In our IM design, our “Conversation” will be similar to a chat room. A Conversation has a unique title and is started by a User; any other User can join the Conversation by adding themselves to the list of the Conversation’s members (issuing a *join* command). A Conversation is destroyed when all of its members have left, either by manually choosing to leave the Conversation or exiting the chat client.

The user communicates with the server through an open socket for each user.

public class Room (“Conversation”)

This class represents a “Conversation” in guichat, and stores a List of members (Users) aHnd a List of Messages. Has functions to to process User commands such as “join-room” and “leave-room”. An instance of Room is created when at least one person joins the room, and then removed when it has no more members.

```
private String title;
private List<User> members;
private List<Message> messages;

//Construct a Room with the t
public Room(String title, User owner);

//Returns true if this Room has User u as one of its current members;
//false otherwise
public boolean hasUser(User u);

//Add User u to this Room’s list of members
public void addUser(User u);

//Remove User u from this Room’s list of members
public void removeUser(User u);

//Add Message m to this Room’s List of Messages
public void addMessage(Message m);
```

```
//Returns the number of users in this Room.
public int getUserCount();
```

public class Server

The chat server. Stores a HashMap linking room names to Rooms and a HashMap of usernames and their respective User classes. Also has functions to process User commands such as “change nickname to john”, “join some_room”, and “whois other_user”

```
private ServerSocket server;
private HashMap<String, User> userMap;
private HashMap<String, Room> roomMap;

// Instantiate a server on the specified port.  Create Server fields of
// a ServerSocket, a HashMap<String, User> and a HashMap<String, Room>
public Server(int port);

// Listen for connections on the specified port; creates User and
// Threads to run
public void listen();

// Try to change a User with oldUsername to have newUsername
public String changeUsername(String oldUsername, String newUsername);

// Process a user “joining” a room, true if join successful
public boolean addUserToRoom(String userNick, String roomName);

// Process a user “leaving” a room, true if leave successful
public boolean removeUserFromRoom(String userNick, String roomName);
```

```
//Remove a User that has disconnected from the Server
public void processUserDisconnect(String username);
```

public class User *implements Runnable*

A new instance of this class gets created in a separate Thread whenever a user connects to the server. The User class processes all Client-Server communication on the Socket called socket, and delegates the execution of the commands passed by the client to the Server.

```
public String nickname;
public Socket sock;
private Server serv;
private List<String> rooms;

// Create a new User given a server, socket, and username
public User(Server server, Socket socket, String nickname);

// Get a list of the names of rooms that this user is in.
public List<String> getRoomNames();

// Process client commands as they arrive on Socket sock until the
// "quit" command is received, or until the socket times out.
public void run();
```

public class Message

This class represents the messages users send to each other in a chat.

```
private User sender;
private Room room;
private Date date;
private String content;
```

```
// Create a new message given an author, room, date, and message content
public Message(User sender, Room room, Date date, String content);

// Returns the contents of this Message
public String getContent();

// Returns the date that this message was sent
public Date getDate();

// Returns the sender of this message
public User getSender();

// Returns the Room of this message
public Room getRoom();

// Returns a chat version of the message: "{date} {sender}: {content}"
public String toString();
```

Client-Server Protocol

command ::= change-nick | list-channels | list-users | whois-user |
my-profile | join-room | leave-room

change-nick ::= "/nick " NICKNAME	// Switch your nickname to NICKNAME
list-channels ::= "/rooms"	// display a list of all current Room
list-users ::= "/who"	// display a list of all logged in users in current Room
whois-user ::= "/whois " NICKNAME	// display a list of channels that NICKNAME is in
my-profile ::= "/me"	// display a list of channels that you're in
join-room := "join " ROOMNAME	// try to join Room with a name of ROOMNAME;
	// If it doesn't exist, create it and then join the room.

leave-room := "leave" ROOMNAME? // exit current room, or room specified by ROOMNAME
quit := "quit" // terminate connection with the server

IP ::= [0-9\.]+ //IP address
DIGITS ::= [0-9]+
NICKNAME ::= WORD
ROOMNAME ::= WORD

WORD ::= [A-Za-z0-9_]+

// Client side only
open-convo := "open " ROOMNAME // open new GUI window for the Room with name ROOMNAME
connect ::= "connect " IP // Connect to the server at IP with a random nickname.

Testing strategy

The main testing strategy is to start a dummy server listening on a port, and then simulate a client connecting to this dummy server and processing different actions (creating a room, quitting a room, closing the client... etc) and verifying that both the response of the server is acceptable and that the rooms and messages specified in the test cases are actually created and are readable on the client.

Also we will have several test cases with multiple clients where one of the clients will try to read and respond to a conversation sent by the first client - we can do this by verifying the input & output sockets for different users and checking that the message order is preserved.

In general, we want to test all commands in our Client-Server protocol, which includes change-nick, list-channels, list-users, whois-user, my-profile, join-room, leave-room *(among others - this is just the minimum list of commands)*

Snapshot Diagram