Nicolas FLASQUE
Yann-Thierry MIEG
Benoît CHARROUX

# Java

## Abstract

Scalability and reusability are the main benefits of the object paradigm. These properties, to become effective have to be planned as soon as possible at the beginning of a project. Many steps compose a project. Here are the main steps :

- needs analysis and system specifications ;
- system analysis ;
- system conception ;
- system implementation ;

The aim of this case studies is, given a chat application analysis and a corresponding conception, to implement a solution in the Java language.

To learn more about the Java language :

http://java.sun.com/javase/6/docs/api/

The analysis and the conception use the UML language.

The rest of this document contains two parts :

- system analysis ;
- system conception.

Advice : we recommend careful study of this document ; you will learn how to apply object-oriented concepts to many stages of the software development life cycle.

# A chat application : system analysis

Typically, an analysis is composed of the stages below :

- domain analysis ;
- application analysis ;
- results comparison.

The first two steps are independent and can be done concurrently.

## 1.1. Application analysis

Application analysis focuses on a particular chat application. It concerns especially functional analysis.

The figure below shows a use case diagram.

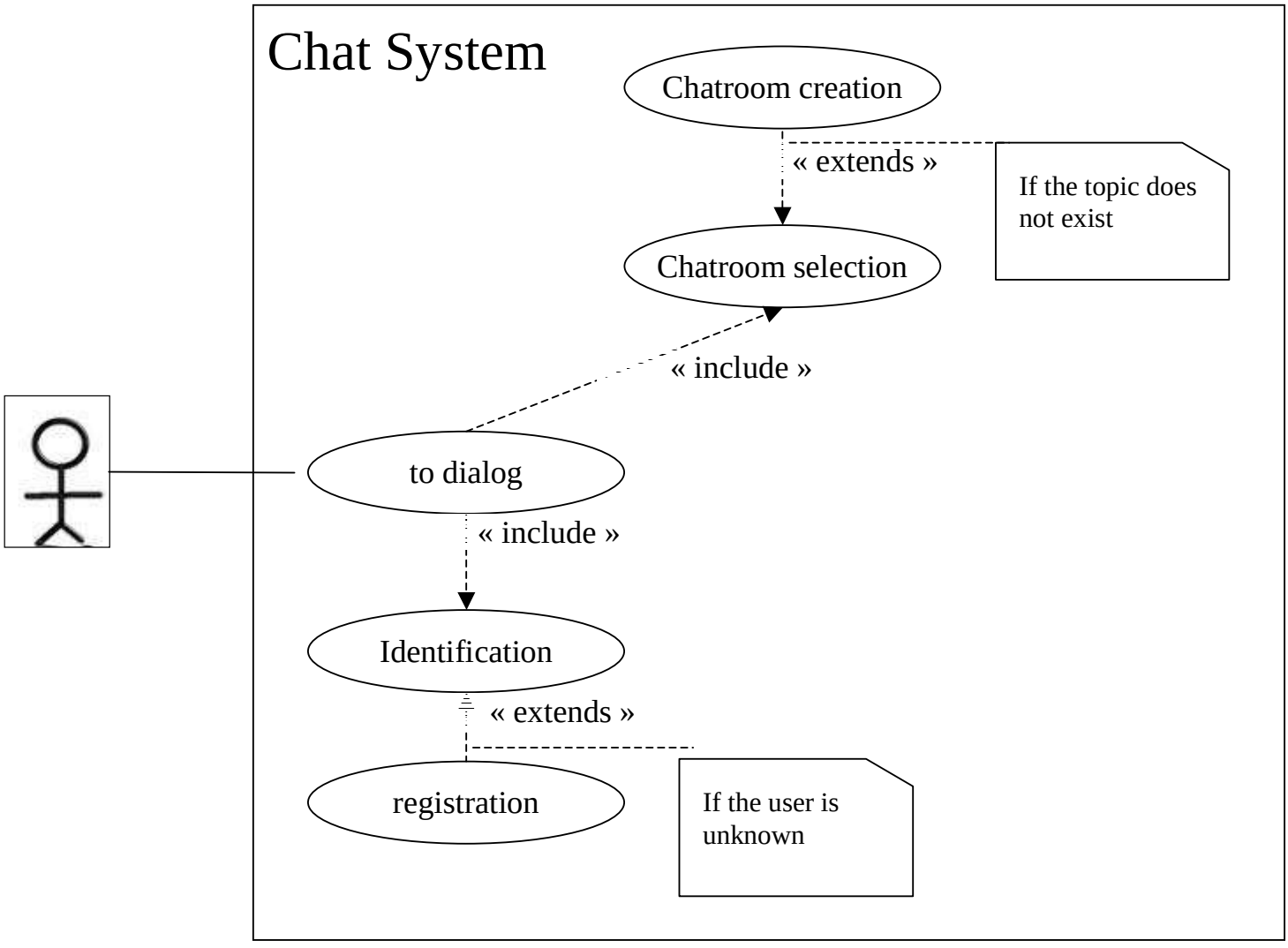



Figure 1 : use case diagram of the Chat System

The activity diagram depicted below describes the main functions of the application.

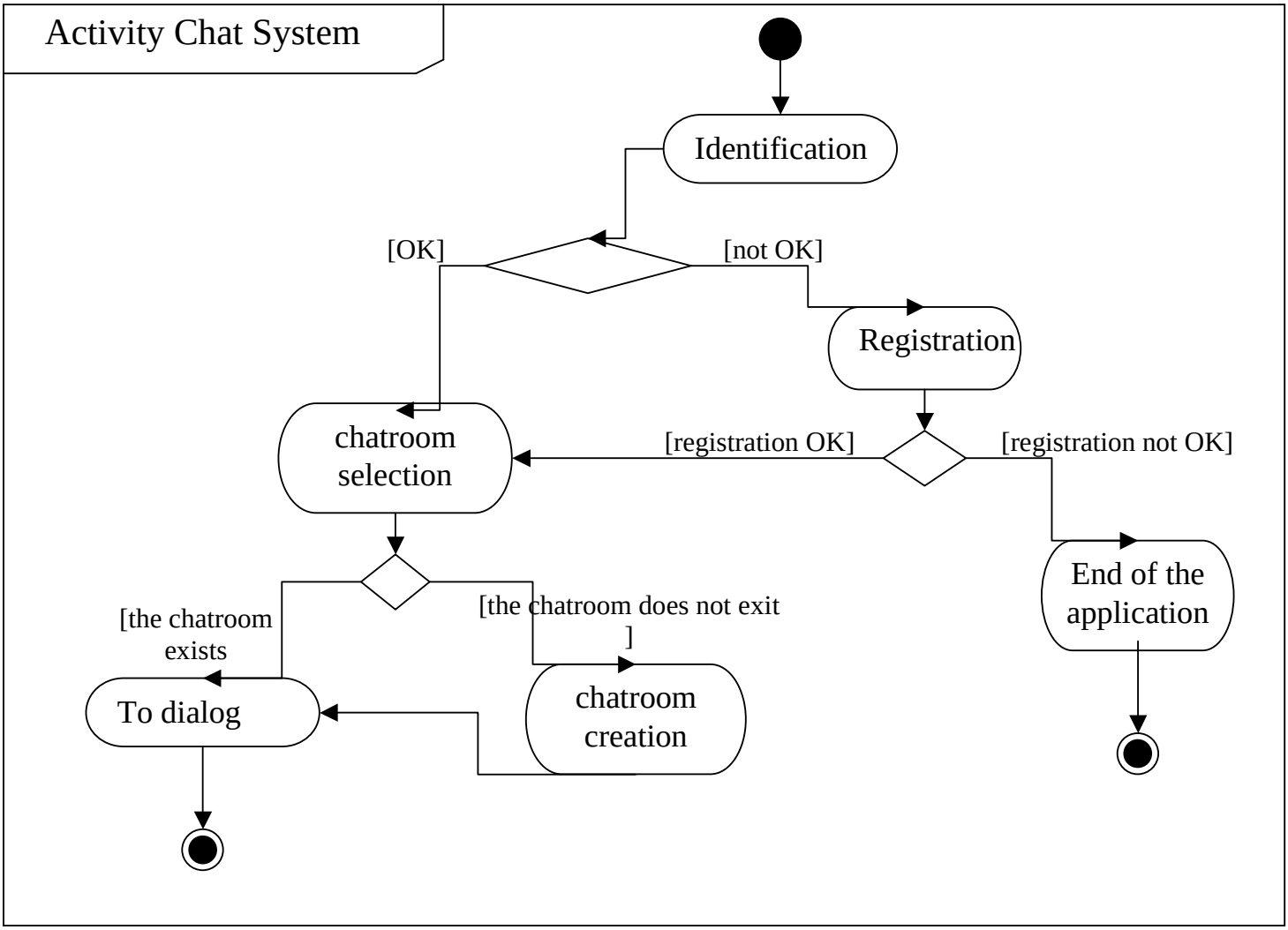



Figure 2 : activity diagram of the Chat System

## 1.2. Domain analysis

Domain analysis concern the chat domain whit out taking into account any particular application. The class diagram below is able to support any chat application.

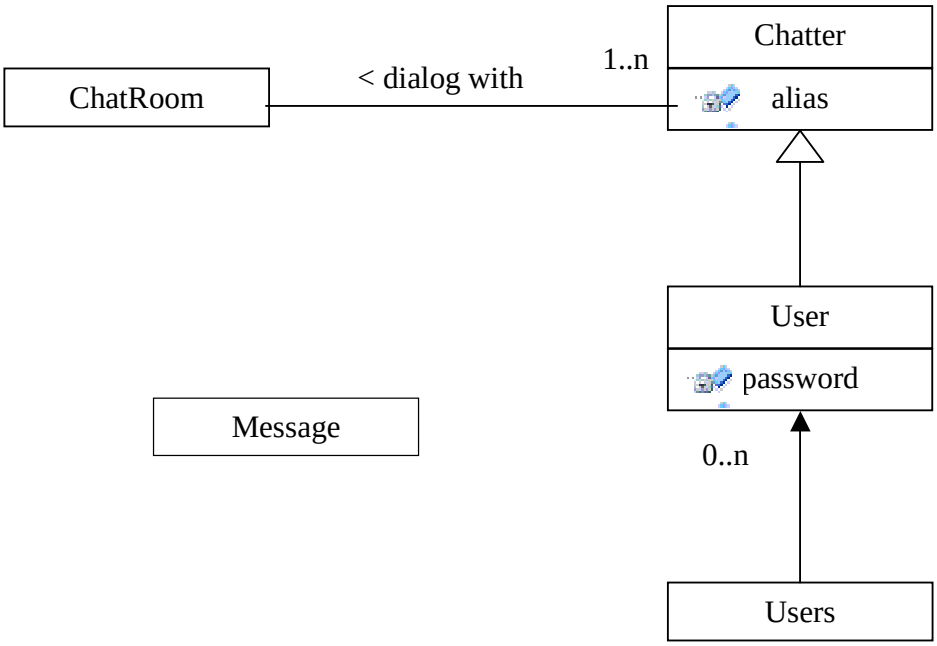



Figure 3 : domain class diagram

## 1.3. Results comparison

The following figures show how to use classes from the domain class diagram to realize the use cases.

.

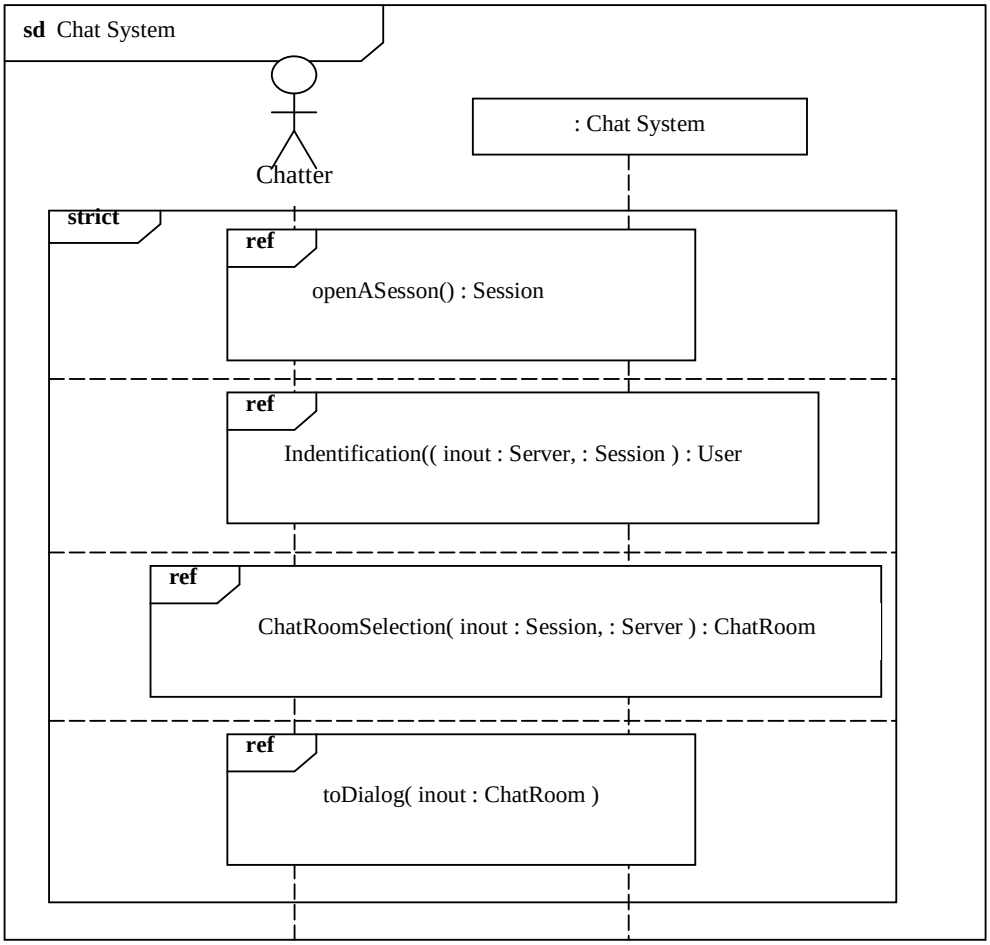



Figure 4 : sequence diagram for the main functions

The addition of new classes (Controller classes) enables the domain analysis and the application analysis to be linked. These classes are parts of a an MVC model (Model View Controller) for the Chat Application widely used in object oriented programming

The figure below shows the opening of a session. Two new controller classes are introduced (Server et Session). Figure 6 shows the interaction between these new classes and the other existing classes.

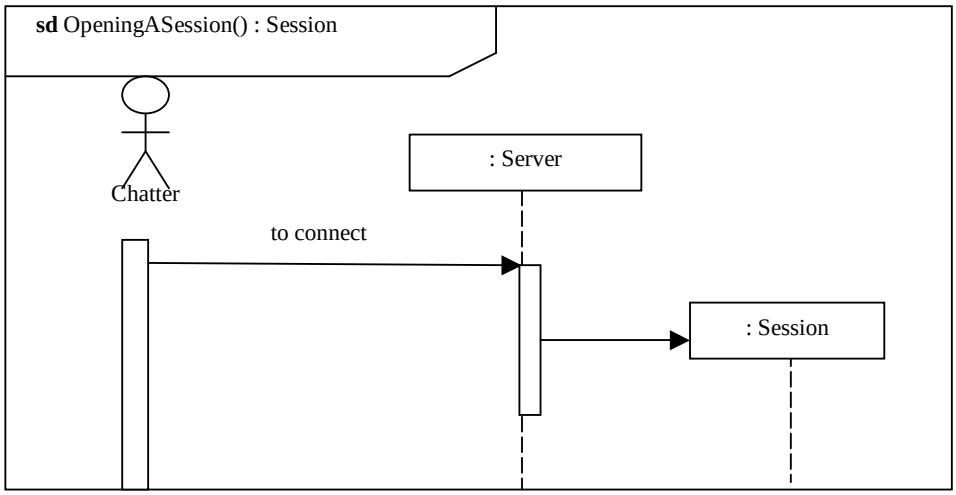



Figure 5 : opening a session

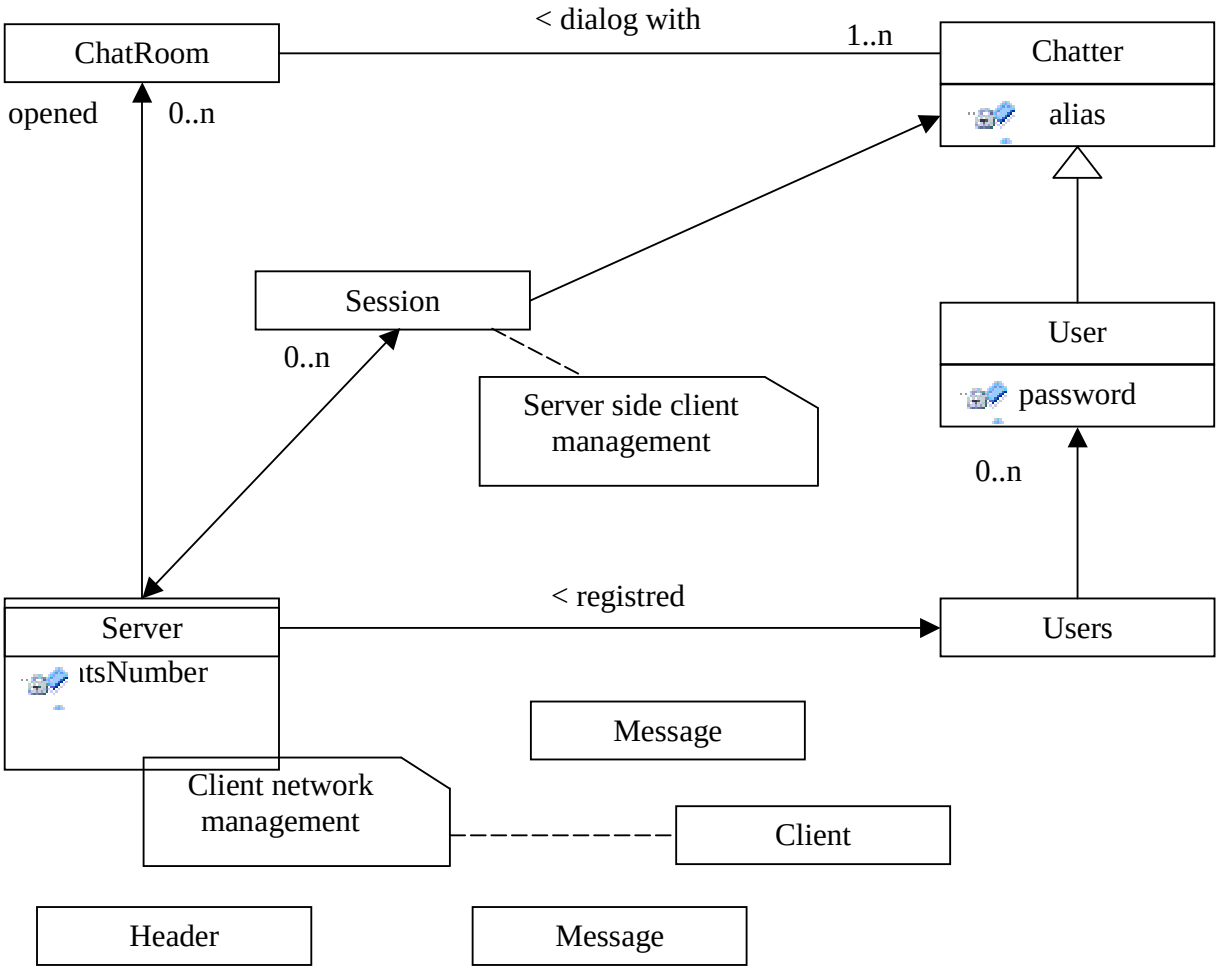



Figure 6 : class diagram with controller classes

Figures 7 and 8 detail the diagram depicted in figure 4.

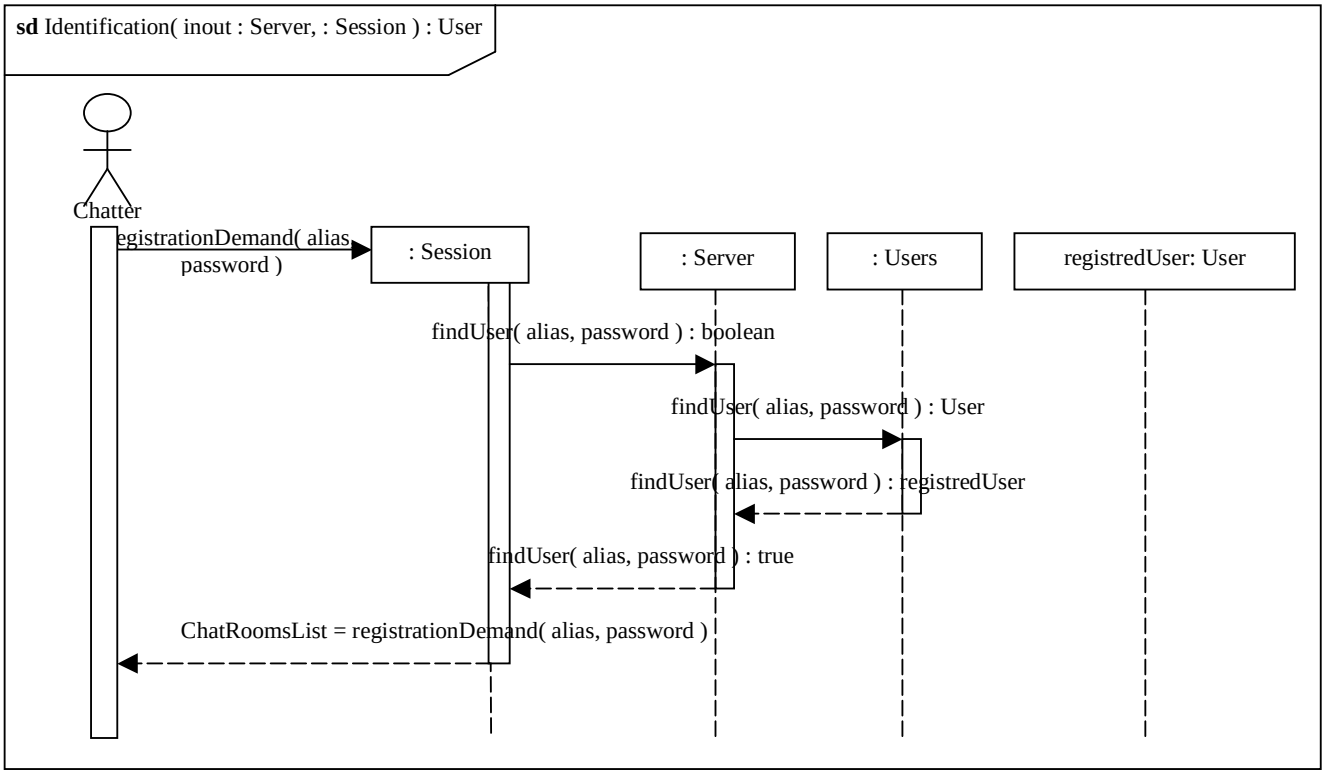



Figure 7 : identification of a chatter

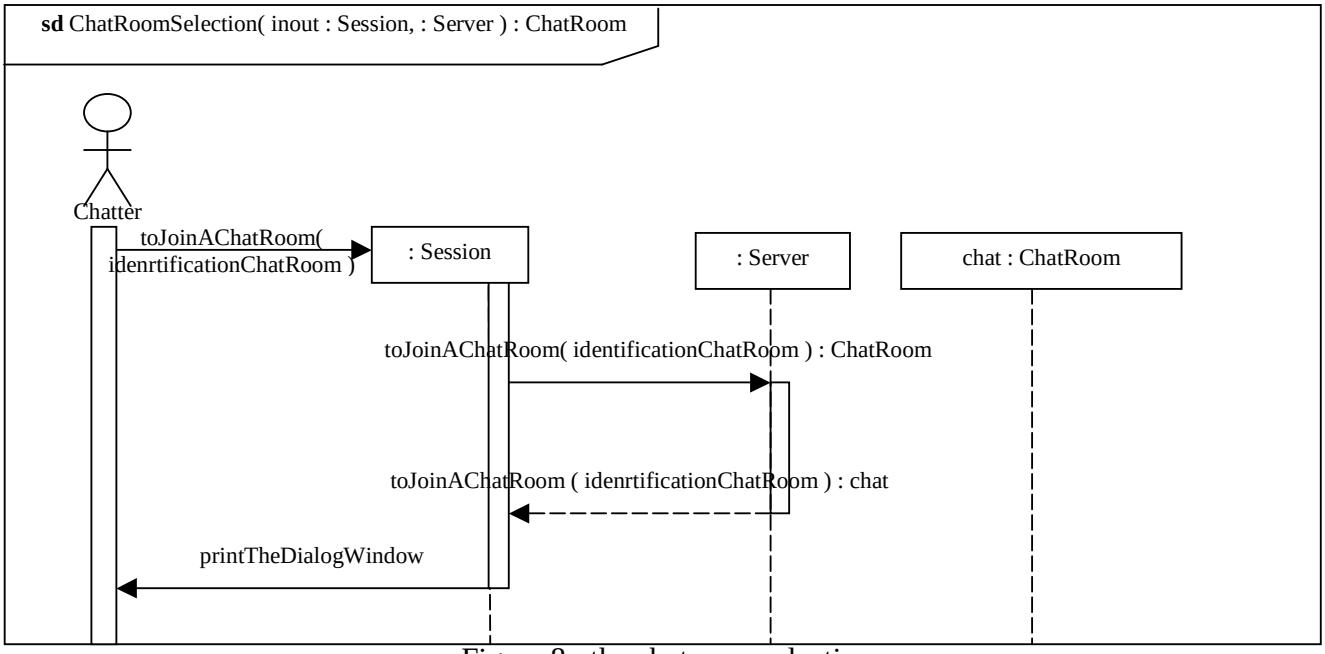



Figure 8 : the chatroom selection

The object diagram below details how to dialog (see figure 4).

.

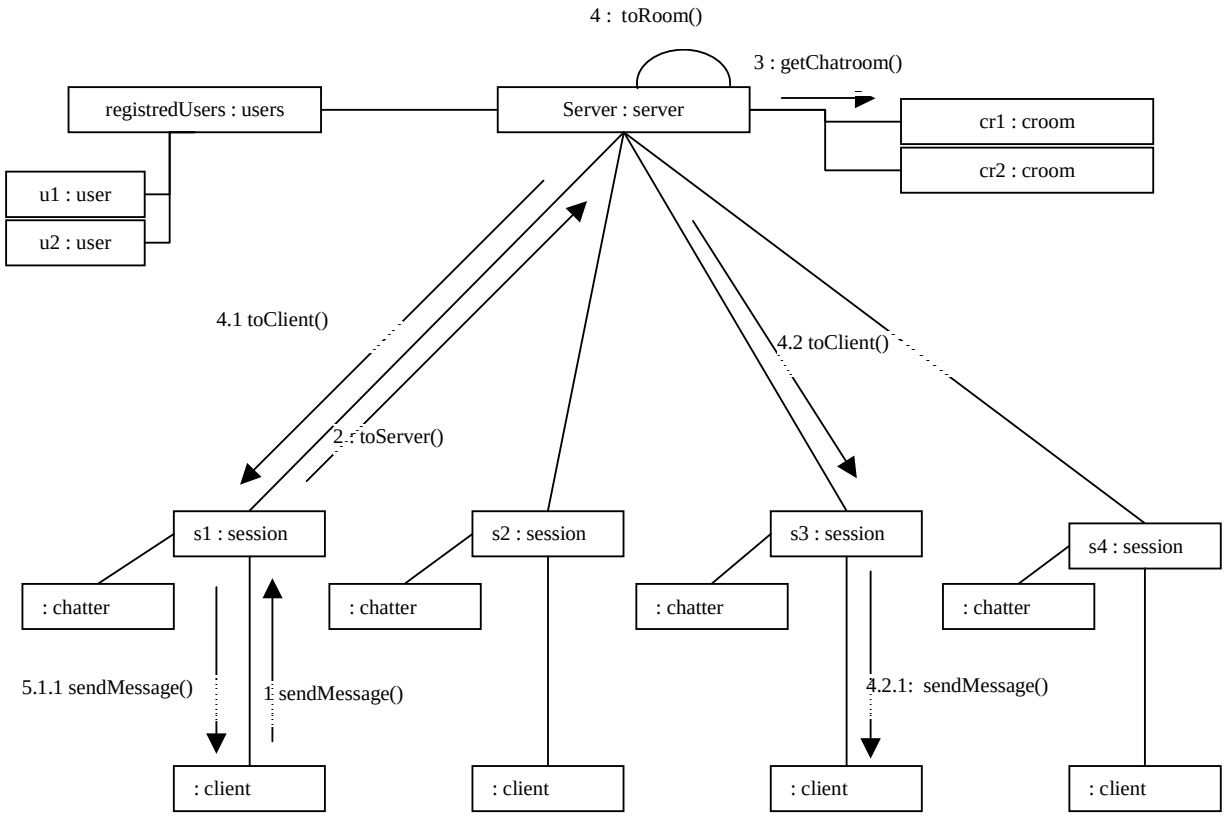



Figure 9 : an object diagram to show how a dialog takes place

# The Conception

The analysis models address domain and functional analyses : based on the Model View Controller modelisation, it shows how Controller classes realize the use cases, while the application domain, the most stable part of a system, is modelised by a class diagram.

The next stage of development is concerned with the partition of the class diagram into packages. The classes in one package can be accessed through an Application Program Interface composed of one or many interfaces. Thus the internal view of a package (the implementation) and the external view (the package contract specified by the interfaces) can be separated.

## *Step 1 : Partition into package.*

Figure 4 (see the analysis above) shows sequences of significance :

- *Opening a session* points out the importance of the network in a Chat application. Many classes deal with the network managment. All these classes are grouped into 3 packages : *Net*, C*lient* and S*erver*. Note that C*lient* and S*erver* have been introduced to separate the application client view from the application server view.
- The identification use classes are grouped in the package A*uthentification*.
- The chat room selection requires a dialog topic manager to find and create chat rooms. The related classes are grouped into the package *Topics*. *Topics* is linked with the package *Chatter* through unidirectional association.
- The *Dialog* package points out the strong interaction between the chat rooms and the chatters (the association between the classes ChatRoom and Chatter is bidirectional).

The chart below indicates dependence between packages.

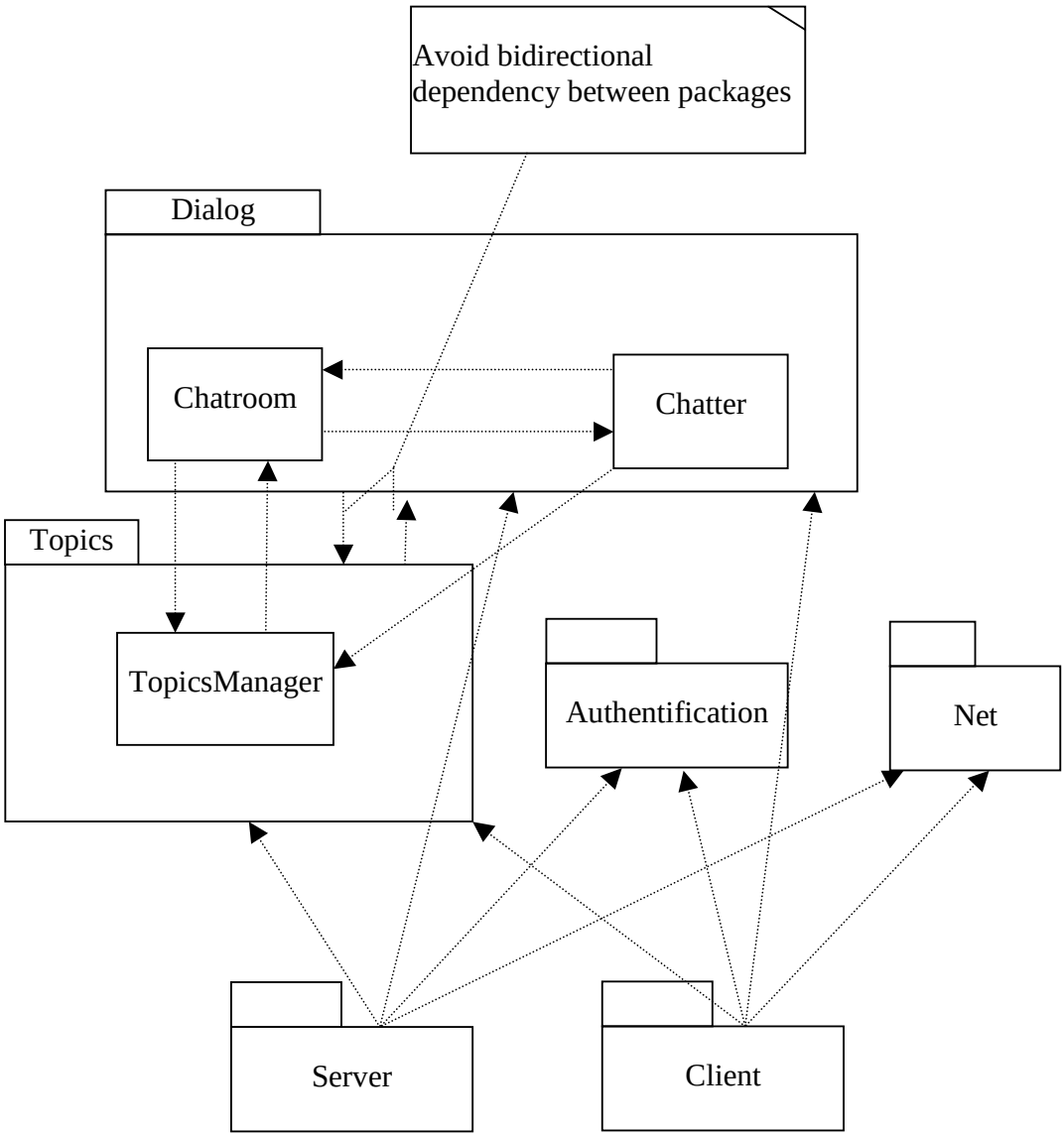



Figure 1. A preliminary partition into packages.

partition includes several good aspects :

- In the *Net* package, *Client* and *Server* are independent.
- The package *Net* is separated from the other packages : no dependency start from the package *Net*.
- *Authentification* is not linked to the packages *Client* and *Server*. Note that this partition breaks the inheritance between the classes User and Chatter (see figure 6 from the analysis). Thus the user's login and the chat session alias are independant. Furthermore, a « guest » account may have different aliases. Two attributes (login and password) act as a substitute for the inheritance.
- No dependency converge to the packages *Client* and *Server* and then *Topic* and *Dialog* are the reusable parts of the application.

Nevertheless some problems come to light :

- The topic manager manages the chat rooms, and then the *Dialog* package depends on the *Topics* package.
- Furthermore, the *Chatter* have to deal with the topics manager in order to find and to create a chatroom. And then a dependency appears from the *Dialog* package to the *Topics* package.

This partition must be changed : the two packages *Topics* and *Dialog* should be grouped into a single package named *Chat*. (see figure 2)

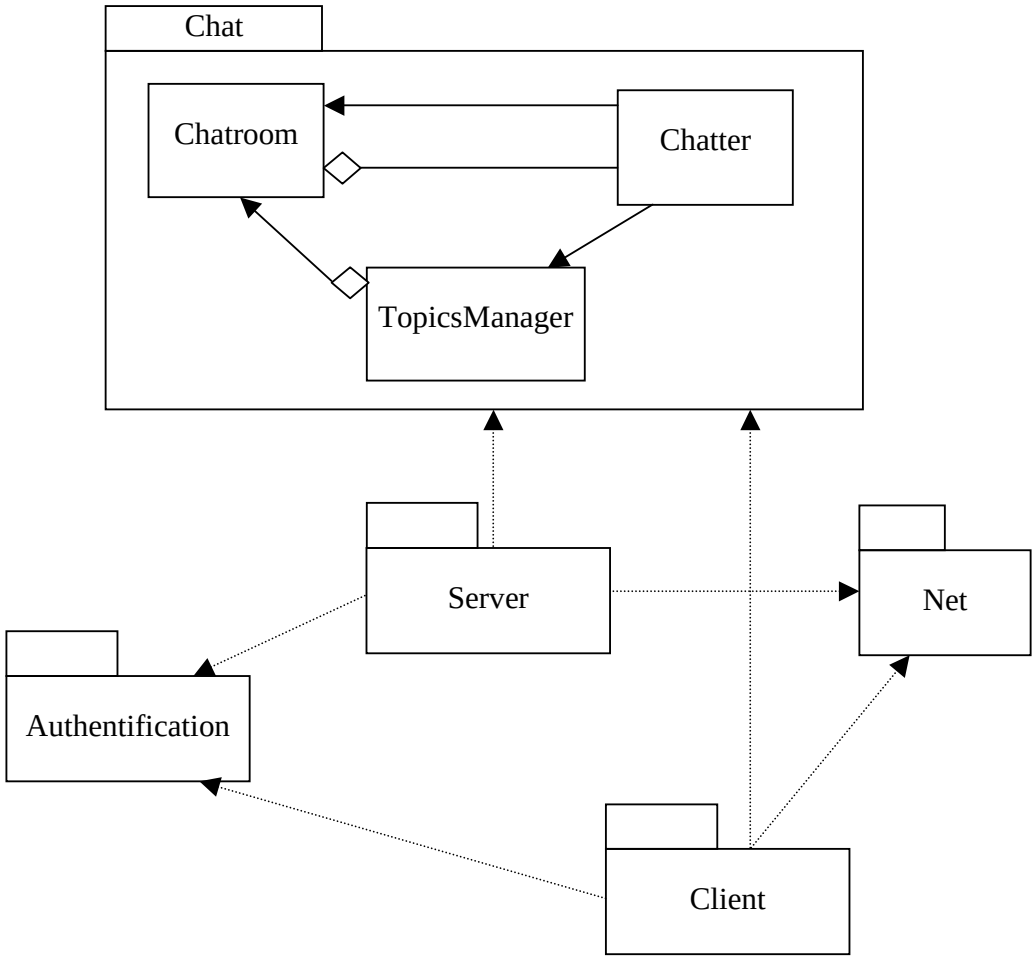



**Figure 2.** The final partition. The Chat package is now completely independent and the reusability is better.

# Etape 2 : package review

## *Session 1 : the Chat package*

Let's have a look at the dependencies between the classes grouped into the *Chat* package.

First of all let's pay attention to the packages interfaces : these interfaces can be deduced from a sequence diagram (figure 3).

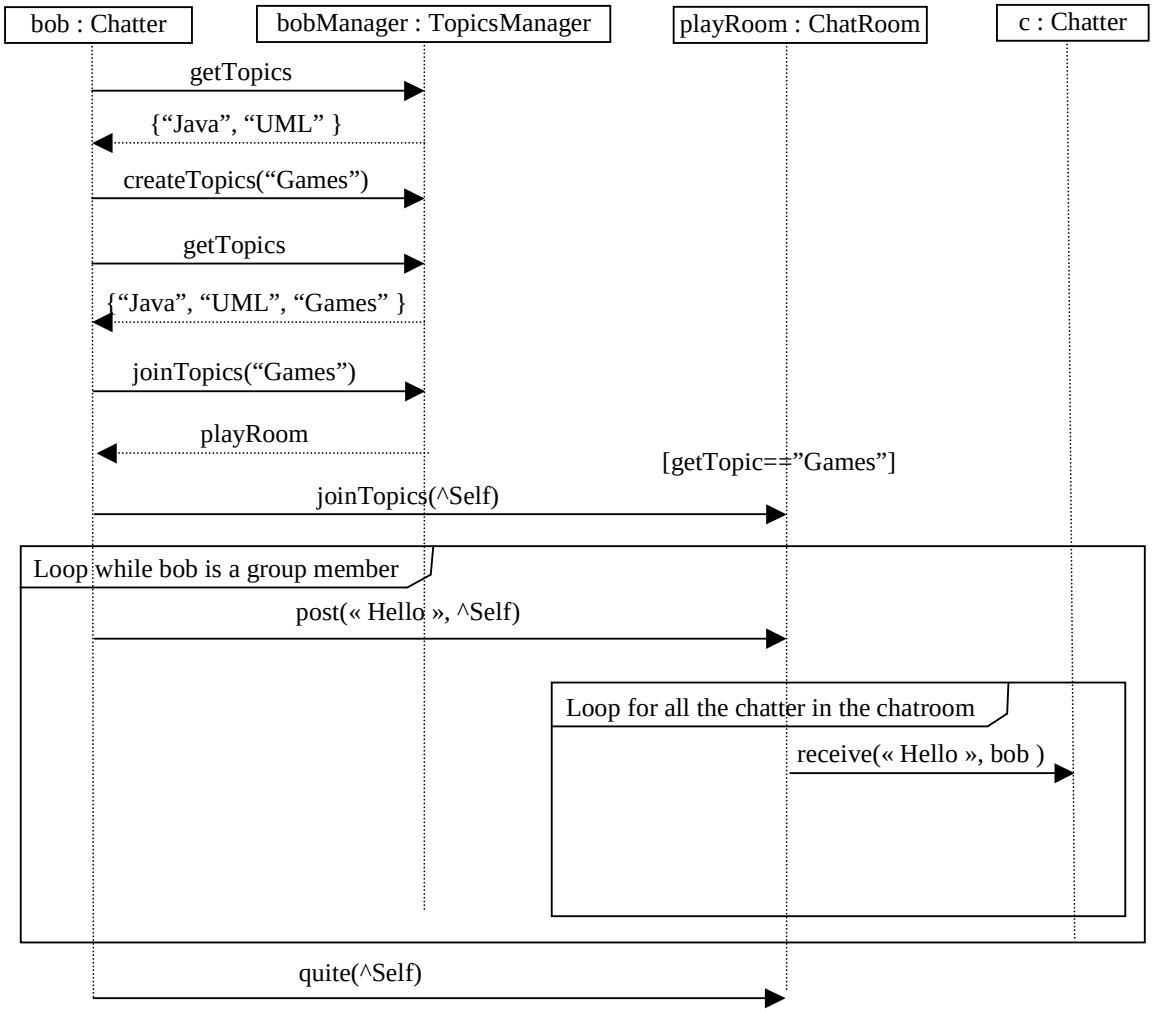



**Figure 3 :** main sequence diagram. Looking for a ChatRoom requires an association from the GestTopics class to the ChatRoom class.

Figure 3 specifies the class interfaces depicted in figure 4.

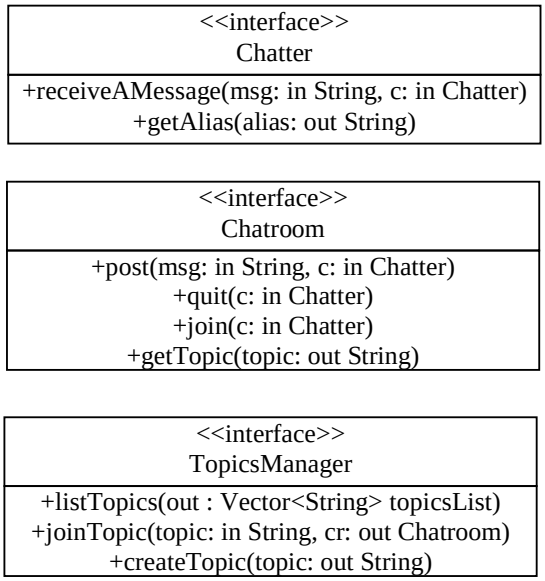
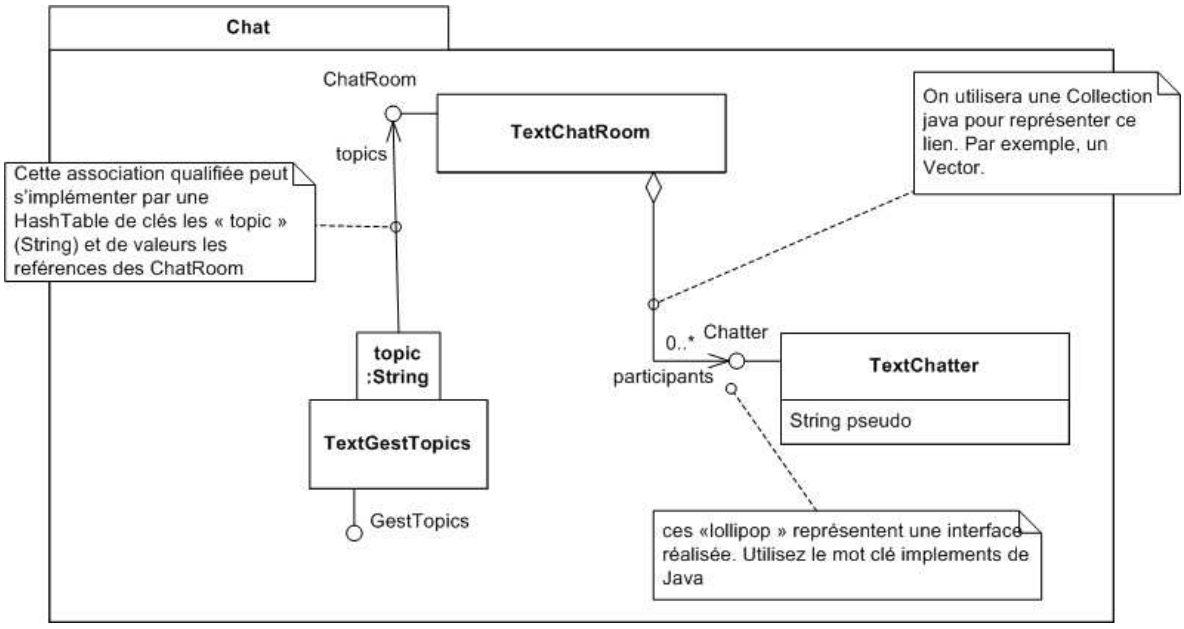



Figure 4 : detailed description of the classes from the package *Chat*.

## *To do :*

**A *Chat* package implementation : code the interfaces and a corresponding possible implementation (use the standard output the test your programs).**




Use the following main to test your implementation.

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

            gt.createTopic("java");
            gt.createTopic("UML");
            gt.listTopics();
            gt.createTopic("jeux");
            gt.listTopics();
            ChatRoom cr = gt.joinTopic("jeux");
            cr.join(bob);
            cr.post("Je suis seul ou quoi ?",bob);
            cr.join(joe);
            cr.post("Tiens, salut Joe !",bob);
            cr.post("Toi aussi tu chat sur les forums de jeux pendant les TP,
Bob?",joe);
      }
```

Your program should display the following messages on the screen :

```
The openned topics are :
UML
java
The openned topics are :
UML
java
jeux
(Message from Chatroom :jeux) Bob has join the room.
(At Bob) : Bob $> Je suis seul ou quoi ?
(Message from Chatroom :jeux) Joe has join the room.
(At Bob) : Bob $> Tiens, salut Joe !
(At Joe) : Bob $> Tiens, salut Joe !
(At Bob) : Joe $> Toi aussi tu chat sur les forums de jeux pendant les TP,
Bob?
(At Joe) : Joe $> Toi aussi tu chat sur les forums de jeux pendant les TP,
Bob?
```

## *Session 2 :* serialization and comparison interfaces, sorted collections applied to *the Authentification* package

This package is in charge of the users authentification management.

First of all, let's define the mains authentification functionalities in a Use Case diagram (figure 5). Two actors play a role during the authentification : Admin and User.

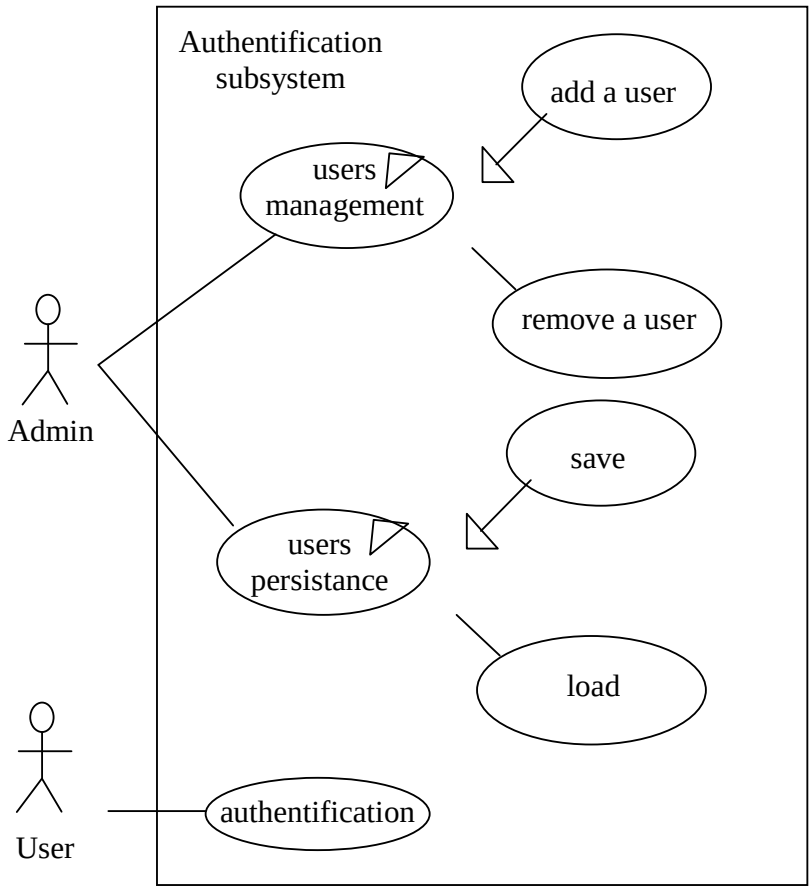



Figure 5 : the use cases for the authentification package.

The figure 5 shows an association between the User and an authetification server (through the use case authentification). Furtheremore, the informations about an authentified user are persistents (saved in a file) (see the save and load use cases).

An interface supplies an Application Program Interface for these services. To specify this interface, the Facade design pattern is used (see figure 6) : the interface is the public access to the package.

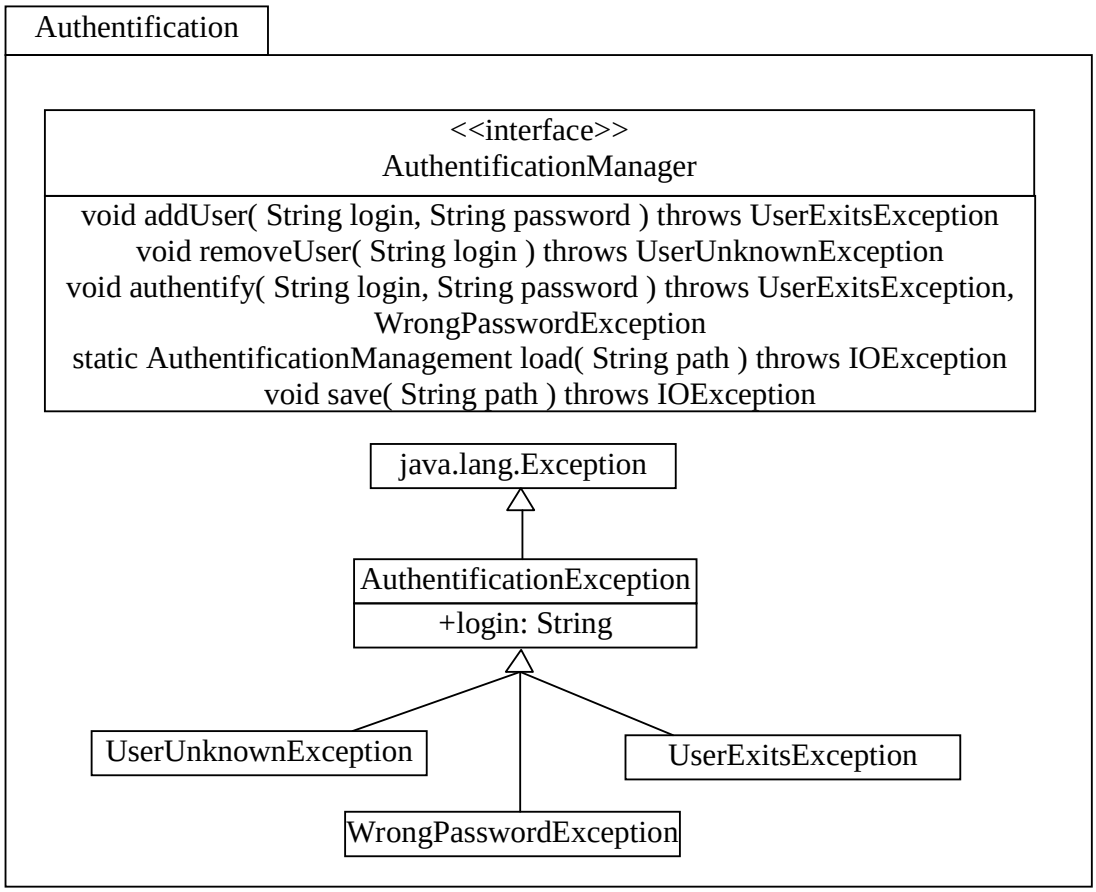



Figure 6 : public part (interface) for the authentification package.

The figure 7 shows a default implementation for this interface.

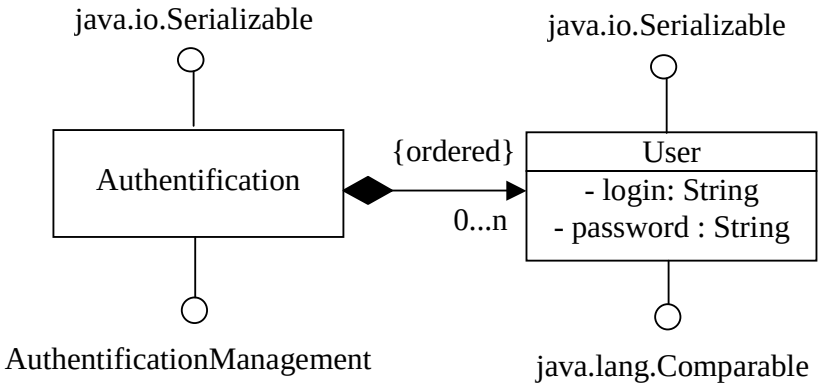



Figure 7 : a default implementation for the AuthentificationManager interface.

## *To do*

Code the authentification package (its interface and the default implementation). Test your code with the following program :

```
public static void main(String[] args) {

	AuthentificationManager am = new Authentification () ;

	// users management

	try {
		am.addUser("bob","123");
		System.out.println("Bob has been added !");
		am.removeUser("bob");
		System.out.println("Bob has been removed !");
		am.removeUser("bob");
		System.out.println("Bob has been removes twice !");
	} catch (UserUnknown e) {
		System.out.println(e.login +" : user unknown (enable to remove)
!");
	} catch (UserExists e) {
		System.out.println(e.login +" has already been added !");
	}

	// authentification

	try {
		am.addUser("bob","123");
		System.out.println("Bob has been added !");
		am.authentify("bob","123");
		System.out.println("Authentification OK !");
		am.authentify("bob","456");
		System.out.println("Invalid password !");
	} catch (WrongPassword e) {
		System.out.println(e.login+" has provided an invalid password !");
	} catch (UserExists e) {
		System.out.println(e.login +" has already been added !");

	} catch (UserUnknown e) {
		System.out.println(e.login +" : user unknown (enable to remove)
!");
	}

	// persistance

	try {
		am.save("users.txt");
		AuthentificationManager am1 = new Authentification();
		am1.load("users.txt");
		am1.authentify("bob","123");
		System.out.println("Loading complete !");
```

```
      } catch (UserUnknown e) {
            System.out.println(e.login +" is unknown ! error during the
saving/loading.");
      } catch (WrongPassword e) {
            System.out.println(e.login+" has provided an invalid password
!error during the saving/loading .");
      } catch (IOException e) {
            System.out.println(e);
      }

}
```

Note : the package java.net provides a complete authentification mecanism (see the class java.net.Authenticator). But the above implementation is quite enough for our system.

Optional work: give another implementation to serialize the authentification's datas in a relational data base. The add/remove operations can be done by using the « insert into » and « delete from » requests ; a commit operation allows to save the authentification datas. To implemants the data base access, you choose among two tehnologies : JDBC or Hibernate. JDBC is a low level API which requires to insert SQL requests within Java code. Nowadays, Hibernate is the most popular tool for persistance but, it is more complicated than JDBC.

## Session 3 : the package Net

This package is in charge of the network management.

The underlying protocol is connected : TCP/IP. The proposed implementation is based on a message exchange. The messages are sent simultaneously by many clients (each client instance on the server is computed in one thread).

A message is composed by a Header and a body (containing the datas). Note that, the read operation will block until an incoming message happens. Furthermore, try to write through a full piped stream will block your program.

## The connected protocol TCP.

TCP avoids transmission errors and guaranty the packets ordering : an acknowledge message is automatically received for each transmitted packet (note that, the performance should be enhanced by using the UDP protocol, because no stream controls are performed).

Using TCP, the connection is asymmetric :

- The server listens (waits for connection) on a particular socket (a socket is defined by a port number - for instance 80 is the port for a http server - and an IP address) ; when a client tries to connect to the server, the server *accepts* a new connection, and then a new socket (a dialog socket) is created.
- The client creates a socket and then it sends a connection message to the server.

After the connection step, client and server can send or receive messages in a full duplex mode.

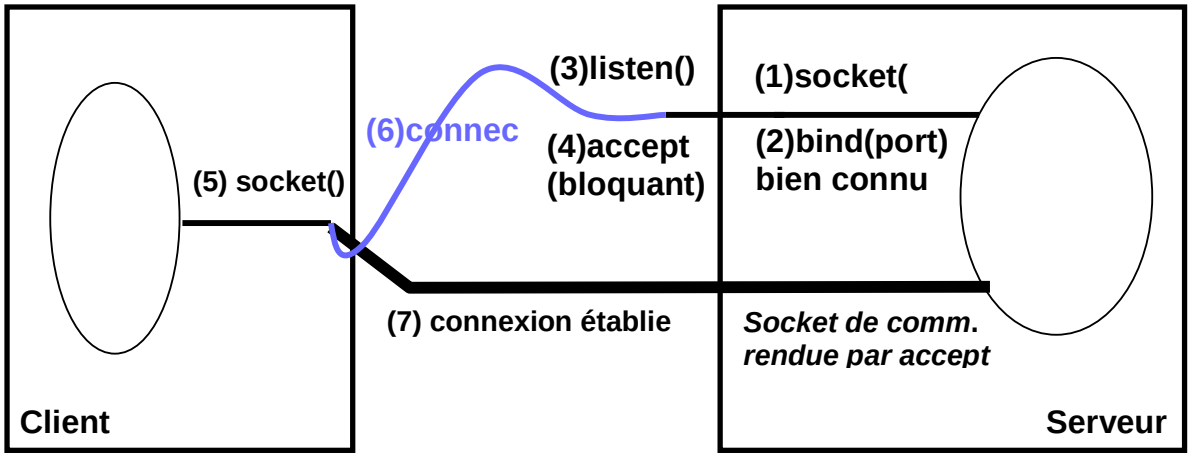



Figure 8 : a TCP connection. Step 1 to 3 are computed by the constructor of the ServerSocket class.

The figure 9 below shows many threads : a thread waits for clients (the main loop) while the other threads allows the communication between the server and each client.

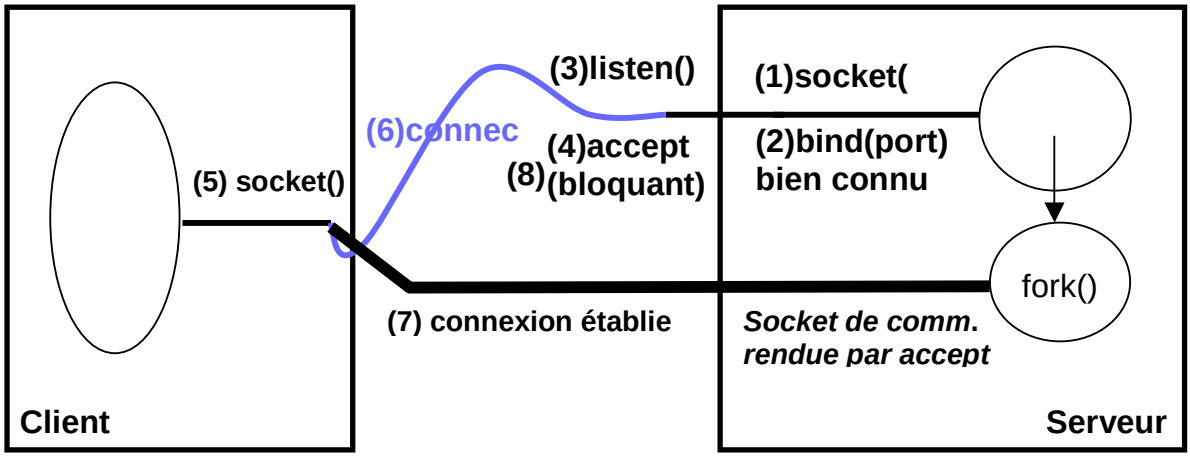



Figure 9 : multiple clients communicate with the server in separated threads.

## *Proposed implementation*

An abstract class ensure the net package reusability. The figure below shows our application skeleton .

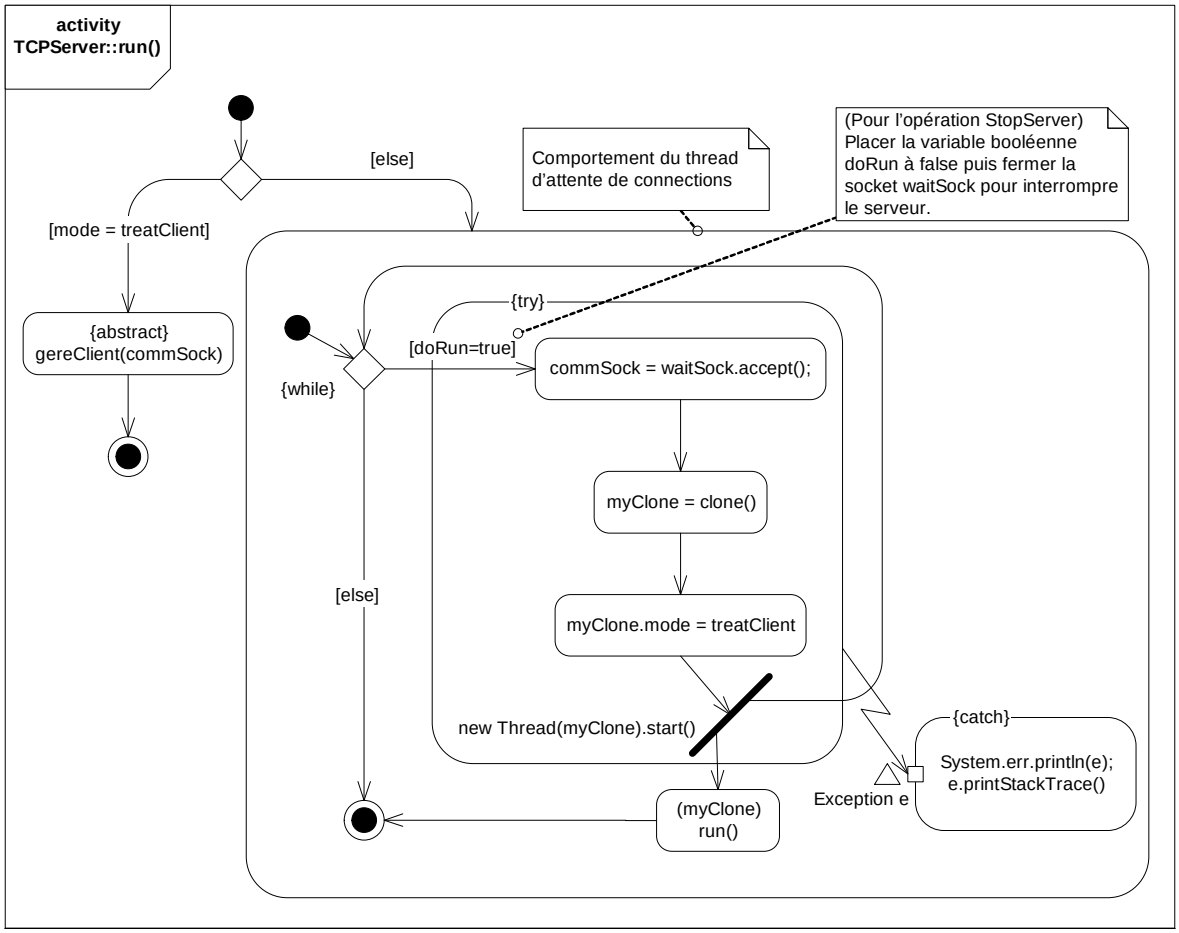



Figure 10 : an activity diagram for the server run's method.

Note that the diagram above use the clone method : thus, each cloned object holds the references towards the others objects (a reference towards the authentification manager for instance).

The following diagram (figure 11) is the net package summery.

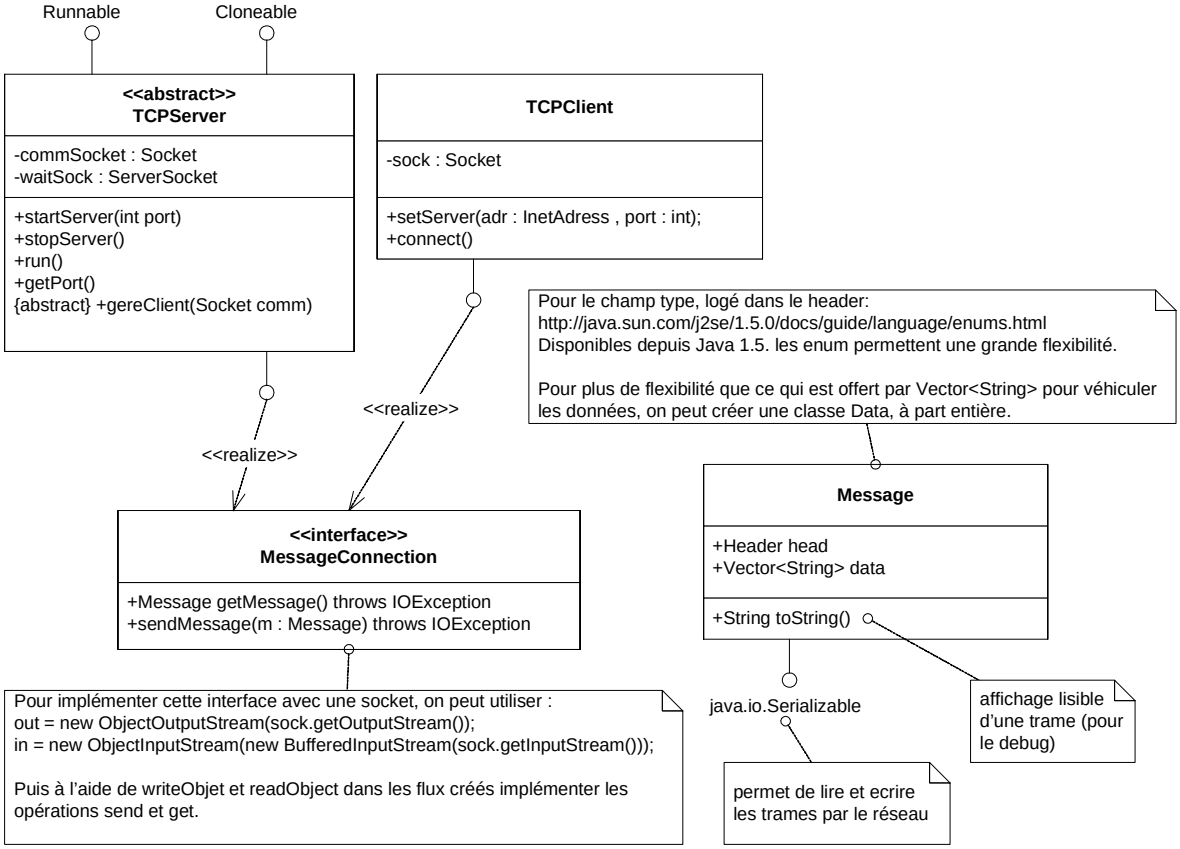



Figure 11 : the net package.

## *To do*

Implement the net package.

Optional work: use RMI rather than a low level implementation using sockets. RMI, Remote Method Invocation, allows to install objects into a server, and to perform remote object's method calls.

## Session 4 : composant assembly.

It is time to clarify the application architecture : which part is computed by the server, and which part resides in the client. The server side contains the topics manager and the chat room, while on the client side is the chatter.

For the implementation, the Proxy design pattern is used : a local component plays the role of a remote component.

Furthermore, we need to define the application protocol (note that the underlying protocol TCP/IP is in charge of the acknowledgements.

## The server package.

A distinct port to communicate is used for each chat room, while a well known port is used by the topics process.

The topics manager needs 3 frames : to list the topics, to create a new topic and to join an existing topic. The gereClient method in the next figure returns one of theses 3 responses.

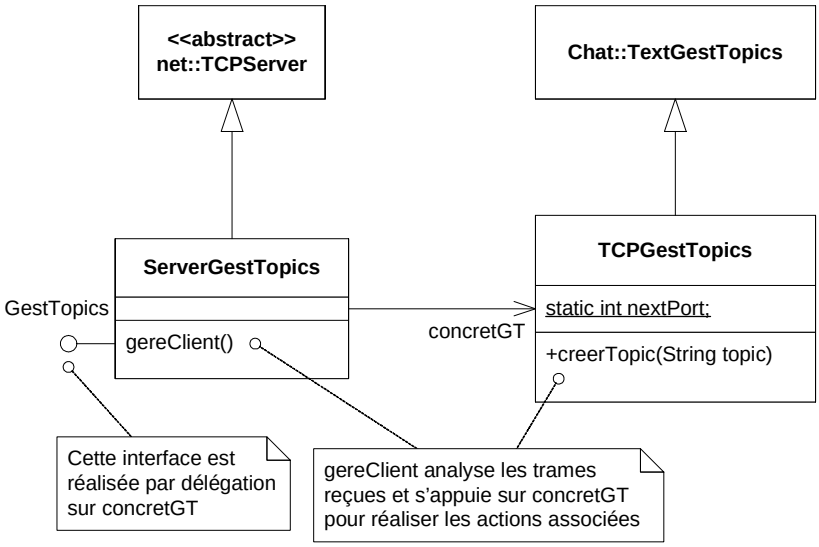



Figure 12 : Topics manager instanciation for the network nanagement.

The same model is used for the chat room management (figure 13) but the chat room server implements the Chatter interface.

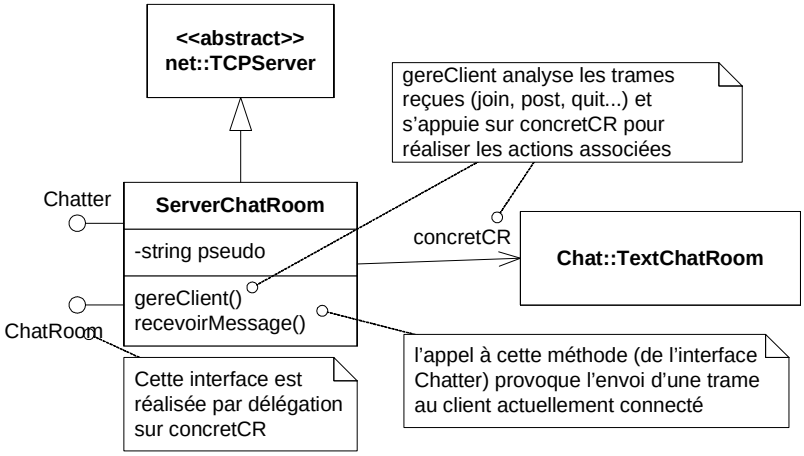



Figure 13 : server side ChatRoom instanciation

## *The Client package*

On the client side, network clients implement the topics and chatrooms managers.

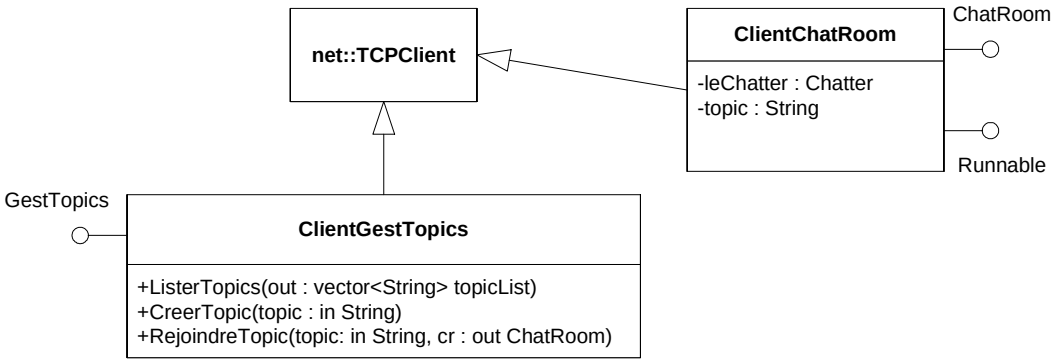



Figure 14 : classes from the Client package.

## *Interaction diagrams using proxies*

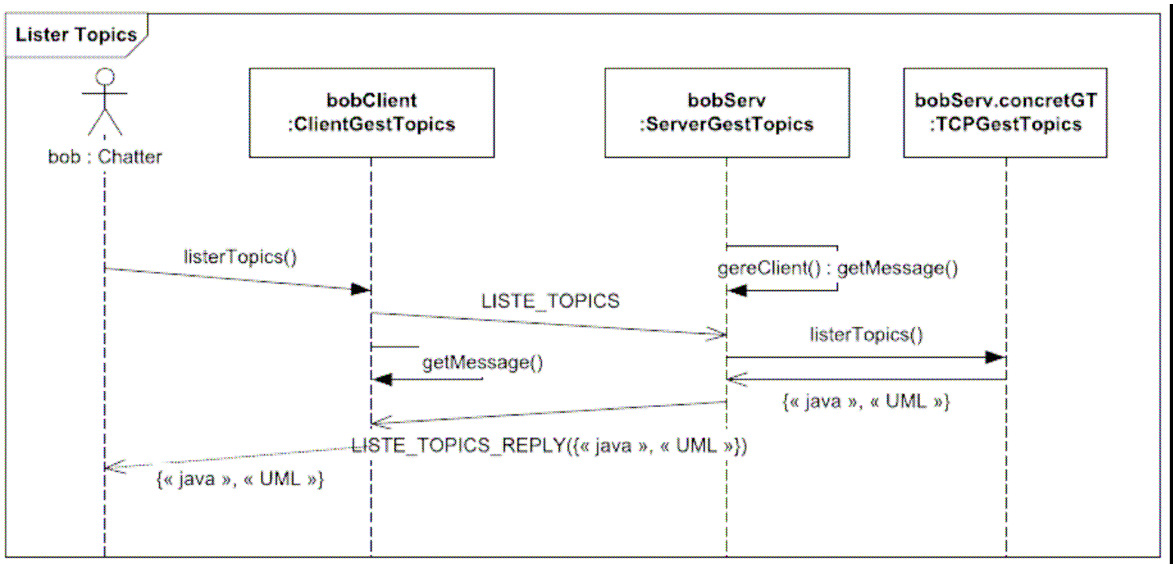



Figure 15 : using of a proxy to list the opened topics.

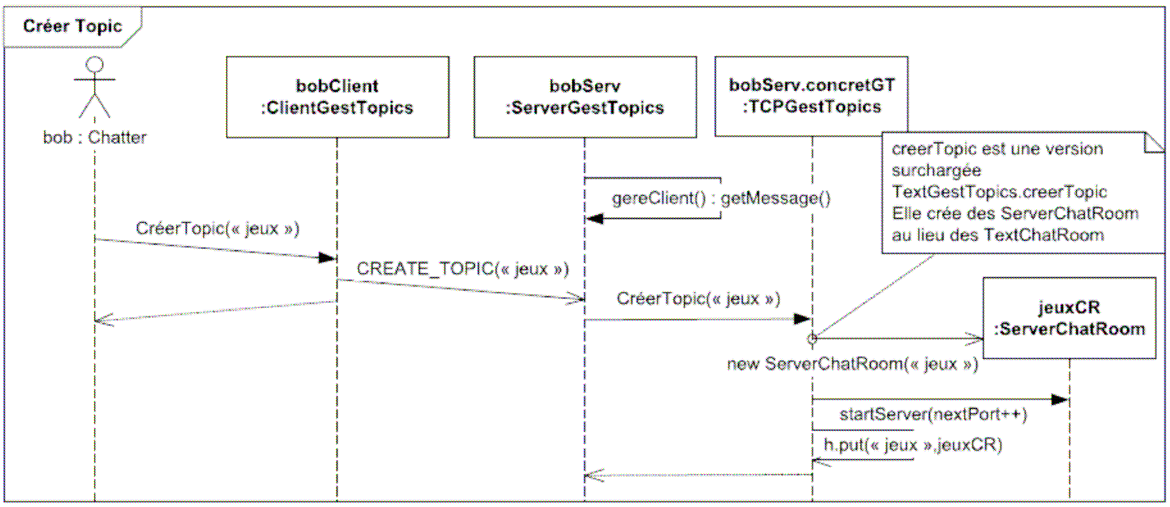



Figure 16 : chatroom creation through the topic manager proxy

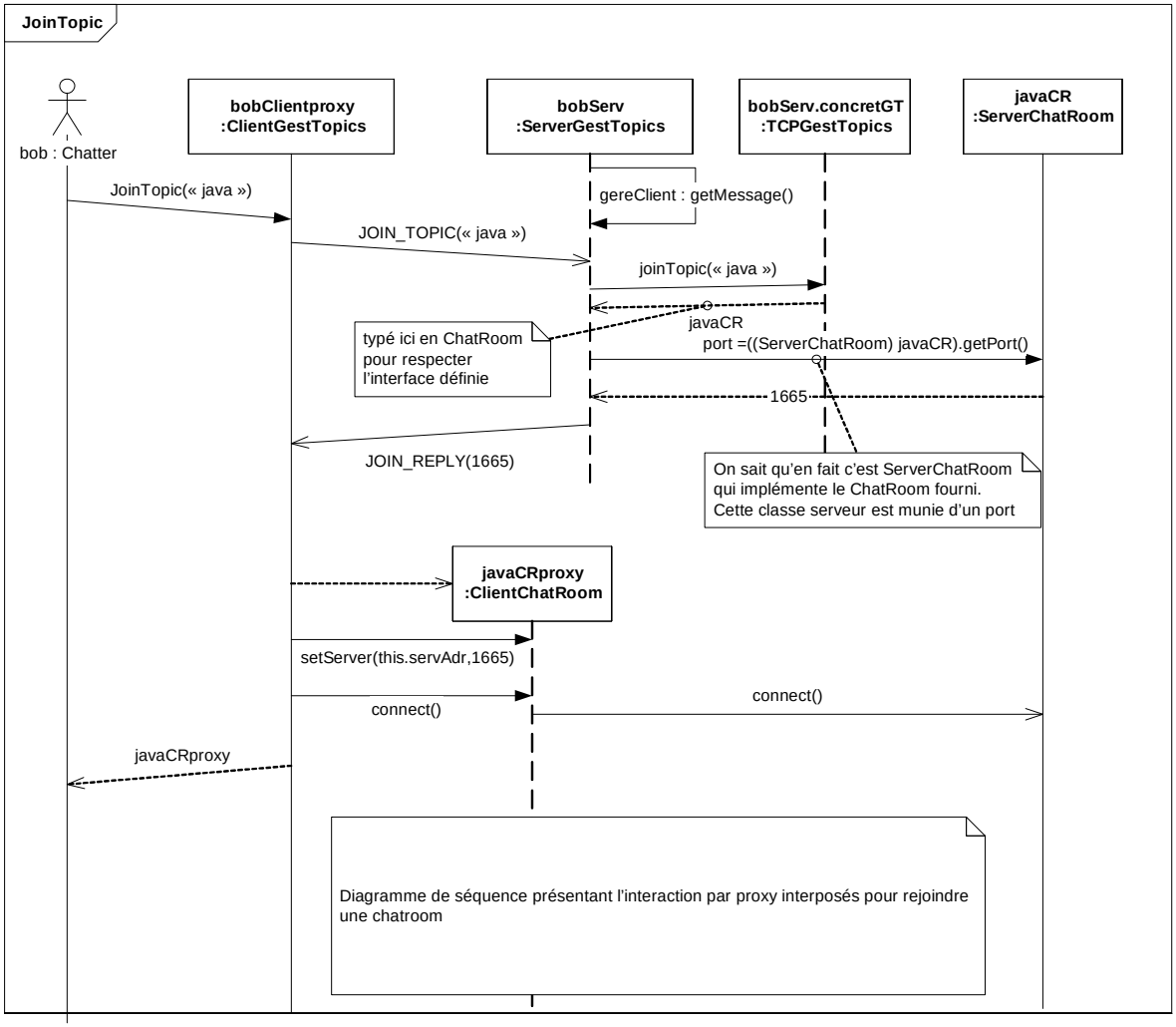



Figure 17 : client side chatroom creation through the topic manager proxy

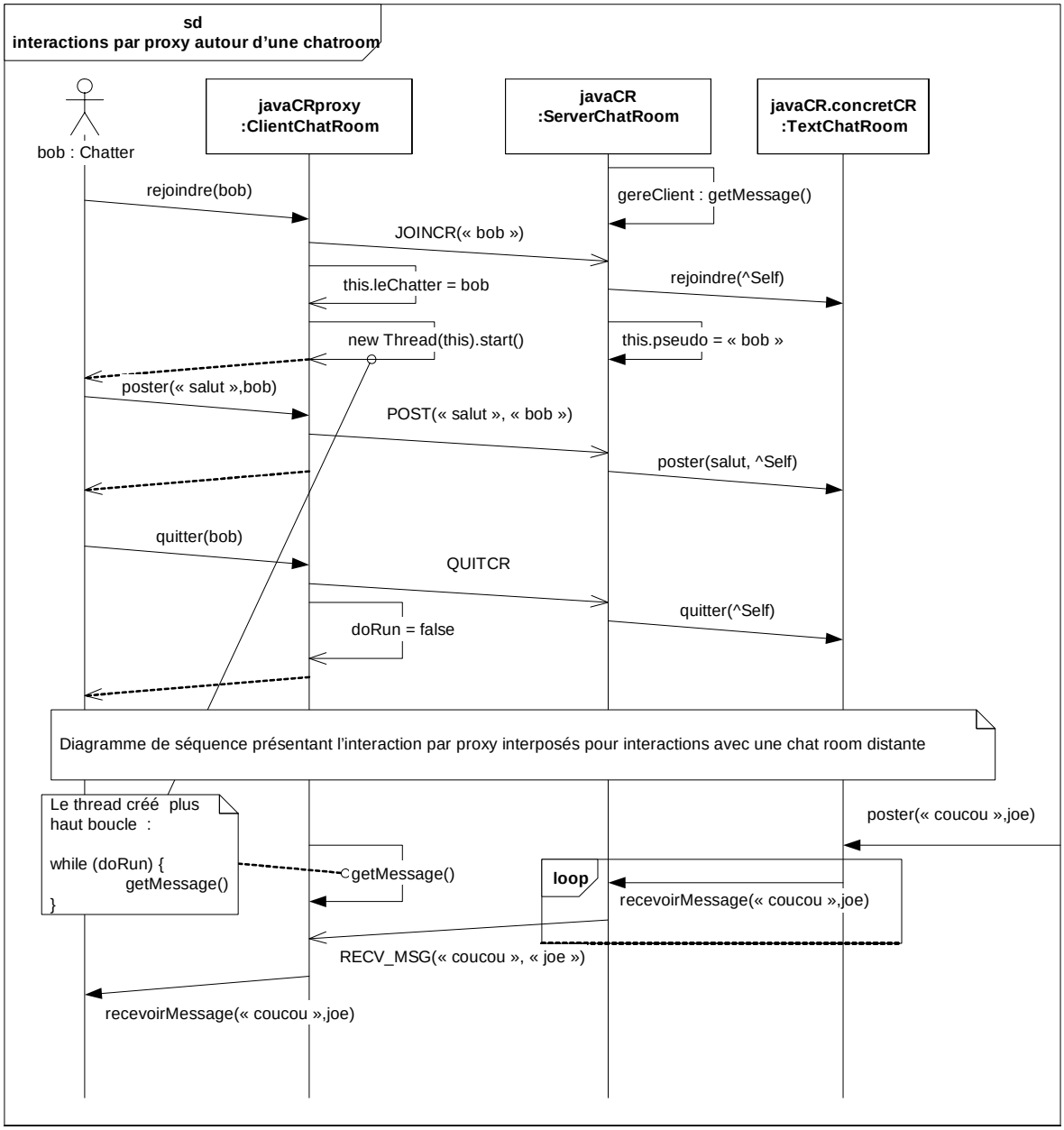


Figure 18 : network interactions with the chat room.

## *To do*

Define and implement the network protocol for managing the communcations.

# Session 5 : Graphical User Interface

Code the application View Layer with the Swings library.