La Phase de Conception

L'analyse a permis d'identifier un certain nombre de concepts métiers que notre application à la charge de réaliser. Il faut maintenant proposer un découpage en package afin de rendre le développement modulaire. Ce découpage permet de mieux appréhender la fonctionnalité de chaque partie de l'application, et permet la définition des *interfaces* essentielles pour un développement modulaire.
Les interfaces permettent de gagner en abstraction, et sont facilement réalisables dans les langages orientés objets à travers l'héritage. C'est une des forces de ce modèle OO, il faut cependant l'utiliser de façon appropriée.

Etape 1 : Découpage en packages.

L'analyse a révélé plusieurs séquences considérées comme primordiales, et cependant suffisamment découplées pour figurer sur des diagrammes de séquence séparés. Ce sont (Figure 4 du document d'analyse) :

1. L'ouverture de session, qui révèle la nécessité d'une gestion du réseau. Il va nous falloir un *package Net* pour loger les fonctionnalités liées au réseau. Si possible, on aimerait limiter les dépendances sur ce package. On voit apparaître la notion de *Client* et *Serveur*. Ce sont aussi deux noms de package candidats, vu leur importance pressentie en termes de développement (on va vraiment faire ce Chat en 5 séances de 4 heures ??). Clairement, ces deux packages dépendront du réseau.
2. L'identification, qui révèle l'existence d'un processus d'authentification. Il va nous falloir un package qui sache gérer les mots de passe des membres et l'authentification. On créera donc un package *authentification*.
3. Le choix de la chatroom, qui nécessite un *gestionnaire de sujets de discussion*, capable de trouver et de créer des salles de discussion. On pourrait créer un package pour ces fonctions de gestion des « *topics* ». Notons que dans ces interactions, une association unidirectionnelle du Chatter vers le gestionnaire de topics suffit.
4. La discussion, qui révèle de fortes interactions entre l'entité d'analyse *ChatRoom* et l'entité *Chatter*. Les dépendances croisées entre ces deux concepts pousse à plutôt les identifier comme des classes au sein d'un même package chargé de gérer les discussions.

Etudions le graphe de dépendances du découpage proposé.

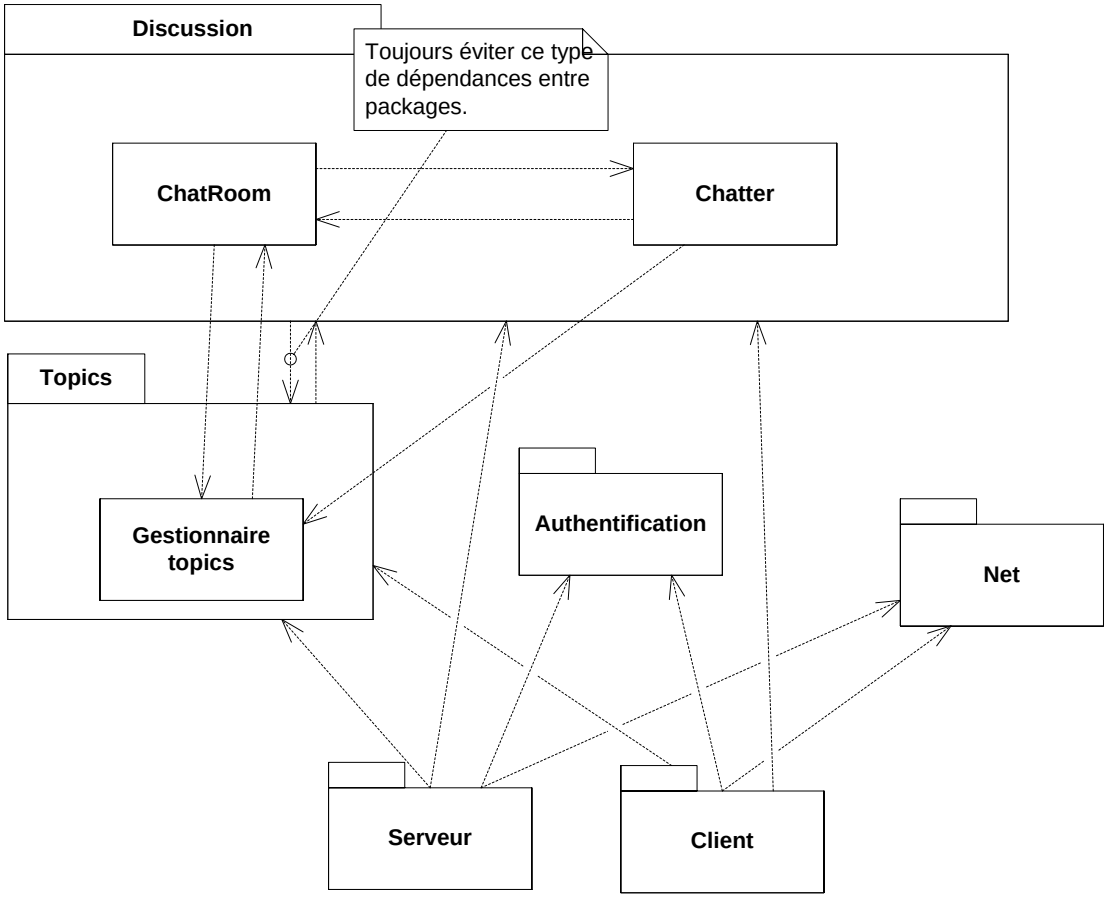


Figure 1. Un découpage préliminaire en package.

Ce découpage présente des bons aspects :

- Le client et le serveur sont indépendants, grâce à l'introduction du package Net, qui devra standardiser les échanges réseau. En effet, il est normal (et désirable en termes de nombres d'utilisateurs) qu'on puisse développer un client indépendant. Un protocole réseau est essentiel pour permettre la compatibilité avec d'autres applications.
- Le package Net est isolé : il ne dépend de rien, et seul client et serveur dépendent de lui. Le réseau est correctement partitionné des autres services de l'application (topics, discussion).
- L'authentification est isolée. Elle ne dépend pas du réseau, et n'interfère pas avec la discussion ou les topics. Notons que ce découpage ne préserve pas le lien d'héritage identifié (en analyse, figure 10) entre les classes User (avec un password, qui relève donc de l'authentification) et Chatter. Nous choisissons de découpler ces classes. Ainsi, un utilisateur pourra avoir un pseudo dans une session de chat et un identifiant de compte pour se connecter différent. Le compte « guest » sans mot de passe pourra avoir plusieurs pseudo, on pourra modifier son pseudo... Ce choix parait raisonnable. La classe user perds donc son héritage sur Chatter, et a maintenant deux attributs le *login* et le *password*.
- La dépendance des Client et Serveur est unidirectionnelle vers les autres morceaux de l'application, ce sont des élements en fin de chaine (produits assemblés). On identifie donc les parties Topics et Discussion comme des parties réutilisables dans l'application.

Par contre certaines dépendances ne sont pas satisfaisantes :

- La dépendance bidirectionnelle entre Chatter et ChatRoom est correctement isolée.

- Cependant, le gestionnaire de topics gère des ChatRoom, ce qui induit une dépendance du package Topics vers le package Discussion.
- D'autre part, le Chatter doit connaître son gestionnaire de topics, pour trouver et créer une chatroom. Ceci induit une dépendance du package Discussion vers Topics.

Ce découpage est à revoir, fusionnons ces deux packages, en un package Chat. (Voir figure 2)

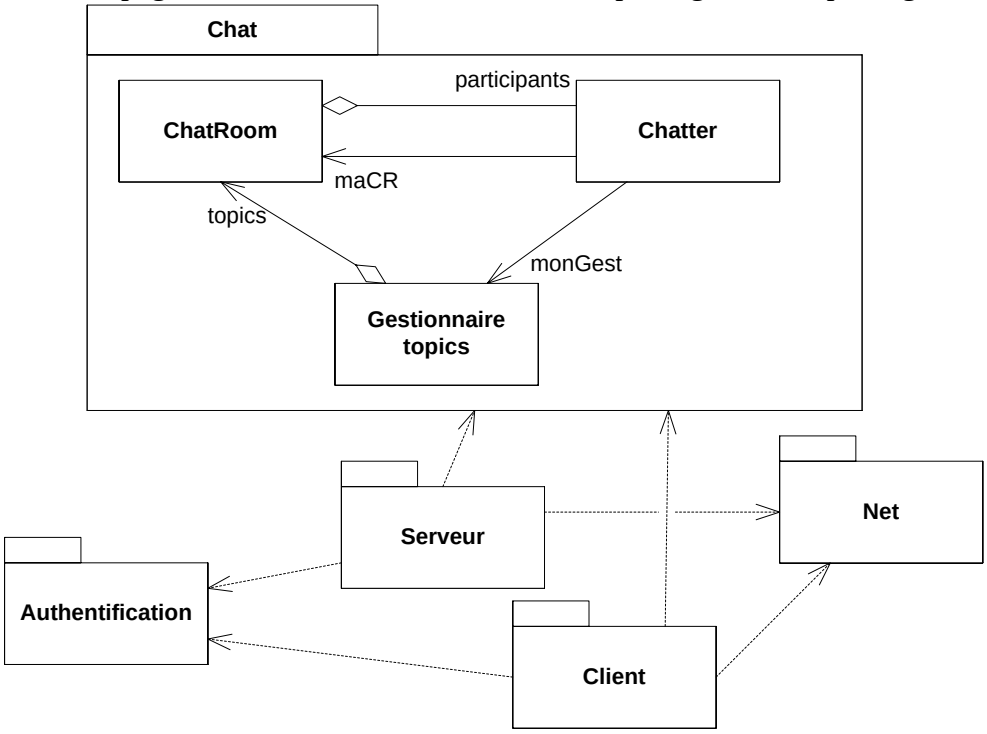


Figure 2. Un découpage en package moins fin. Nous avons précisé les liens de dépendances entre les trois classes identifiées du package Chat en fixant mieux les associations entre ces classes.

Ce nouveau découpage rends le package Chat proprement isolé. Il ne reste que nos deux applications (package serveur et client) qui dépendent du reste. Notre découpage favorise la réutilisation de certains packages.

Etape 2 : Etude détaillée des packages en bout de chaîne.

Séance 1 : Le package Chat

Etudions de plus près les interactions entre les classes de notre package Chat.
Notons que cette étude est ainsi découplée des problèmes d'authentification ou de réseau.

On va dans un premier temps manipuler ces concepts comme des classes abstraites ou des interfaces, deux notions très proches. Un diagramme de séquence (Figure 3) permet de capturer les interactions entre ces interfaces.

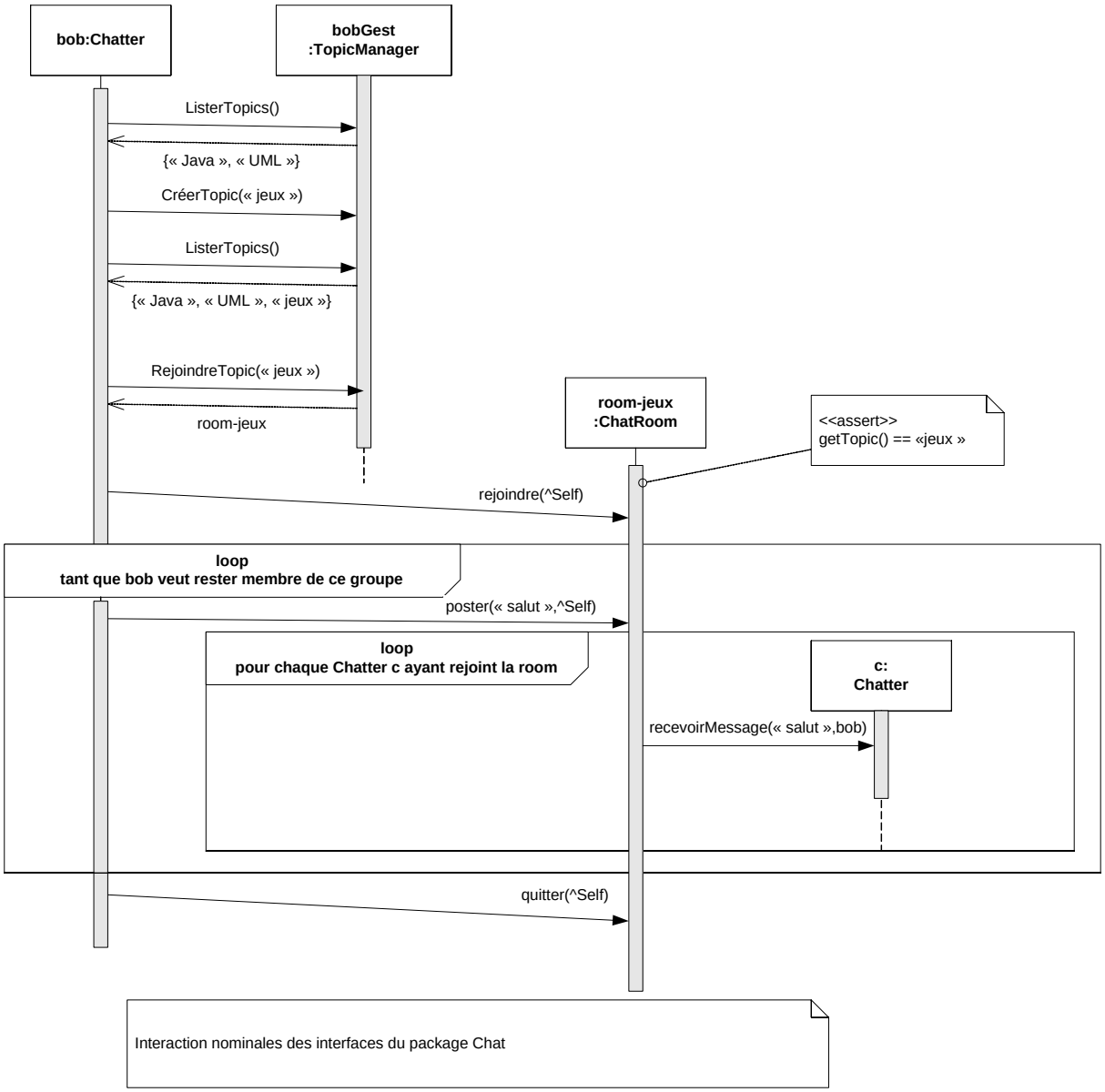


Figure 3 : Séquence principale d'interaction nominale, du point de vue d'un Chatter. La recherche de la ChatRoom (pour RejoindreTopic, CréerTopic …) suppose l'utilisation de l'association « topics » de la classe TopicManager vers ChatRoom.

Cette interaction permet de préciser les interfaces de ces trois classes (figure 4).

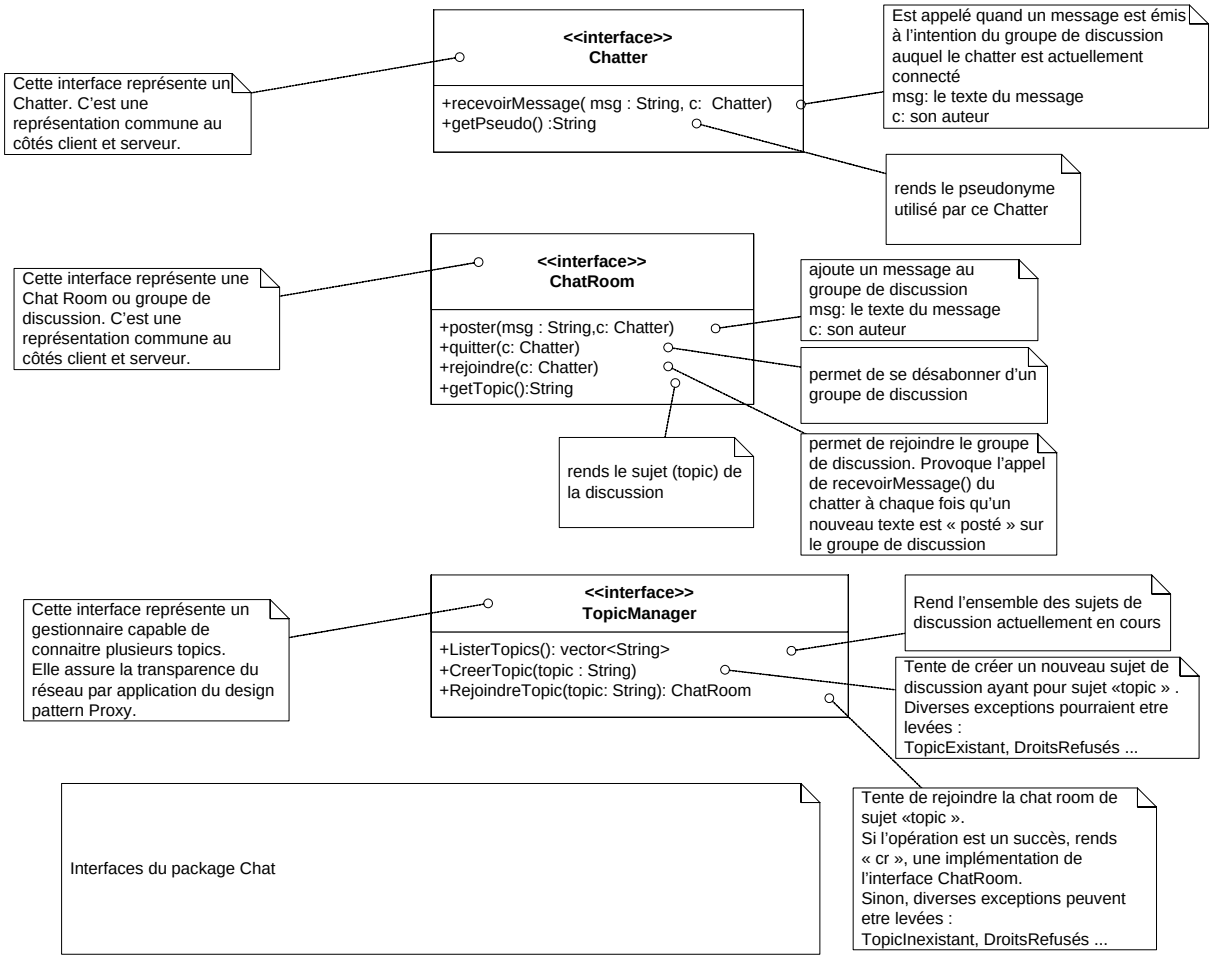
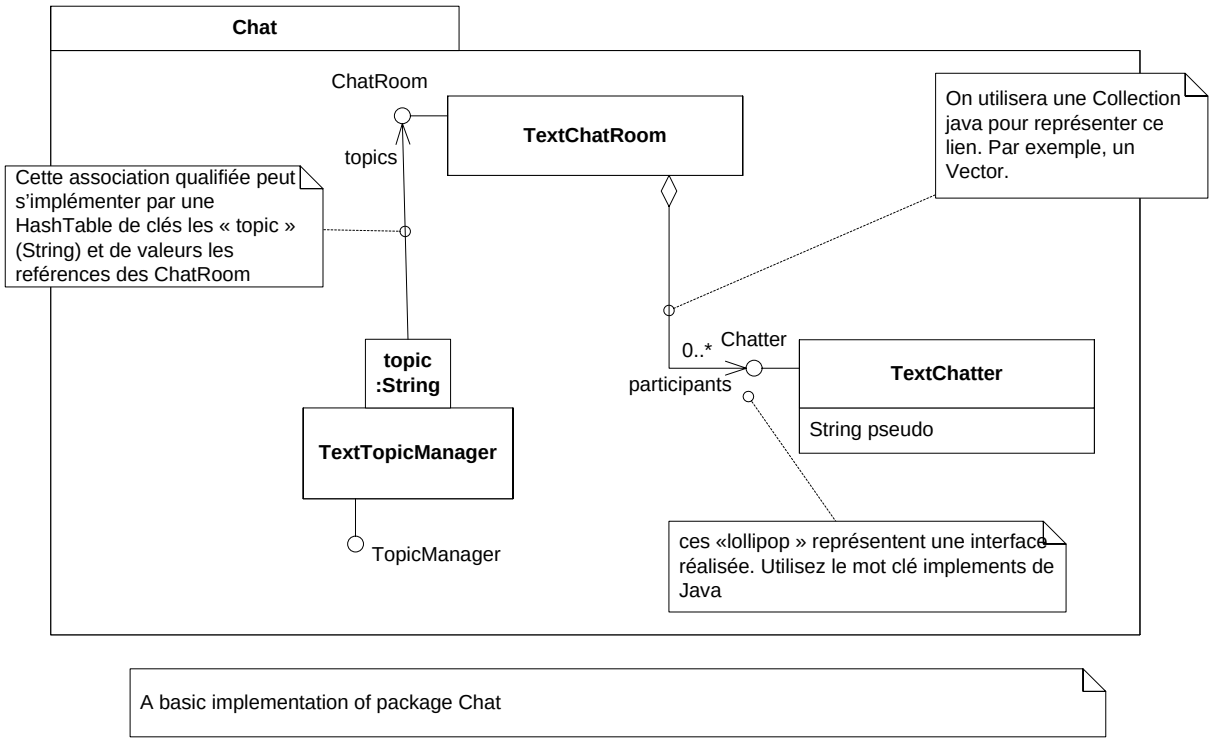


Figure 4 : Description détaillée du rôle des classes principales du package Chat.

On peut à ce stade proposer une implémentation minimaliste mais fonctionnelle de ce sous-système (un *prototype*).

Travail Demandé :

Implémentez ce prototype : codez les interfaces en java, et leur réalisation par une implémentation simple qui s'appuie sur la sortie standard.
Dans un package Chat, créez ces trois interfaces.
Dans le même package, implémentez les classes ci-dessous, qui se contenteront d'afficher des messages (System.out.println()).



Ces classes devront pouvoir être testées à l'aide du main suivant : c'est une séquence de test intéressante, qui valide notre conception à ce stade.

```
      public static void main(String[] args) {
            Chatter bob = new TextChatter ("Bob");
            Chatter joe = new TextChatter ("Joe");
            TopicManager gt = new TextTopicManager();

            gt.creerTopic("java");
            gt.creerTopic("UML");
            System.out.println("Les topics ouverts sont:"
                              + gt.listerTopics());
            gt.creerTopic("jeux");
            System.out.println("Les topics ouverts sont:"
                              + gt.listerTopics());
            ChatRoom cr = gt.rejoindreTopic("jeux");
            cr.rejoindre(bob);
            cr.poster("Je suis seul ou quoi ?",bob);
            cr.rejoindre(joe);
            cr.poster("Tiens, salut Joe !",bob);
            cr.poster("Toi aussi tu chat sur les forums de jeux pendant les
TP, Bob?",joe);
      }
```

Fournissant un affichage au moins aussi riche que celui-ci :

```
Les topics ouverts sont : [UML,java]
Les topics ouverts sont : [UML,java,jeux]
(Message de Chatroom :jeux) Bob a rejoint la room.
(chez Bob) : Bob $> Je suis seul ou quoi ?
(Message de Chatroom :jeux) Joe a rejoint la room.
(chez Bob) : Bob $> Tiens, salut Joe !
(chez Joe) : Bob $> Tiens, salut Joe !
(chez Bob) : Joe $> Toi aussi tu chat sur les forums de jeux pendant les
TP, Bob?
(chez Joe) : Joe $> Toi aussi tu chat sur les forums de jeux pendant les
TP, Bob?
```

Notons l'utilisation des interfaces dans ce source. Essayez de référer au types TextXXX le moins possible : manipulez vous aussi dans votre implémentations des références sur des Chatter, ChatRoom, TopicManager. Ainsi, nous fournissons une implémentation par défaut de ces interfaces, utilisable dans d'autres tests des composants de l'application.
Notons également que l'implémentation de ce schéma simplifié permet naturellement à un même Chatter d'être connecté à plusieurs ChatRoom. Cet aspect pourra être exploité pour traiter la connexion simultanée à plusieurs groupes de discussion, quand on réalisera les composants Client et Serveur, instanciations plus « solides » de ces interfaces du package chat.

Séance 2 : Le package Authentification

Ce package est chargé de gérer l'authentification d'utilisateurs. Analysons ses cas d'utilisation ; dans ce niveau d'analyse, les autres composants de l'application sont vus comme des acteurs (ils sont externes au sous-système considéré, et interagissent avec lui). On découvre deux acteurs : un administrateur et un utilisateur.

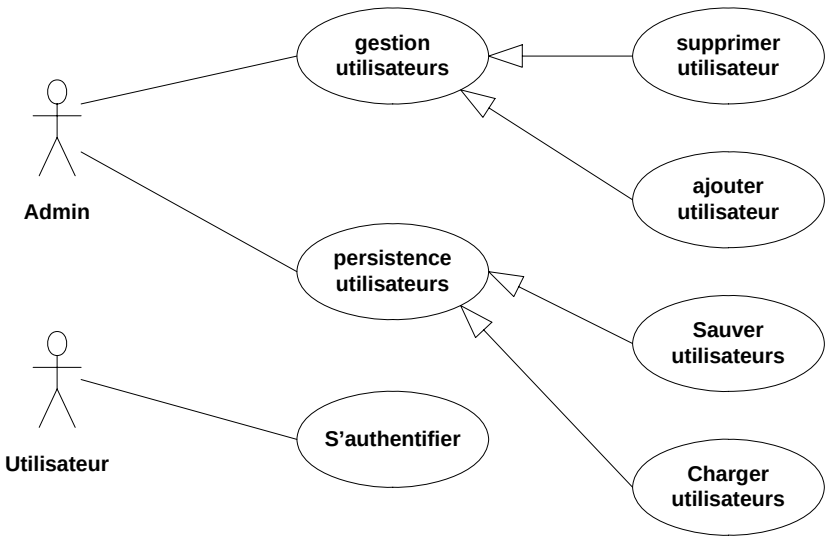


Figure 5 : Les cas d'utilisation du package authentification.

Une première analyse montre l'interaction entre un utilisateur (client) et un serveur d'authentification (authentificateur ou *Authentifier*).

Il nous apparaît également nécessaire de fournir la possibilité de se souvenir des utilisateurs inscrits entre deux sessions. Ceci suppose un mécanisme permettant de sauvegarder dans un fichier une liste d'utilisateurs, et de charger une liste d'utilisateurs depuis un fichier.

Enfin, le package doit fournir un service d'authentification.

Capturons ces services dans une interface, ce qui correspond à appliquer le design pattern Façade : cette interface est la partie publique du package, qui émerge. Elle définit le rôle (*responsabilités*) de ce package vis-à-vis du reste de l'application (voir figure 6).

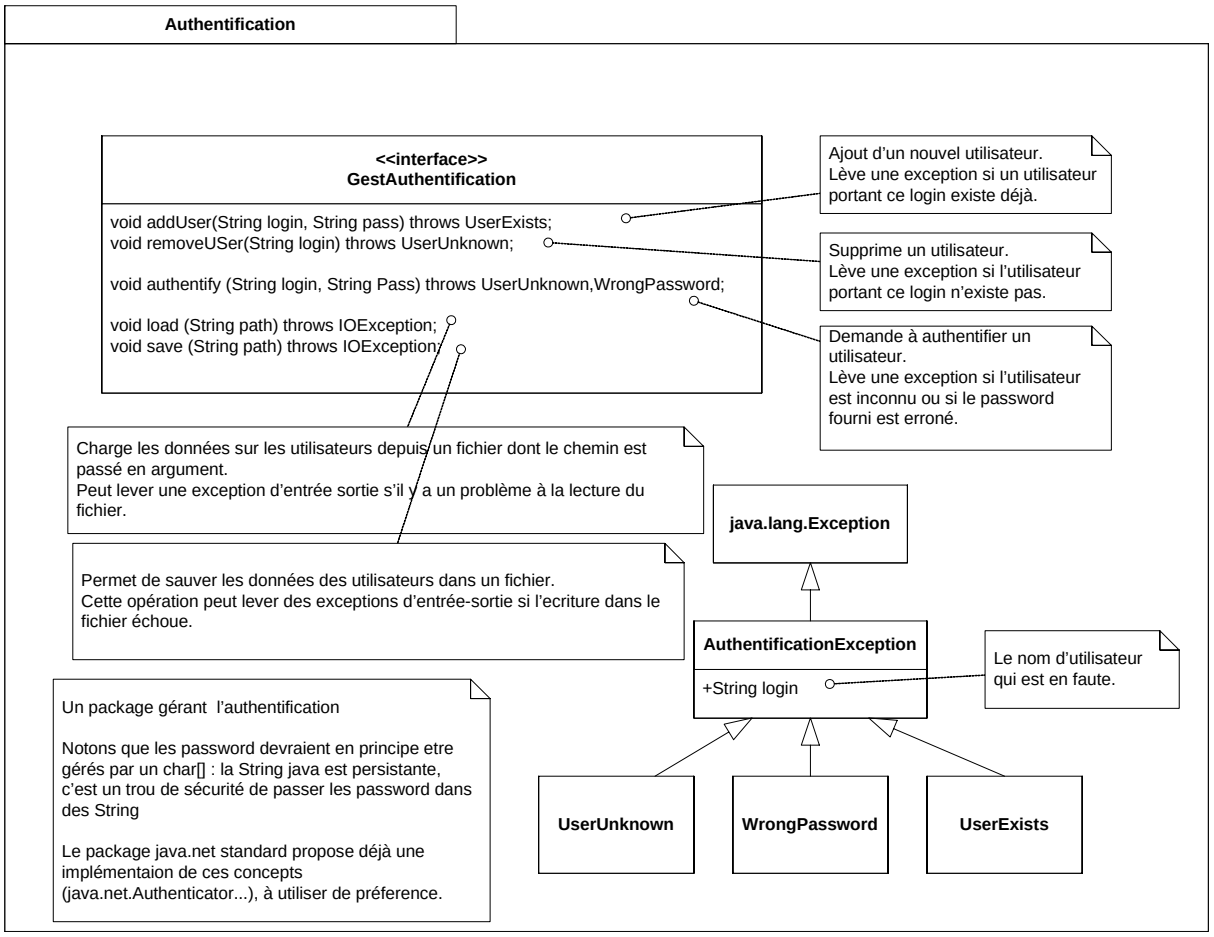


Figure 6 : Partie publique/ Interface du package d'authentification. Les classes décrites fournissent une abstraction de ce package.

On introduit également des classes support simples, qui proposent une implémentation par défaut de cette interface (voir figure 7).

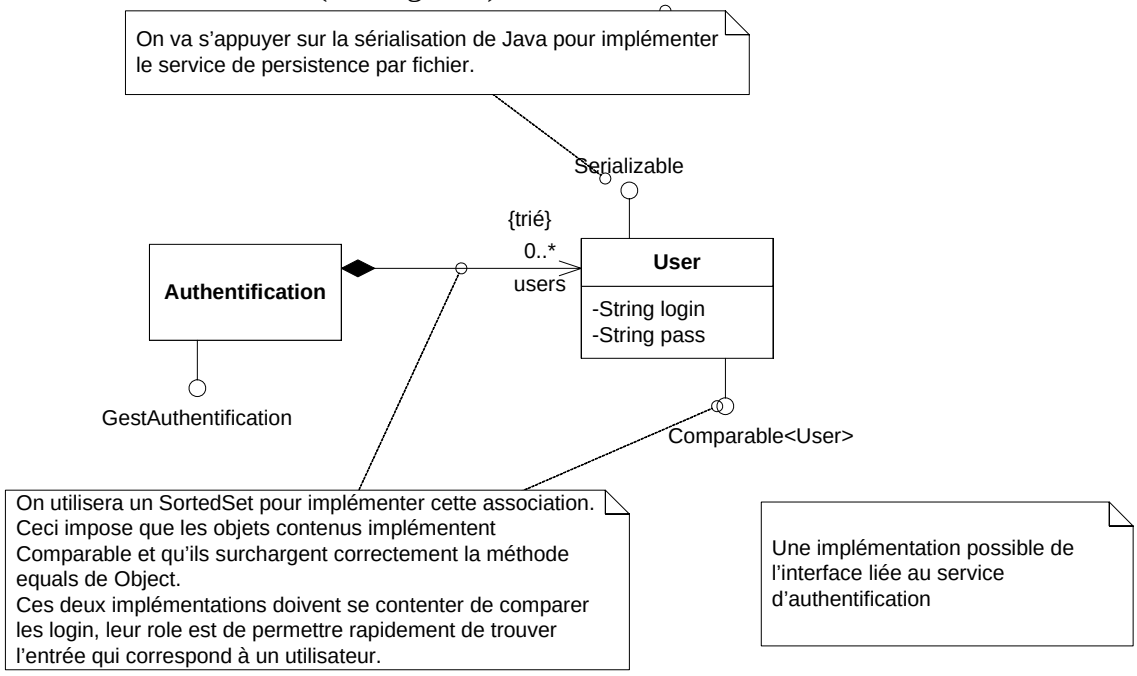


Figure 7 : Une implémentation par défaut du package Authentification.

Travail demandé

Concepts abordés : interfaces de sérialisation et de comparaison, manipulation de collection triée.

Créez ce package, son interface et son implémentation par défaut. Les classes développées devront pouvoir s'utiliser avec le main suivant :

```
public static void main(String[] args) {
            GestAuthentification ga = new Authentification () ;

            // test gestion utilisateurs

      try {
            ga.addUser("bob","123");
            System.out.println("Bob est ajouté !");
            ga.removeUser("bob");
            System.out.println("Bob est retiré !");
            ga.removeUser("bob");
            System.out.println("Bob est retiré  deux fois !");
      } catch (UserUnknown e) {
            System.out.println(e.login +" est inconnu du système! on ne
peut le retirer.");
      } catch (UserExists e) {
            System.out.println(e.login +" est déjà dans le système! on ne
peut le recréer.");
      }

      // test authentification
      try {
            ga.addUser("bob","123");
            System.out.println("Bob est ajouté !");
            ga.authentify("bob","123");
            System.out.println("Bob est authentifié !");
            ga.authentify("bob","456");
            System.out.println("Bob est authentifié avec un autre password
!");
      } catch (WrongPassword e) {
            System.out.println(e.login+" s'est trompé de password !");
      } catch (UserExists e) {
            System.out.println(e.login +" est déjà dans le système! on ne
peut le recréer.");
      } catch (UserUnknown e) {
            System.out.println(e.login +" est inconnu du système! on ne
peut le retirer.");
      }

      // test sauvegarde
      try {
            ga.sauver("users.txt");
            GestAuthentification gb = new Authentification();
            gb.charger("users.txt");
            gb.authentify("bob","123");
            System.out.println("Test sauvegarde/chargement réussi !");
      } catch (UserUnknown e) {
            System.out.println(e.login +" est n'est plus connu. Problème de
sauvegarde/chargement.");
      } catch (WrongPassword e) {
            System.out.println(e.login+" s'est trompé de password !Problème
de sauvegarde/chargement.");
```

```
	} catch (IOException e) {
		System.out.println(e);
	}

}
```

On obtiendra une sortie similaire à celle-ci, quand toutes les fonctionnalités seront implémentées (on peut exécuter le test dès la définition de l'interface GestAuthentification!!):

```
Bob est ajouté !
Bob est retiré !
bob est inconnu du système! on ne peut le retirer.
Bob est ajouté !
Bob est authentifié !
bob s'est trompé de password !
Test sauvegarde/chargement réussi !
```

Notons que java.net définit déjà une version des concepts d'authentification (voir classes java.net.Authenticator et classes liées). Cependant nous avons choisi ici développer nos propres classes, en raison de la simplicité relative des besoins en authentification de notre application.

Pour aller plus loin :
Sauriez-vous implémenter cette interface par une classe réalisant les opérations à travers un accès sur une base de données relationnelles ?
Notons que la méthode « charger » perds son sens, vu la persistance de la base. « Sauver » peut être réalisé par un « commit » sur la base ; l'ajout/suppression par des updates simples (« INSERT INTO », « DELETE FROM »). Enfin « authentify » pourra être réalisé par un « SELECT ».
Le fichier est fourni ici, il suppose l'existence d'une table dans l'espace visible de l'utilisateur.

```
create table Chat_users(
	login  varchar2(20),
	pass  char(6) not null,
	Primary key (login));
```

/** Fichier `AuthentificationOracle.java` */

```
package authentification;
import java.sql.*;
import java.io.*;

public class AuthentificationOracle implements GestAuthentification {

	Connection conn;
	Statement stmt;

	public AuthentificationOracle (String DBlogin,String DBpass) {
		try {
			Class.forName("oracle.jdbc.driver.OracleDriver");
		 } catch (ClassNotFoundException e) {
			System.err.println("Could not load the driver");
		 }
		 try {
				conn = DriverManager.getConnection
					("jdbc:oracle:thin:@ten10:1521:acdb", DBlogin,
DBpass);
```

```
                                                // @machineName:port:SID,
userid,  password

                        stmt = conn.createStatement();
                } catch (SQLException e) {
                    e.printStackTrace();
                    System.err.println("Could not connect to DB"+e);

                    }
      }

      public void close () {
            try {
                    stmt.close();
                    conn.close();
            } catch (SQLException e) {
                    e.printStackTrace();
            }
      }

      public void addUser(String login, String pass) throws UserExists {
            try {
                    stmt.executeUpdate("INSERT INTO chat_users(login,pass)"+
                                                "VALUES ('"+login +"','"+
pass + "')");
            } catch (SQLException e) {
                    // TODO Auto-generated catch block
                    e.printStackTrace();
            }
      }

      public void removeUser(String login) throws UserUnknown {
            try {
                    stmt.executeUpdate("DELETE from chat_users where login =
'"+login+"'");
            } catch (SQLException e) {
                    // TODO Auto-generated catch block
                    e.printStackTrace();
            }
      }

      public void authentify(String login, String Pass) throws UserUnknown,
WrongPassword {
            try {
                    ResultSet rset = stmt.executeQuery
                    ("SELECT s.login, s.pass " +
                     "FROM chat_users s " +
                     "WHERE s.pass = '"+Pass + "' and s.login = '" +login+
"';");
                    if (rset.next()) {
                          // print example :
                          // System.out.println(rset.getString(1) + ", " +
rset.getString(2));
                          System.out.println("User "+rset.getString(1)+ "is
authentified.");
                    } // else no result !! bad user !
            } catch (Exception e) {
                    e.printStackTrace();
            }
            throw new UserUnknown(login);
    }
```

```
	public void charger(String path) throws IOException {
		// nothing to do
	}

	public void sauver(String path) throws IOException {

		try {
			stmt.execute("commit;");
		} catch (SQLException e) {
			e.printStackTrace();
			throw new IOException("DB access failed !"+e);
		}
	}

}
```

Séance 3 : Le package Net

Ce package est chargé de définir et d'encapsuler les parties afférentes à l'utilisation du réseau.

Nous allons implémenter un modèle **client**/**serveur** simple s'appuyant sur des échanges de **messages**. Il nous faut de plus pouvoir traiter plusieurs clients simultanément.

Pour cela nous définissons tout d'abord la notion de Message, contenant un champ Header (en-tête du message) et un champ data pour les données véhiculées. Une *ConnectionMessage* permet d'émettre et de recevoir des Message. La réception et l'émission sont des opérations potentiellement bloquantes : la lecture est bloquante quand il n'y a pas de message à lire, l'écriture est bloquante si la capacité de la connection est atteinte (le tube est plein).

Ensuite nous introduisons la notion de serveur TCP multi-threadé, et de client TCP.

Rappels : Fonctionnement en mode connecté TCP.

La solution retenue s'appuiera sur le protocole TCP, donc en mode connecté. Ce type de connexion persistante permet d'assurer l'absence de pertes ou d'erreurs dans la transmission des données.
Un modèle s'appuyant sur UDP multicast pour la diffusion des posts dans une chatroom pourrait être envisagé, mais l'on perdrait alors le contrôle de la chatroom, car on ne peut pas contrôler qui envoie vers une adresse de multicast (adresses IP de classe D). Ce mode non-connecté offre cependant en théorie de meilleures performances grâce à une meilleure utilisation de la bande passante disponible, mais peut provoquer des pertes de message.
Notons que si notre conception est solide, nous devrions pouvoir adopter cet autre mode de communication en ne réalisant que des modifications bien localisées.

En TCP l'établissement de la connexion est asymétrique :

- Le serveur crée une socket dite « d'écoute », sur un port bien connu des clients (80 pour un serveur http, 21 pour un serveur ftp) ensuite il attends (*accept()*) qu'un client se connecte. Quand un client effectivement se connecte, une nouvelle socket dite « de communication » est créée et rendue (valeur de retour) par le accept. Cette socket est connectée au client, on peut écrire et lire dedans comme dans un flux (fonctionnemnt de « pipe » unix, muni d'une capacité).

- Le client crée une socket et essaie de se connecter au serveur, dont il doit connaître l'adresse IP et le port bien connu. Si le serveur est en écoute, la socket se connecte, et on peut écrire et lire dedans de façon symétrique au serveur.

Passé l'établissement de la connection, les deux entités sont équivalentes (plus de logique client/serveur), c'est le protocole qui détermine dans quel sens circulent les messages.

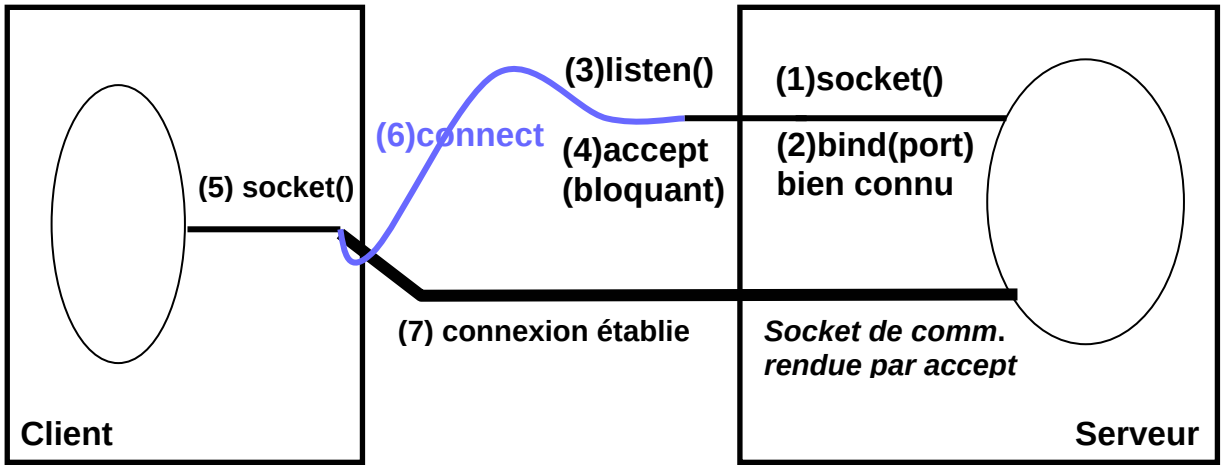


Figure 8 : Connection TCP, étapes de l'établissement de connection. Notons qu'en Java les événements numérotés de 1 à 3 sont directement assurées par le constructeur de ServerSocket. (non UML)

Ce comportement nécessite un ajustement (voir figure 9) pour permettre de traiter plusieurs clients simultanément : il faut traiter la connexion (échanges sur la socket de communication) de chaque client dans un nouveau thread ou processus (crée à la volée, par un appel de type fork()), pendant que le thread initial reste dédié à l'attente de connexions, et se remet donc en attente (accept) sur la socket d'écoute.

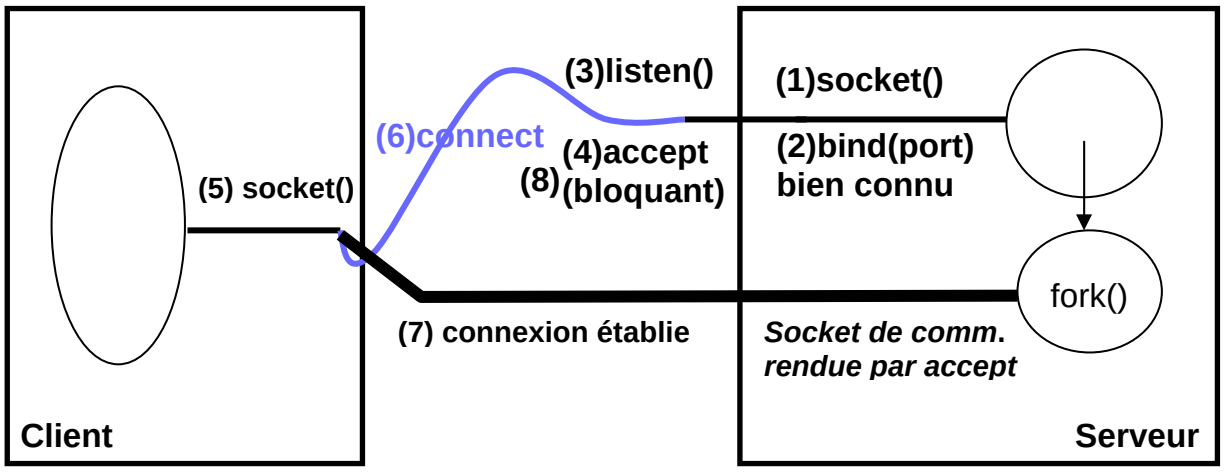


Figure 9 : Solution multi-threadée pour traiter plusieurs clients simultanément. Chaque client est géré par un nouveau thread, un thread dédié à l'attente boucle infiniment sur accept (étapes 4 et 8). (non UML)

Ligne d'implémentation proposée

Ce comportement d'un serveur multi-threadé est suffisamment complexe pour mériter d'être isolé, et traité d'une façon réutilisable dans nos applications. Nous allons développer une classe abstraite qui permet de gérer l'ensemble de ce protocole de mise en place d'une connexion. Nous proposons de nous appuyer sur le squelette de comportement suivant (figure 10):

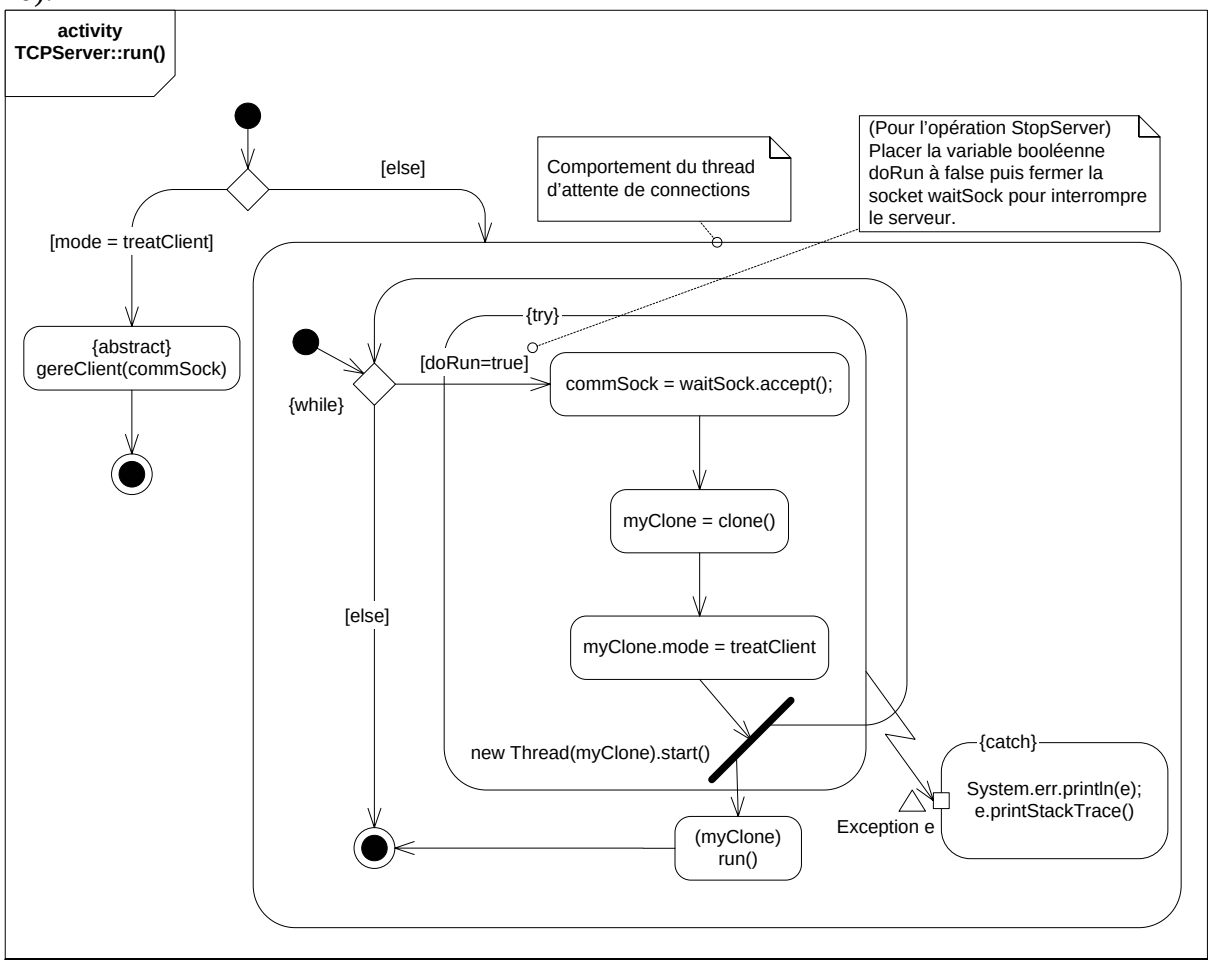


Figure 10 : Diagramme d'activité exprimant le déroulement de la méthode *run()* du serveur TCP multi-threadé abstrait.

L'intérêt de ce fonctionnement avec clonage de l'objet courant, est que les différentes connections qui sont établies avec les clients partageront naturellement les références sur objets que possède le thread d'attente. Par exemple, si le serveur d'attente (ou plutôt sa classe dérivée qui implémente la méthode abstraite « gereClient ») possède une référence sur un gestionnaire d'authentification, chaque connexion client aura accès à ce même gestionnaire.

Le diagramme suivant (Figure 11) résume le contenu du package net. Nous avons opté pour une classe Message relativement simple, on pourrait l'étendre pour plus de flexibilité.

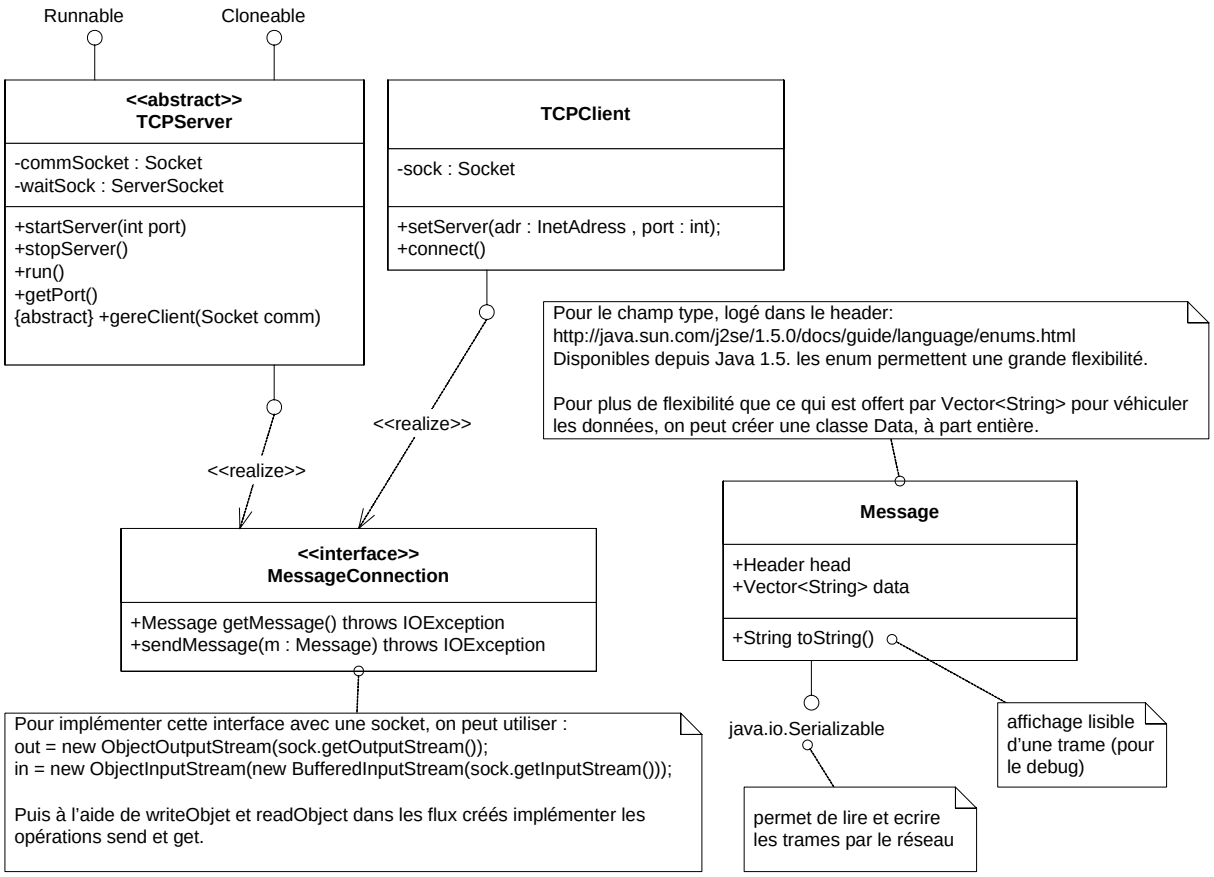


Figure 11 : Contenu du package net.

Travail demandé

Concepts abordés : interfaces de sérialisation, classe abstraite, exceptions, manipulation de Socket.

Réalisez ce package. On pourra tester le serveur à l'aide d'un service (dérivé de TCPServer et implémentant gereClient) simple, qui se contente d'afficher les messages reçus dans une boucle infinie d'attente sur getMessage(), et d'un ou plusieurs clients qui envoient des trames au serveur.

```
public class BasicTCPServer extends TCPServer {

	@Override
	public void gereClient() {
		System.out.println(getMessage());
		sendMessage(new Message(Header.DEBUG,"pong from server"));
		System.out.println("Server finished handling client,
quitting.");
	}

	public static void main(String[] args) {
		BasicTCPServer serv = new BasicTCPServer();

		try {
			serv.startServeur(1664);
			System.out.println("Press enter to quit...");
```

```
                System.in.read();
                serv.stopServeur();
            } catch (Exception e) {
                e.printStackTrace();
            }
      }
}
```

```
Un client simple de test :
public class BasicTCPClient {
      public static void main(String[] args) {
            TCPClient cli = new TCPClient();
            try {
                cli.setServer(InetAddress.getLocalHost(), 1664);
                cli.connect();
                cli.sendMessage(new Message(Header.DEBUG,"Ping from
client !"));
                System.out.println(cli.getMessage());
                cli.disconnect();
            } catch (Exception e) {
                e.printStackTrace();
            }

      }
}
```

Séance 4 : Assemblage de composants.

Il est temps de préciser l'infrastructure de notre application de Chat globale, c'est-à-dire munie de composants client et d'un composant serveur, déployés sur des machines à priori différentes. Nous propososns de découper en package server et client. Côté serveur, le chatter est distant, côté client, le gestionnaire de topics et les chatroom sont distants.
Nous allons appliquer le design pattern Proxy, pour cacher cette répartition : un composant local va jouer le rôle du composant distant, en répercutant les appels sur une entité distante.

Il faut également à présent s'intéresser au protocole réseau (de niveau application) à mettre en place. On va créer une trame par méthode distante à appeler, et une trame pour loger la valeur de retour quand celle-ci existe. Notons que l'utilisation sous-jacente de TCP assure la fiabilité de la connexion, il est donc inutile d'acquitter les messages reçus, sauf si la réponse porte de l'information.

Au niveau de la gestion des connexions, il existe plusieurs approches. Notons que :

- Le gestionnaire de topics joue un rôle serveur dans les interactions
- Une chatroom joue un rôle serveur dans les interactions
- Un chatter joue également un rôle serveur, puisqu'il offre son opération recevoirMessage() aux chatroom.

Package Serveur.

On propose de créer un processus serveur par chatroom (sur des ports distinct), et un processus serveur gérant les topics sur un port « bien connu ».

Pour le gestionnaire de topics (figure 12), il nous faut trois trames : pour lister les topics (+réponse contenant une liste de noms de topics), pour créer un topic, et pour rejoindre un topic (+réponse contenant le numéro de port associé au processus serveur correspondant à la chatroom souhaitée). Le comportement de la méthode gereClient du serveur de gestion topics est donc de répondre à ces différentes trames de façon appropriée (oublions dans un premier temps l'authentification).

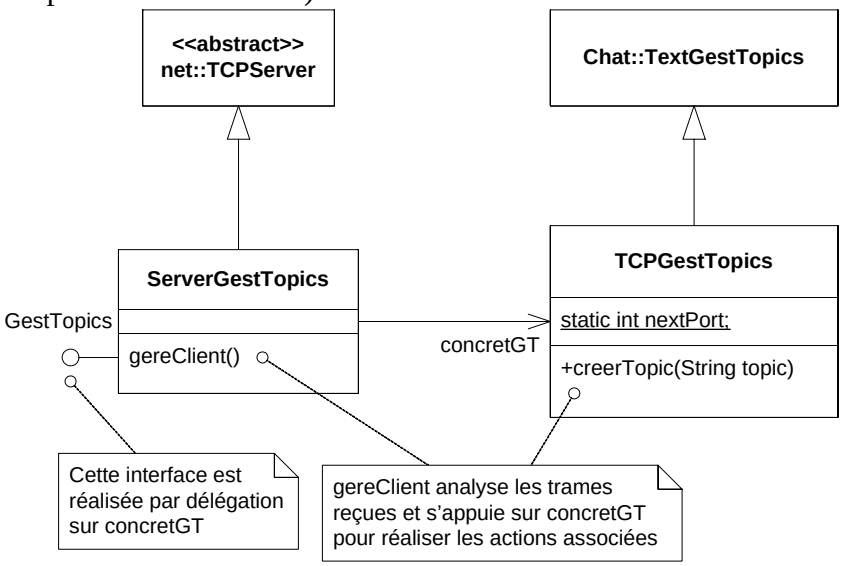


Figure 12 : Instanciation avec gestion du réseau du Gestionnaire de topics. Le ServerTopicManager pourrait se passer d'implémenter TopicManager, cela permet simplemnt une meilleure homogénéité de la solution.

Pour la gestion des chatroom (figure 13) on va utiliser un schéma analogue. La seule différence essentielle est que notre serveur de chatroom va implémenter l'interface Chatter, pour satisfaire les besoins de la chatroom concrète.

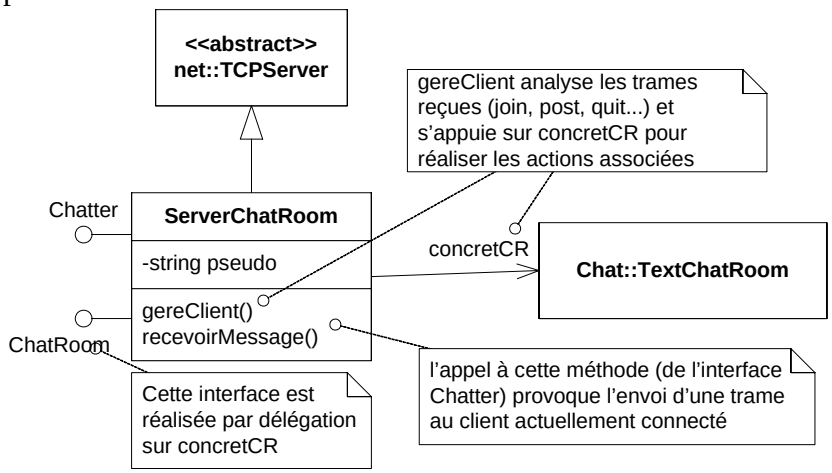


Figure 13 : Instanciation de la partie ChatRoom du serveur. Le ServerChatRoom implémente ChatRoom pour plus d'homogénéité et pour permettre son utilisation dans le TextTopicManager (qui contient des références de ChatRoom).

Package Client

Côté client, on va implémenter l'interface de gestion topics et chat room par des clients réseau.
Pour le gestionnaire de topics, les appels à lister topics et créer topics sont implémentés par l'envoi de trames réseau au serveur. L'appel à rejoindreTopic doit questionner le serveur pour obtenir le numéro de port de la chatroom appropriée, et instancier un ClientChatRoom et le connecter à la chatroom serveur.
Pour le gestionnaire de chatroom, on simplifie le comportement de la chatroom générale : la chatroom client ne gère qu'un unique Chatter.
Le ClientChatRoom instancie de plus un Thread d'écoute réseau pour attendre les messages (trame réseau correspondant à recevoirMessage) provenant de la chatroom serveur, et les répercuter sur son Chatter. Ce Thread sera créé suite à un appel à « rejoindre » et interrompu suite à un appel à « quitter ».
Dans un premier temps on pourra utiliser un simple TextChatter pour tester cette architecture.

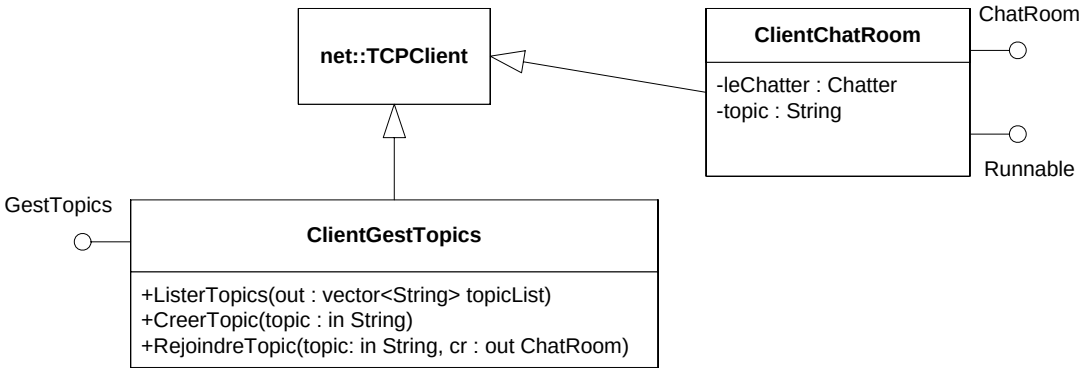


Figure 14 : Les classes du package client.

Interactions Client-Serveur par proxies

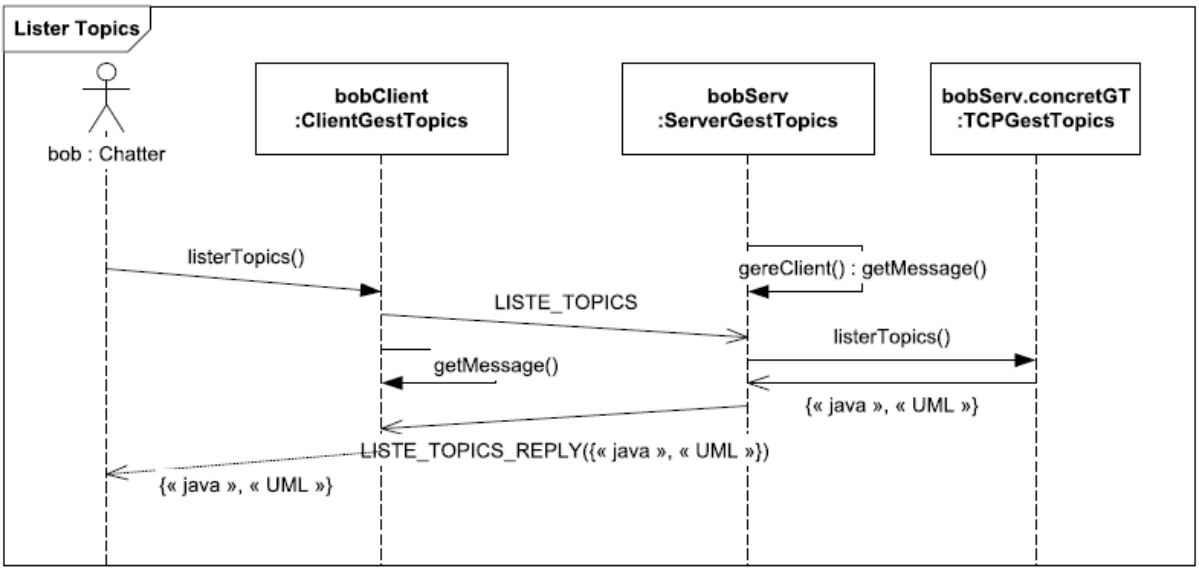


Figure 15 : Interaction par proxy avec le gestionnaire de topics pour lister les topics ouverts.

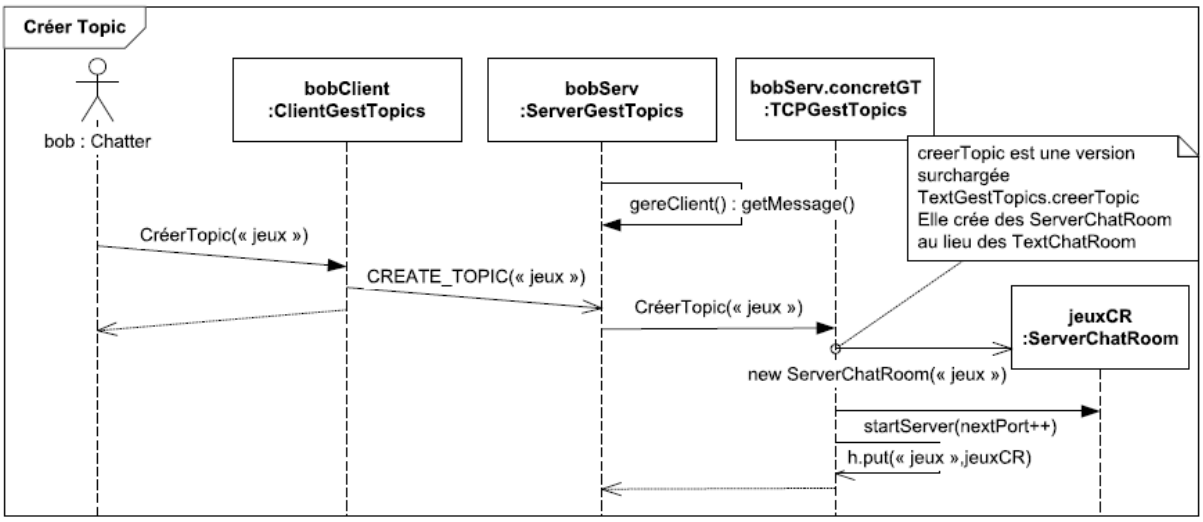


Figure 16 : Interaction par proxy avec le gestTopic pour la création d'une chatRoom

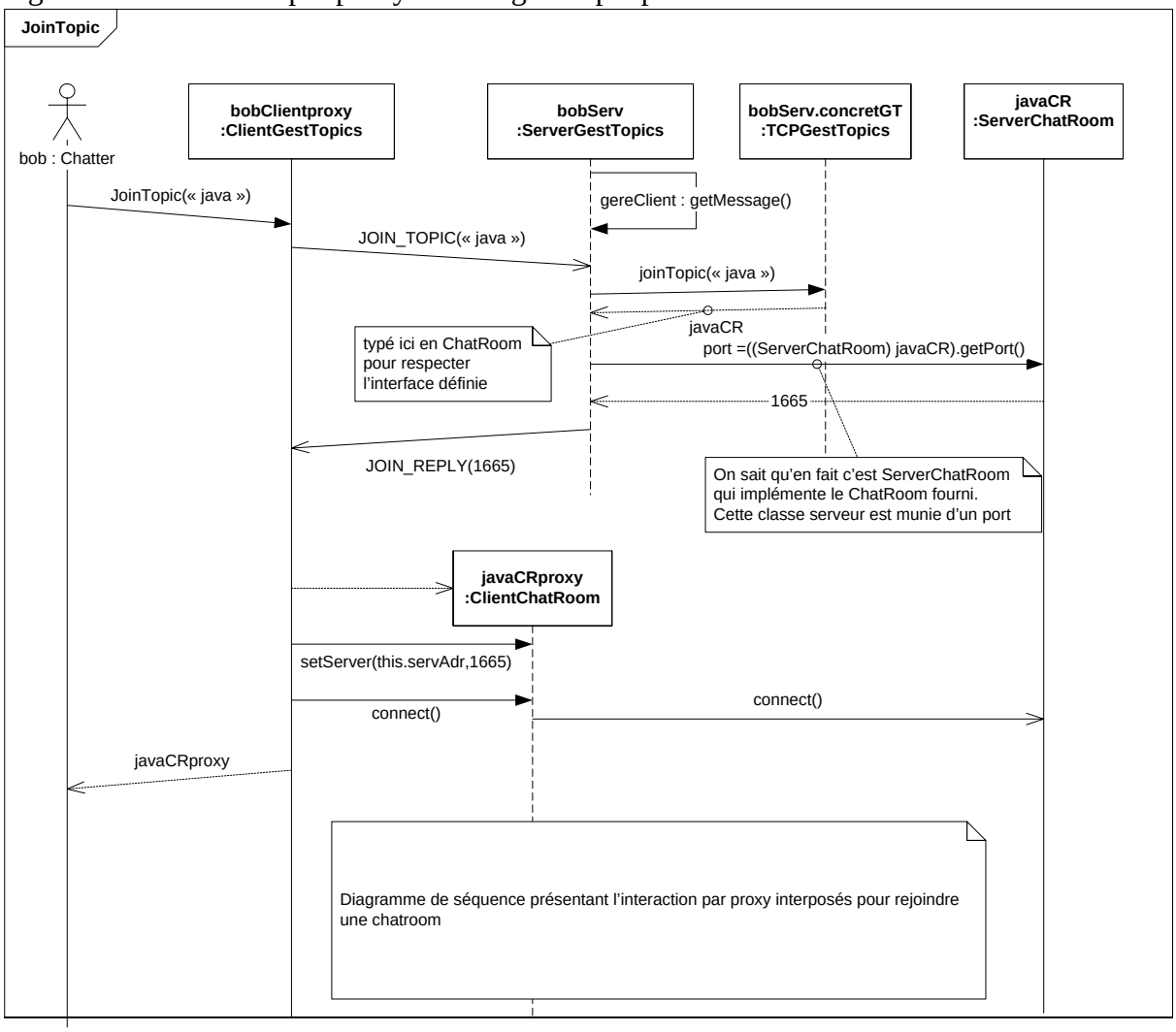


Figure 17 : Interaction par proxy avec le gestTopic pour la création d'une chatRoom côté client

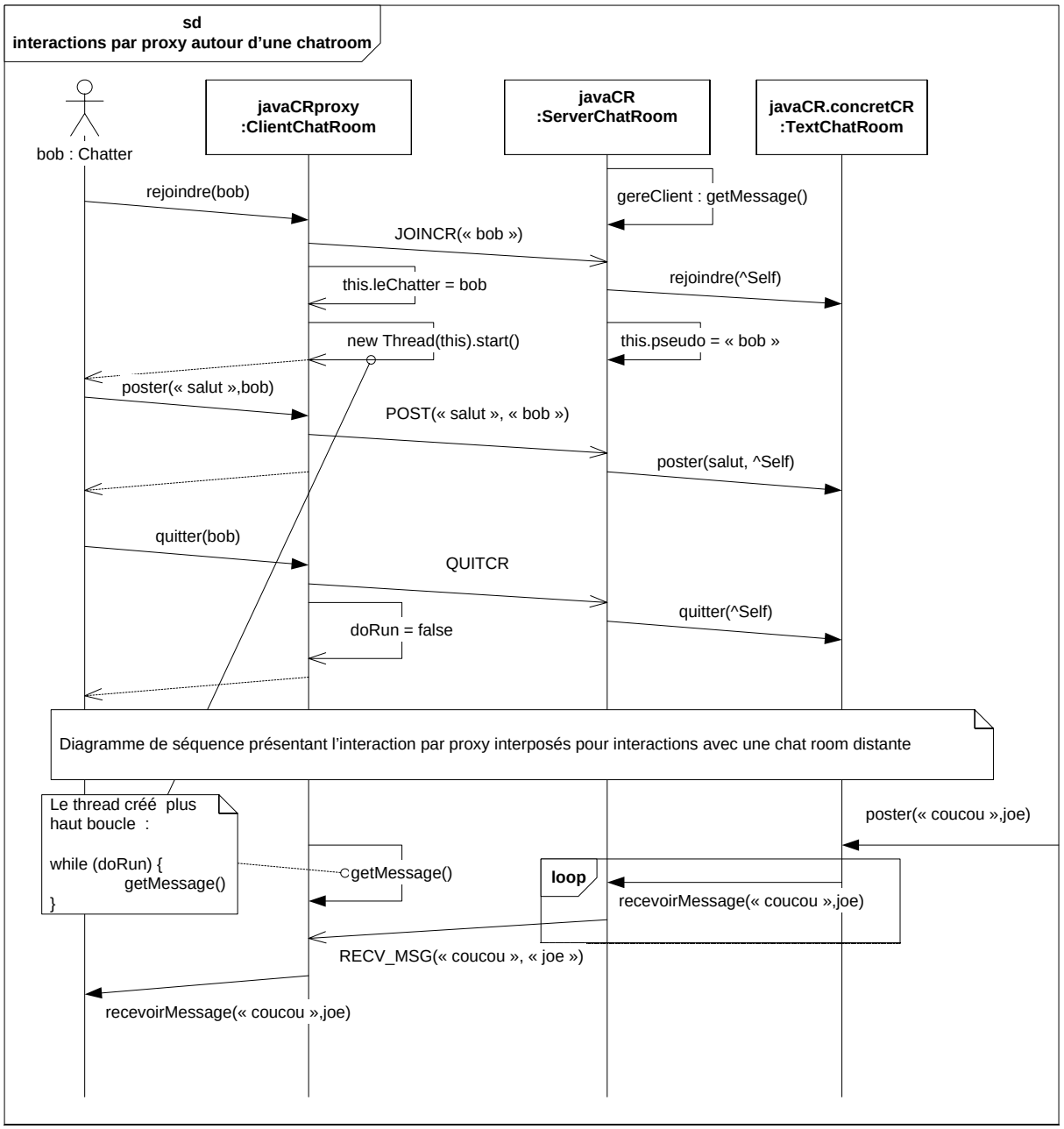


Figure 18 : Interactions réseau avec la chat room.

Travail demandé

Définissez les trames réseau permettant la mise en place de ce protocole de communication. Implémentez cette architecture.

Séance 5 : Une IHM Client en Swing.

On va implémenter à présent une IHM graphique pour le client de notre service de Chat.

On propose l'architecture suivante :

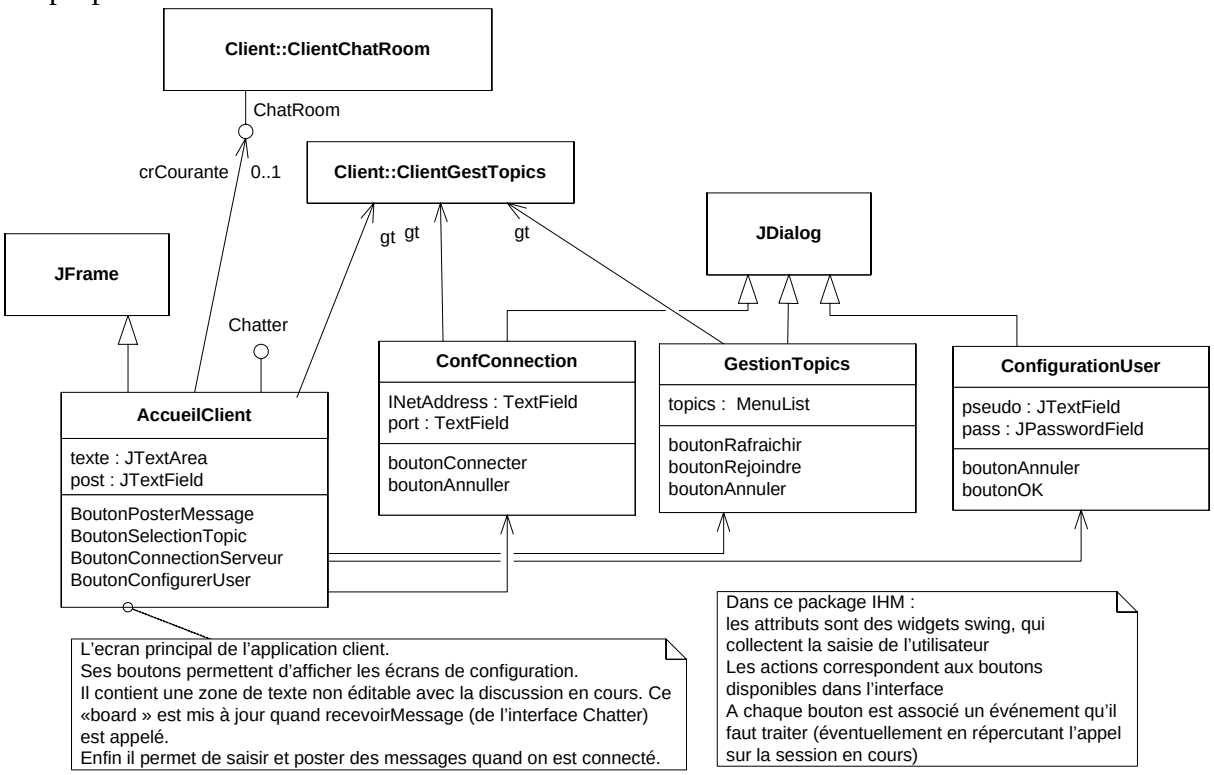


Figure 15 : IHM client : architecture générale.

A la création de la page d'accueil, on instancie les différentes pages de dialogue et on crée une instance de ClientTopicManager, qui sera connue des différents écrans de configuration.

On va considérer qu'on ne peut être connecté qu'à une unique chatroom à la fois, pour simplifier. L'extension pour pouvoir se connecter à plusieurs chatrooms simultanément est assez simple.

Vous pourrez ajouter des références des pages de dialogue vers la page d'accueil si cela vous parait utile.

Cette structure n'est qu'indicative : libre à vous de donner l'aspect et l'organisation que vous souhaitez à votre application, à condition que les fonctionnalités essentielles de l'application soient présentes.