. La classe session se présente grossièrement de la manière suivante (on n'indique que le constructeur par défaut de cette classe) :

```
class session // attention, il faut autoriser la création d'un Thread  
{  
  
    public session() // attention il y aura probablement des paramètres  
    {  
        ouverture d'un canal de communication pour recevoir des données du  
        client via une socket  
  
        ouverture d'un canal de communication pour émettre des données  
        vers client via une socket  
  
        démarrage d'un thread d'écoute du client;  
    }  
  
    public void run()  
    {  
        boucle infinie  
        | lecture des données du client  
        | ajout de l'identifiant de session  
        | transmission de message au serveur pour traitement  
    }  
}
```

2. Programme client

Le programme client est une application distante du serveur, elle peut être exécutée sur une machine quelconque. Il lui suffit a priori de connaître l'adresse IP ou le nom de la machine hôte du serveur pour s'y connecter via une socket. La classe client ressemble beaucoup à la classe session, dans la mesure où leurs rôles sont symétriques vis à vis de l'application.

Ecrivez la classe **client** à partir de la description suivante :

```
public class client // attention, il faut autoriser la création d'un Thread
{

    public client() // attention il y aura probablement des paramètres
    {
        création de socket vers serveur(machine hôte, port)

        ouverture d'un canal de communication pour recevoir des données de
        la session via une socket

        ouverture d'un canal de communication pour émettre des données
        vers la session via une socket

        démarrage d'un thread d'écoute de la session;
    }

    public void run()
    {
        boucle infinie
        | lecture des données provenant de la session
        | traitement du message reçu
    }

    public static void main(String arg[])
    {
        client c = new client("nom_de_machine_hote_du_serveur");
        // utilisez           comme nom pour vous connecter à un serveur
        // sur la même machine
    }
}
```

```
}  
}
```

Une fois ces classes rédigées, vous pouvez mettre en place les méthodes de traitement des messages au niveau du client et du serveur pour que l'utilitaire de chat devienne pleinement fonctionnel. La transmission des messages sur le réseau implique de sérialiser des objets. Vous trouverez à l'adresse : http://perso.efrei.fr/~charroux/cours/java/java_entrees_sorties.pdf des explications sur la sérialisation.

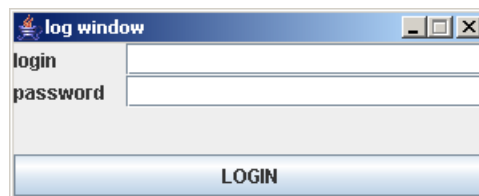
TP5

Notions abordées : construction d'une IHM, l'interfaçage avec une IHM.

http://perso.efrei.fr/~charroux/cours/java/java_AWT.pdf

Durant ces TP vous allez concevoir l'interface graphique de votre application. Reportez-vous souvent au cours (voir l'URL ci-dessus) où tout est déjà quasiment fait, mais de façon parcellaire, et suivant la norme AWT. Vous utiliserez pour ce TP la norme SWING, qui est une extension de l'AWT et qui permet la création et l'utilisation rapide de composants graphiques. Les fenêtres présentées sont indicatives, vous pouvez bien entendu organiser les composants à votre guise, tant que l'interface reste ergonomique. Libre à vous de rajouter des fonctionnalités supplémentaires.

La fenêtre d'accueil du client doit avoir l'apparence suivante :



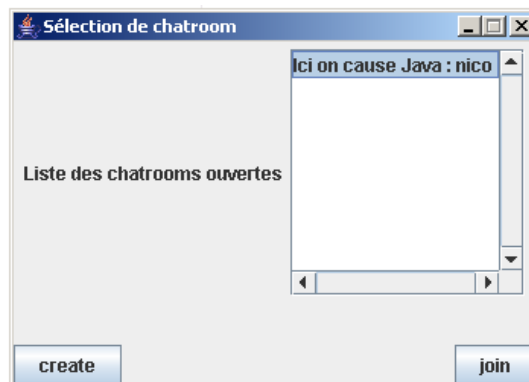
Selon l'inscription préalable de l'utilisateur et l'existence ou non de chatrooms déjà ouvertes, plusieurs fenêtres peuvent se succéder :

Utilisateur non inscrit : fenêtre de confirmation d'inscription :



Si l'inscription est confirmée, l'application continue comme si l'utilisateur était déjà inscrit

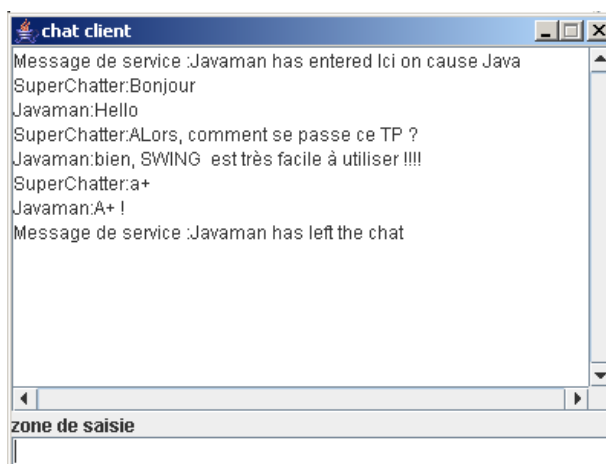
Si des chatrooms sont ouvertes, une fenêtre de sélection de chatroom indiquant les sujets (topics) et les utilisateurs y participant s'affiche.



Il est également possible de créer une nouvelle chatroom. Dans ce cas, ou encore si aucune chatroom n'est ouverte (l'utilisateur est le premier à s'identifier auprès du serveur), il doit créer une chatroom en précisant le sujet :



Enfin, la fenêtre principale de l'application de chat se présente sous la forme suivante :



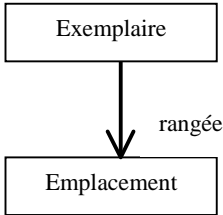
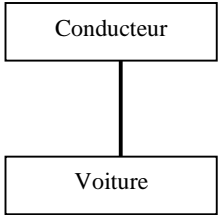
Vous pouvez par exemple ajouter un bouton "logout" à cette fenêtre.

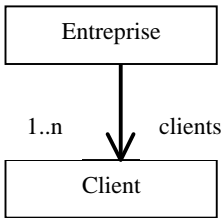
Votre œil aguerri n'aura pas manqué de noter la présence de messages de service.

Annexe

Implémentation des relations entres classes

Les langages de programmation comme Java n'offrent pas de technique particulière pour implémenter des associations, des agrégations ou des compositions. Ce type de relation s'implémente en ajoutant des attributs dans des classes.

<p>Association unidirectionnelle de 1 vers 1 :</p>  <pre> classDiagram class Exemplaire class Emplacement Exemplaire --> Emplacement : rangée </pre> <p><u>Remarque</u> : pour accéder à l'association plus facilement (sans passer par les méthodes <code>setEmplacement</code> et <code>getEmplacement</code>), il est possible de supprimer le mot clef <code>private</code> (ce n'est possible que si les classes sont dans un même paquetage).</p>	<pre> public class Emplacement{ } public class Exemplaire{ private Emplacement rangée; public void setEmplacement(Emplacement emplacement){ this.rangée = emplacement; } public Emplacement getEmplacement(){ return rangée; } public static void main(String [] argv){ Emplacement emplacement = new Emplacement(); Exemplaire exemplaire = new Exemplaire(); exemplaire.setEmplacement(emplacement); Emplacement place = exemplaire.getEmplacement(); } } </pre>
<p>Association bidirectionnelle de 1 vers 1 :</p>  <pre> classDiagram class Conducteur class Voiture Conducteur -- Voiture Voiture -- Conducteur </pre>	<pre> package bagnole; public class Conducteur{ Voiture voiture; public void addVoiture(Voiture voiture){ if(voiture != null){ this.voiture = voiture; voiture.conducteur = this; } } public static void main(String [] argv){ Voiture voiture = new Voiture(); } } </pre>

	<pre> Conducteur conducteur = new Conducteur(); conducteur.addVoiture (voiture); } } package bagnole; public class Voiture{ Conducteur conducteur; public void addConducteur(Conducteur conducteur){ this.conducteur = conducteur; conducteur.voiture = this; } } </pre>
<p>Association unidirectionnelle de 1 vers plusieurs :</p>  <pre> classDiagram Entreprise "1" --> "n" Client : clients </pre>	<pre> public class Client{ } import java.util.Vector; public class Entreprise{ private Vector clients = new Vector(); public void addClient(Client client){ clients.addElement(client); } public void removeClient(Client client){ clients.removeElement(client); } public static void main(String [] argv){ Entreprise monEntreprise = new Entreprise(); Client client = new Client(); monEntreprise.addClient(client); monEntreprise.removeClient(client); } } </pre>