

TP ILO - Réalisation d'un système de chat en JAVA

Clément BRUN

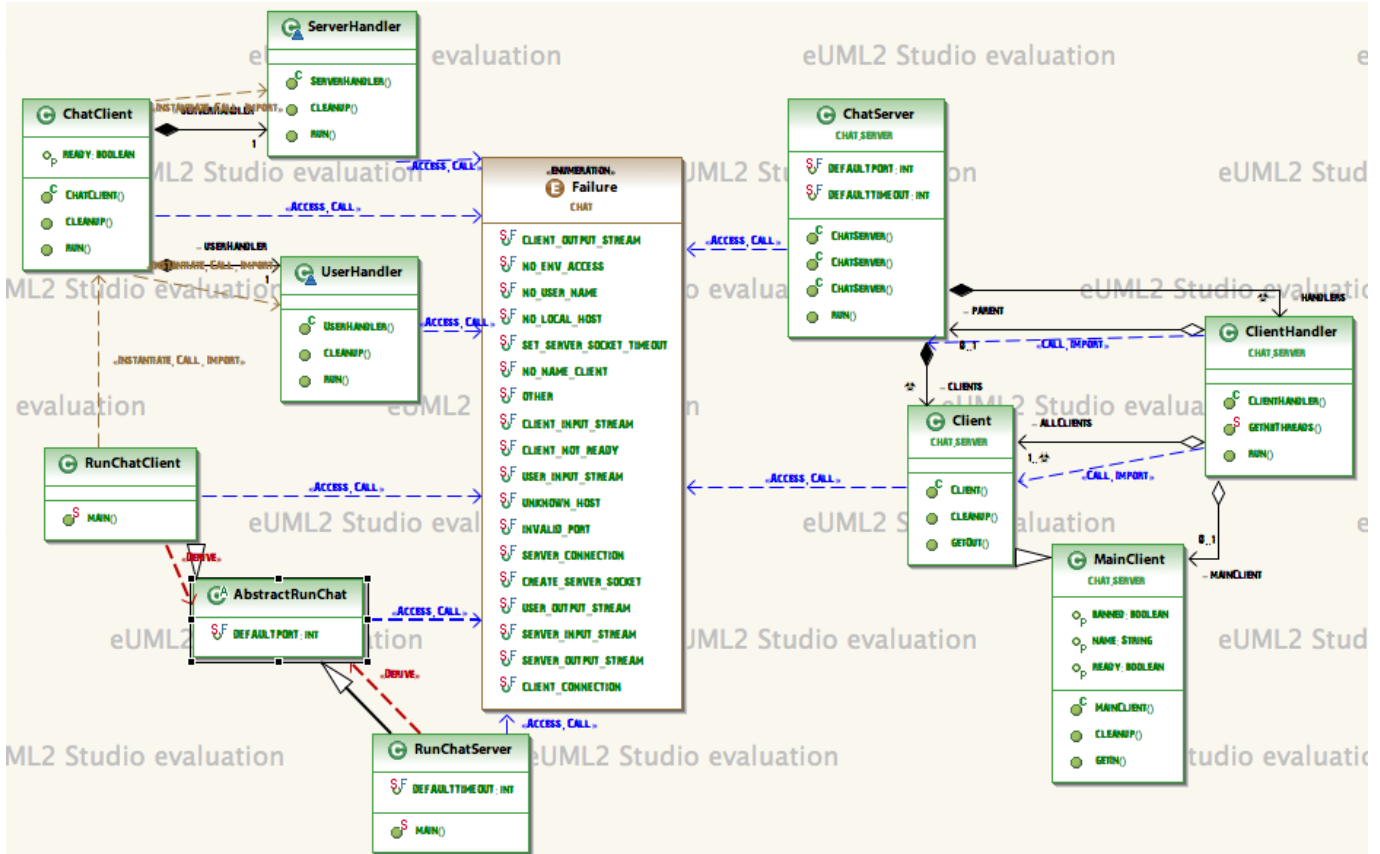
21 janvier 2013

Table des matières

1	UML	2
1.1	Diagramme généré	2
1.2	Réponses aux questions	2
2	Commande kick	3
3	Interface graphique	3
3.1	Réponse à la question	3
3.2	Création de la fenêtre	3
3.3	Pannels utilisés	3
3.4	Mise en place de la structure de la fenêtre	4
4	Liaison entre le client et le serveur	4
4.1	Réception des messages	4
4.2	Envoyer un message au serveur	5
4.3	Coloration du code	6
4.4	Déconnexion du serveur	6
5	Annexe 1 : Interface graphique dans RunChatClient.java	7
6	Annexe 2 : ClientFrame.java	7

1 UML

1.1 Diagramme généré



1.2 Réponses aux questions

1. La classe *AbstractRunChat* permet d'avoir une interface entre *RunChatClient* et *RunChatServer*. Ainsi, nous n'avons qu'une seule classe pour faire le lien entre le port du serveur. Elle sert aussi à partager des fonctions utilisées dans les deux classes.
2. *ChatServer* contient un vecteur de *Client*. Le serveur possède donc 0, 1 ou plusieurs clients. La classe *Client* permet de stocker les informations sur le client en question. Le serveur peut donc avoir accès à ces informations.
3. Le *mainClient* est le super utilisateur. Il a le droit de modérer les autres clients sur le serveur.
4. Le client dit *maître* peut avoir accès à la liste des clients connectés au serveur.
5. Le serveur attend la connexion d'un client par *clientSocket = serverSocket.accept()*; . Par la suite, le nom du client est récupéré grâce à *clientName = reader.readLine()*;
Si ce nom est valide, nous créons un client maître : *ClientHandler handler = new ClientHandler(this, newClient, clients)*; Cet instance nous permet de créer notre *thread*. Nous avons donc un *thread* par client connecté au serveur. Dans le cas figurant sur l'annoncé, nous avons deux *thread* coté serveur.
6. Coté client, le *ChatClient* lance deux *threads*. Un pour l'utilisateur maître et l'autre pour la liaison avec le serveur sur le *ServerHandler*.
7. Le *join()* permet d'attendre la fin du thread (sa mort). Une fois que les *threads* sont fermés, nous pouvons continuer la suite du programme.

2 Commande kick

```
//StringTokenizer pour repérer la commande kick
StringTokenizer st = new StringTokenizer(clientInput.toLowerCase());
String nameKick=null;

if (st.hasMoreTokens()) {
    //Nous vérifions si le premier token est la commande kick
    if (st.nextToken().equals(Vocabulary.kickCmd) && st.hasMoreTokens())
        //Si c'est le cas, le deuxième est forcément le pseudo
        nameKick=st.nextToken();
}

//Si nous avons un nom
if (nameKick != null)
{
    //Nous vérifions les privilèges
    if (mainClient.getName().equals(allClients.get(0).getName()))
    {
        //Nous recherchons le client parmi les connectés
        Client ck = parent.searchClientByName(nameKick);

        if (ck!=null)
        {
            System.out.println(nameKick+" est kick du serveur par "+mainClient.getName());
            broadcast.append(" kick de "+nameKick+ "!");
            ck.setBanned(true); //Client bannis
        }
        else
        {
            broadcast.append("Pseudo introuvable.");
        }
    }
    else
    {
        broadcast.append(" Cette instruction requiert les droits de super utilisateur.");
    }
}
}
```

3 Interface graphique

3.1 Réponse à la question

Nous souhaitons intégrer une architecture *MVC*. En effet, la vue que l'on va créer ne va pas modifier l'architecture de notre client. Nous pouvons donc créer autant d'interfaces graphiques que nécessaires car elles sont indépendantes du modèle (Code du client).

3.2 Création de la fenêtre

3.3 Pannels utilisés

```
protected JButton btnLeftButton; //Bouton de déconnexion
protected JLabel serveur; //Affichage des informations sur le serveur
protected JTextField message; //Zone de texte pour insérer notre message
protected JButton send; //Bouton pour envoyer le message
protected JButton btnClearButton; //Bouton pour effacer les anciens messages
protected JScrollPane chat; // Permet d'afficher le chat
```

3.4 Mise en place de la structure de la fenêtre

```
//Barre d'outils en haut
JToolBar toolBar = new JToolBar();
toolBar.setFloatable(false);
getContentPane().add(toolBar, BorderLayout.NORTH);
//Bouton de déconnexion
btnLeftButton = new JButton("Se deconnecter");
toolBar.add(btnLeftButton);

//Effacer les anciens messages
btnClearButton = new JButton("Clear");
toolBar.add(btnClearButton);
//Espace entre les boutons
Component horizontalGlue = Box.createHorizontalGlue();
toolBar.add(horizontalGlue);

//Affichage des informations du serveur
serveur = new JLabel("Serveur : 127.0.0.1:0000");
toolBar.add(serveur);

//Affichage des messages du chat
chat = new JScrollPane();
getContentPane().add(chat, BorderLayout.CENTER);

JTextPane textPane = new JTextPane();
textPane.setEditable(false);

chat.setViewportViewView(textPane);

//Deuxième barre d'outils en bas de la fenêtre

JToolBar toolBar2 = new JToolBar();
getContentPane().add(toolBar2, BorderLayout.SOUTH);
toolBar2.setFloatable(false);

//Input pour insérer un message
message = new JTextField(8);
    message.setLocation(0, 0);
    message.setSize(100, 30);

toolBar2.add(message);

toolBar2.add(Box.createHorizontalGlue());
//Bouton pour envoyer
send = new JButton("Send button");
toolBar2.add(send);
```

4 Liaison entre le client et le serveur

4.1 Réception des messages

Dans un premier temps, nous devons créer un flux d'entrée dans notre classe *ClientFrame* :

```
PipedInputStream in;
```

Nous pouvons accéder à ce flux par un accesseur *getIn()*. Après l'instanciation de *ChatFrame* dans notre *RunChatClient*, nous pouvons rediriger le *out* sur notre *getInt()*.

```

try {
    out = new PipedOutputStream(cl.getIn());
} catch (IOException e) {
    // TODO Auto-generated catch block
    e.printStackTrace();
}

```

Ensuite, il nous suffit de traiter le flux d'entrée dans la méthode *run()* de *ChatFrame*.

```

public void run()
{
    InputStreamReader isr = new InputStreamReader(in);
    BufferedReader br = new BufferedReader(isr);

    String ligne;
    try {
        while ((ligne=br.readLine())!=null){
            textPane.setText(ligne); //Affichage dans la fenetre sans style
        }
    } catch (IOException e) {
        e.printStackTrace();
    }

    cleanup();
}

```

4.2 Envoyer un message au serveur

Nous devons créer un flux de sortie dans notre classe *ClientFrame* :

```
protected PipedOutputStream out;
```

Ce flux est accessible par la méthode *getOut()*. Le flux d'entrée du serveur sera redirigé vers ce flux de sortie.

```

try {

    in = new PipedInputStream(cl.getOut());
} catch (IOException e) {
    // TODO Auto-generated catch block
    e.printStackTrace();
}

```

Dans notre *ClientFrame*, il faut créer un *BufferWriter* pour écrire dans ce flux de sortie. J'ai donc mis en place deux variables privées :

```

protected OutputStreamWriter osw;
protected BufferedWriter bw;
.
.
.
osw = new OutputStreamWriter(out);
bw = new BufferedWriter(osw);
.
.
.

```

Avec ces variables d'instance, j'ai pu créer une méthode qui me permet d'envoyer les informations au serveur.

```
protected void sendMessage(String msg)
{
    try {

        bw.write(msg); //Ecriture dans le flux.
        bw.newLine(); //Saut de ligne pour finir le message.
        bw.flush(); //Envoi au serveur.

    } catch (IOException e) {

        e.printStackTrace();
    }

}
```

4.3 Coloration du code

La coloration du code se fait avec le *HashCode* du pseudo. Voici la fonction qui permet de générer automatiquement une couleur :

```
protected Color generateColor(String name)
{
    Color c=null;
    switch(name.hashCode()%8)
    {
        case 0 : c=Color.BLACK ; break;
        case 1 : c=Color.BLUE ; break;
        case 2 : c=Color.GRAY ; break;
        case 3 : c=Color.GREEN ; break;
        case 4 : c=Color.MAGENTA ; break;
        case 5 : c=Color.ORANGE ; break;
        case 6 : c=Color.PINK ; break;
        case 7 : c=Color.RED ; break;
    }

    return c;
}
```

4.4 Déconnexion du serveur

Comme fait dans la console, il suffit d'envoyer *bye* au serveur lorsque l'on appuie sur le bouton.

```
protected class QuitServer implements ActionListener
{
    public void actionPerformed(ActionEvent e)
    {
        sendMessage("bye");
    }
}
```

5 Annexe 1 : Interface graphique dans RunChatClient.java

```
if (gui)
{
    commonRun = null;

    /*
     * Création de la ClientFrame
     */

    ClientFrame cl = new ClientFrame(host,// hôte du serveur
                                     name, // nom d'utilisateur
                                     commonRun, // commonRun avec le GUI
                                     verbose);// verbose

    /*
     * Création du flux de sortie vers l'utilisateur
     * (à connecter au flux d'entrée de la ClientFrame)
     */

    try {
        out = new PipedOutputStream(cl.getIn());
    } catch (IOException e) {
        // TODO Auto-generated catch block
        e.printStackTrace();
    }

    /*
     * Création du flux d'entrée depuis l'utilisateur
     * (à connecter au flux de sortie de la ClientFrame)
     */
    try {
        in = new PipedInputStream(cl.getOut());
    } catch (IOException e) {
        // TODO Auto-generated catch block
        e.printStackTrace();
    }

    /*
     * Création et lancement du thread de la frame
     */
    Thread ct = new Thread(cl);
    threadPool.add(ct);
    ct.start();

}
```

6 Annexe 2 : ClientFrame.java

Les imports ne sont pas pris en compte.

```
public class ClientFrame extends JFrame implements Runnable
{
    protected JButton btnLeftButton; //Bouton de déconnexion
    protected JLabel serveur; //Affichage des informations sur le serveur
    protected JTextField message; //Zone de texte pour insérer notre message
    protected JButton send; //Bouton pour envoyer le message
    protected JButton btnClearButton; //Bouton pour effacer les anciens messages
    protected JScrollPane chat; // Conteneur pour l'affichage du chat
    protected JTextPane textPane; // Affichage du chat

    protected StyledDocument document; //Style pour le document
    protected Style style; //Style pour le document

    protected PipedOutputStream out; //Sortie std
    protected PipedInputStream in; // Entrée std

    protected OutputStreamWriter osw; //Permet de traiter la sortie
    protected BufferedWriter bw; //Permet de traiter la sortie

    protected String name; //Nom du client

    /**
     * Constructeur de la fenêtre
     *
     * @param name le nom de l'utilisateur
     * @param host l'hôte sur lequel on est connecté
     * @param commonRun État d'exécution des autres threads du client
     * @param verbose pour les messages de debug
     * @throws HeadlessException
     */
    public ClientFrame(String name, String host, Boolean commonRun,
        boolean verbose) throws HeadlessException
    {

        //INITIALISATION DES DONNÉES
        //Nous enregistrons le nom du client
        this.name = name;

        //Instanciation des flux
        out = new PipedOutputStream();
        in = new PipedInputStream();

        //Le flux de sortie doit être traité dans un buffer
        osw = new OutputStreamWriter(out);
        bw = new BufferedWriter(osw);

        //Affichage de la fenêtre + dimensions
        this.setVisible(true);
        this.setPreferredSize(new Dimension(400, 200));
        this.setSize(new Dimension(400, 200));

        //CONSTRUCTION DE LA FENÊTRE
        JToolBar toolBar = new JToolBar();
        toolBar.setFloatable(false);
```



```

getContentPane().add(toolBar, BorderLayout.NORTH);

btnLeftButton = new JButton("Se deconnecter");
toolBar.add(btnLeftButton);

btnClearButton = new JButton("Clear");
toolBar.add(btnClearButton);

//Espace entre cler+déco et paramètres du serveur
Component horizontalGlue = Box.createHorizontalGlue();
toolBar.add(horizontalGlue);

//Affichage des informations serveur
serveur = new JLabel("Serveur : "+host);
toolBar.add(serveur);

chat = new JScrollPane();
getContentPane().add(chat, BorderLayout.CENTER);

textPane = new JTextPane();
textPane.setEditable(false);
document = textPane.getStyledDocument();

chat.setViewportView(textPane);

//Barre en dessous pour le message + envoi.
JToolBar toolBar2 = new JToolBar();
getContentPane().add(toolBar2, BorderLayout.SOUTH);
toolBar2.setFloatable(false);

message = new JTextField(8);
    message.setLocation(0, 0);
    message.setSize(100, 30);

toolBar2.add(message);

toolBar2.add(Box.createHorizontalGlue());

send = new JButton("Send button");
send.addActionListener(new SendButtonListener());
toolBar2.add(send);

style = textPane.addStyle("New Style", null);

//Ajout des listeners
btnClearButton.addActionListener(new ClearListener(document));
btnLeftButton.addActionListener(new QuitServer());
}

//Accesseurs sur les flux
public PipedOutputStream getOut() {
    return out;
}

public PipedInputStream getIn() {
    return in;
}

```

```

/**
 * Exécution de la boucle d'exécution. La boucle d'exécution consiste à lire
 * une ligne sur le flux d'entrée avec un BufferedReader tant qu'une erreur
 * d'IO n'intervient pas indiquant que le flux a été coupé. Puis afficher
 * ce message dans le document sous-jacent du textPane.
 */
@Override
public void run()
{
    //Initialisation en local des flux d'entrée
    InputStreamReader isr = new InputStreamReader(in);
    BufferedReader br = new BufferedReader(isr);

    //Lecture du buffer
    String ligne;
    try {
        while ((ligne=br.readLine())!=null){
            //Affichage du message
            addMessage(ligne, generateColor(name));
        }
    } catch (IOException e) {
        e.printStackTrace();
    }

    cleanup();
}

/**
 * Fermeture de la fenêtre et des flux à la fin de l'exécution
 */
public void cleanup()
{
    try {
        //Fermeture des flux
        bw.close();
        osw.close();
        out.close();
        in.close();
    } catch (IOException e) {
        e.printStackTrace();
    }
}

/**
 * Ajout d'un texte à la fin du document avec la couleur voulue
 * @param text le texte à ajouter
 * @param color la couleur du texte
 */
protected void addMessage(String text, Color color)
{
    StyleConstants.setForeground(style, color);

    try

```

```

{
    //La date
    DateFormat df = new SimpleDateFormat("yyyy/MM/dd");
    String formattedDate = df.format(new Date());

    document.insertString(document.getLength(),
        "["+formattedDate+"] "+text
        + Vocabulary.newLine, style);
}
catch (BadLocationException ex)
{
    System.err.println("write at bad location");
    ex.printStackTrace();
}
}

protected void sendMessage(String msg)
{
    try {

        bw.write(msg); //Ecriture dans le flux de sortie
        bw.newLine(); //Nouvelle ligne
        bw.flush(); //Envoi

        //bw.close();

    } catch (IOException e) {

        e.printStackTrace();
    }

}

/*
 * Gestion des couleurs
 */
protected Color generateColor(String name)
{
    Color c=null;
    switch(name.hashCode()%8)
    {
        case 0 :    c=Color.BLACK ; break;
        case 1 :    c=Color.BLUE  ; break;
        case 2 :    c=Color.GRAY   ; break;
        case 3 :    c=Color.GREEN  ; break;
        case 4 :    c=Color.MAGENTA ; break;
        case 5 :    c=Color.ORANGE ; break;
        case 6 :    c=Color.PINK   ; break;
        case 7 :    c=Color.RED    ; break;
    }

    return c;
}

protected class SendButtonListener implements ActionListener

```

```

{
    @Override
    public void actionPerformed(ActionEvent e)
    {
        if (!message.getText().equals(""))
        {
            sendMessage(message.getText());
            message.setText("");
        }

    }
}

protected class QuitServer implements ActionListener
{
    @Override
    public void actionPerformed(ActionEvent e)
    {
        sendMessage("bye");
    }
}
}

```