

I. Feature Implemented

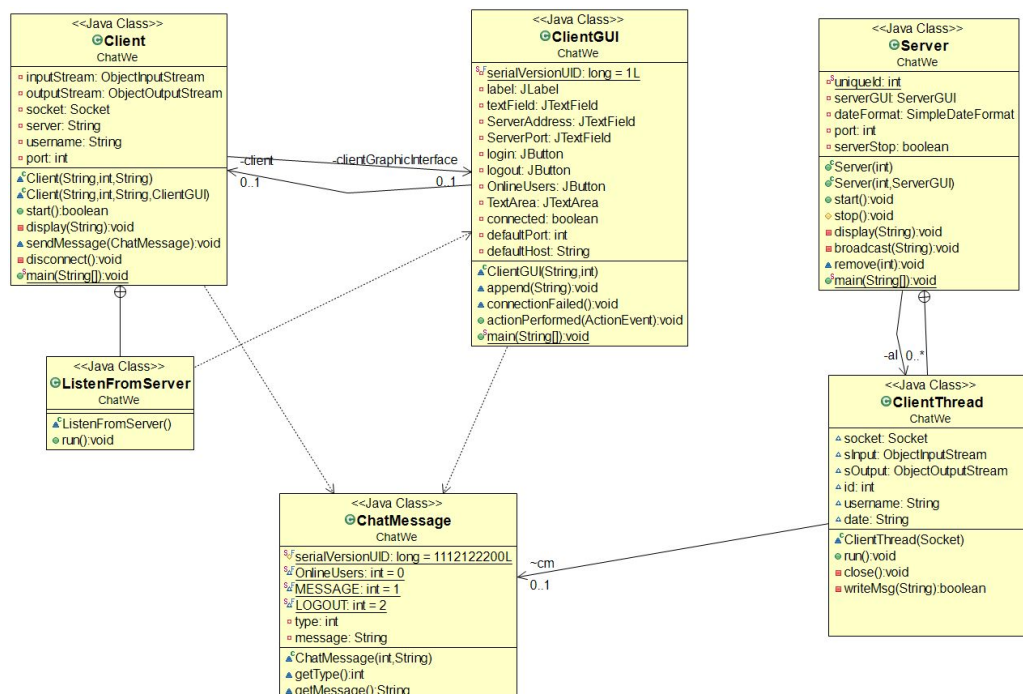
Id	Title
UC-01	Login
UC-02	Logout
UC-03	Search Contact List
UC-04	Add contact
UC-05	Chatting, both individual and group chatting.

II. Feature Not Implemented

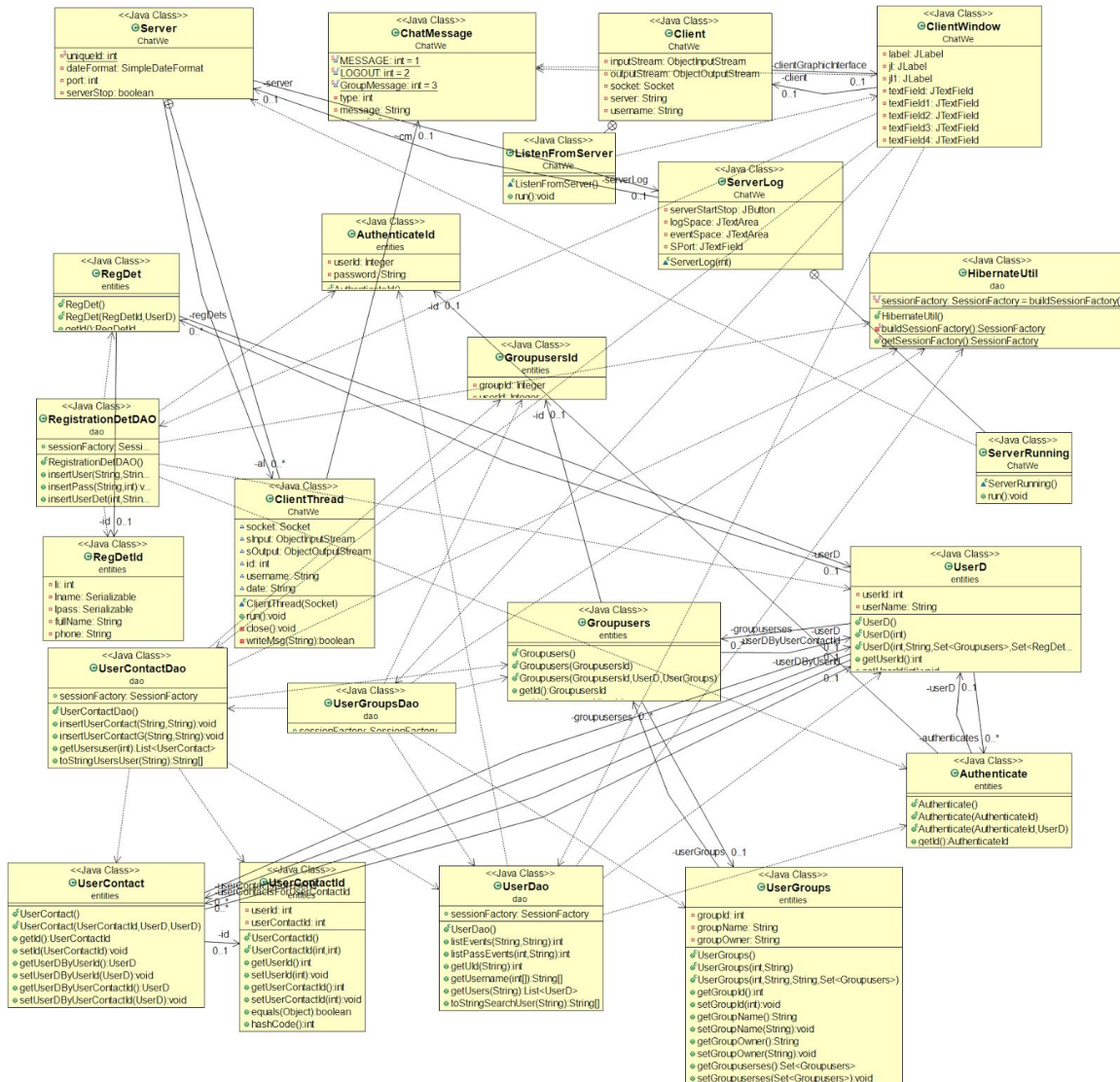
Id	Title
UC-08	remove user

III. Class Diagram Comparison

The old Class Diagram:



In the final project,



IV. Design patterns used

Observer design pattern

Let us summarize the design pattern:

1. There is an observer class(abstract) or have an interface
2. Define the class that would extend the observer class or implement the interface in case you have an interface
3. Make a constructor/method attach/store the list of observers in a list
4. Define a notify method that calls the implemented update method on every observer

Please see the below implementation of the above:

Source: (https://sourcemaking.com/design_patterns/observer/java/1)

```
abstract class Observer {
protected Subject subj;
    public abstract void update();
}

class HexObserver extends Observer {
    public HexObserver( Subject s ) {
        subj = s;
        subj.attach( this );
    }

    public void update() {
        System.out.print( " " + Integer.toHexString( subj.getState() ) );
    }
} // Observers "pull" information

class OctObserver extends Observer {
    public OctObserver( Subject s ) {
        subj = s;
        subj.attach( this );
    }

    public void update() {
        System.out.print( " " + Integer.toOctalString( subj.getState() ) );
    }
} // Observers "pull" information

class Subject {
    private Observer[] observers = new Observer[9];
    private int totalObs = 0;
    private int state;
    public void attach( Observer o ) {
        observers[totalObs++] = o;
    }

    public int getState() {
        return state;
    }

    public void setState( int in ) {
        state = in;
        notify();
    }

    private void notify() {
        for (int i=0; i < totalObs; i++) {
            observers[i].update();
        }
    }
}
```

In our code base we have done the above fur steps:

1. The abstract class: `ClientThread`

The list of observer/`ClientThread` class:

```
// this is my observer class, it conta
private ArrayList<ClientThread> al;
// for the graphics
```

2. The start() method adds all the observer to the server instance

```
public void start() {
    serverStop = true;
    /* create socket server and wait for connection requests */
    try
    {
        // the socket used by the server
        ServerSocket serverSocket = new ServerSocket(port);

        // infinite loop to wait for connections
        while(serverStop)
        {
            // format message saying we are waiting
            display("Server waiting for Clients on port " + port + ".");

            Socket socket = serverSocket.accept();    // accept connection
            // if I was asked to stop
            if(!serverStop)
                break;
            ClientThread t = new ClientThread(socket); // make a thread of it
            //add the observer t into the list of observers
            al.add(t);                                // save it in the ArrayList
            t.start();
        }
        // I was asked to stop
    }
}
```

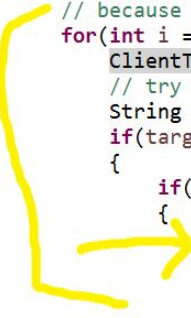
3. The broadcast() method is what acts like the notifier. It checks all the valid observers and send the message to them. Originally in the example code above the notify() method calls all the observer.update() method. In our case the broadcast() method calls all the clientthread.writemessage() method.

```

//In case of peer to peer chat we have single valid observer
private synchronized void broadcast(String message, String target, String targets[], String so) {
    // add HH:mm:ss and \n to the message
    String time = dateFormat.format(new Date());
    String messageLf = time + " " + message + "\n";
    // display message on console or GUI
    if(serverLog == null)
        System.out.print(messageLf);
    else
        serverLog.appendRoom(messageLf);    // append in the room window

    // we loop in reverse order in case we would have to remove a Client
    // because it has disconnected
    for(int i = al.size(); --i >= 0;) {
        ClientThread ct = al.get(i);
        // try to write to the Client if it fails remove it from the list
        String h = ct.username;
        if(target != null)
        {
            if(target.equals(h) || so.equals(h))
            {
                if(!ct.writeMsg(messageLf)) {
                    al.remove(i);
                    display("Disconnected Client " + h + " removed from list.");
                }
            }
        }
    }
}

```



V. Lesson Learn

What we learned from analysis and design is, during analysis, don't go too deep with design, especially low level design like class diagrams. Because things will be changed, the final design and class diagram will be dramatically different from the initial design (class diagram). It's very good to get a pro-type work earlier, since it will help to validate the requirements and figure out potential technical difficulties at the early stage.