1. Feature Implemented

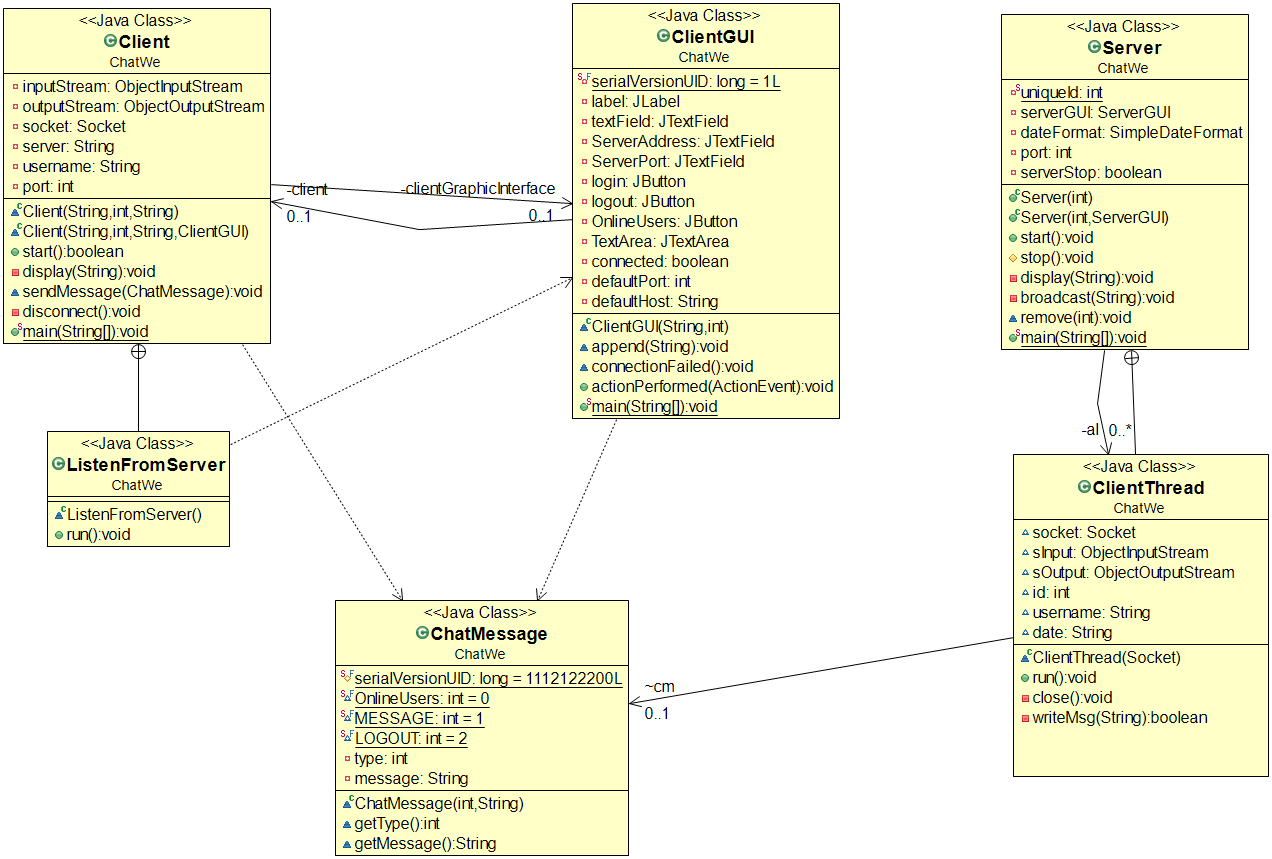
|  |  |
| --- | --- |
| **Id** | **Title** |
| **UC-01** | Login |
| **UC-02** | Logout |
| **UC-03** | Search Contact List |
| **UC-04** | Add contact |
| **UC-05** | Chatting, both individual and group chatting. |

1. Feature Not Implemented

|  |  |
| --- | --- |
| **Id** | **Title** |
| **UC-08** | remove user |

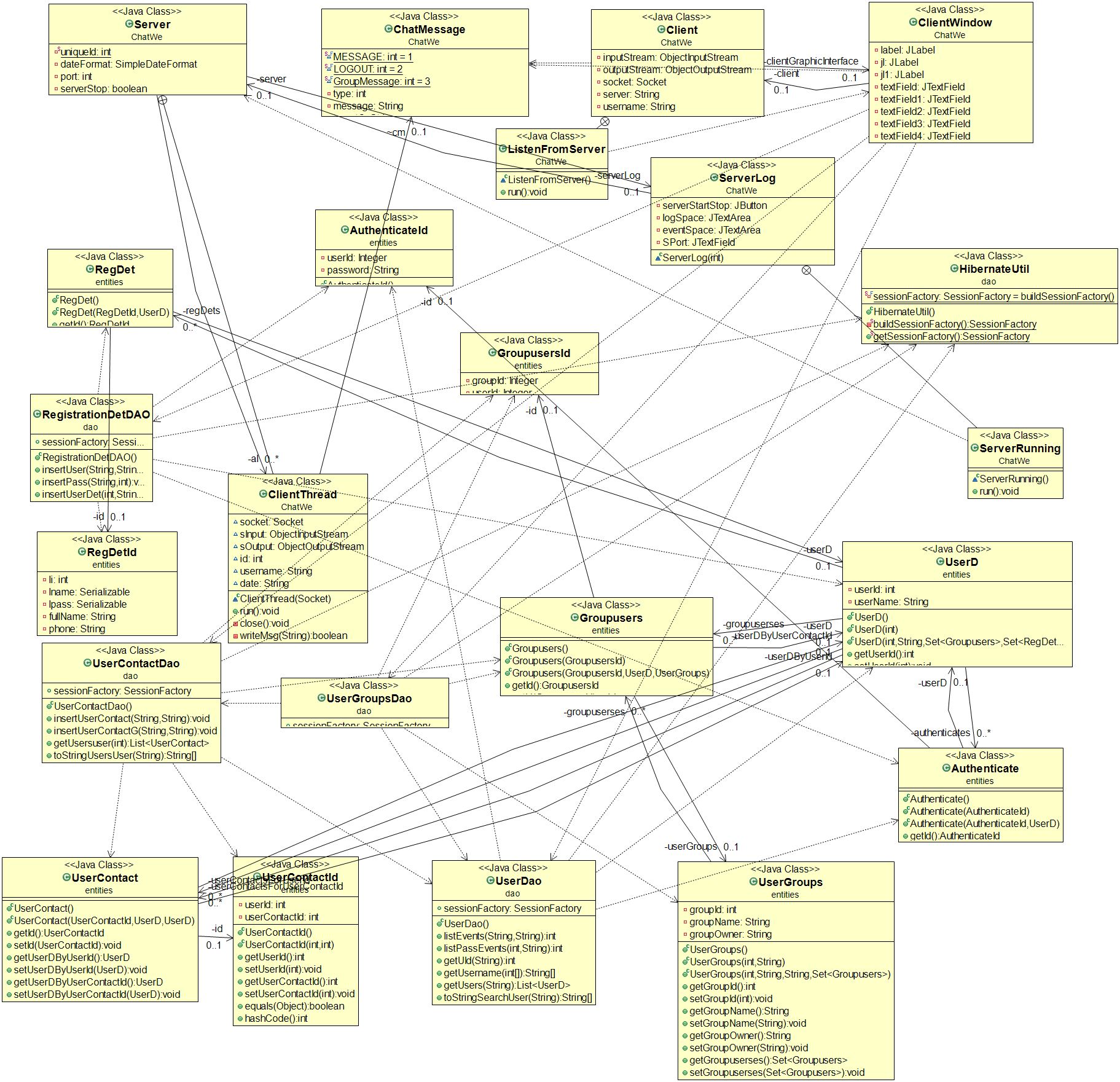
1. Class Diagram Comparison

The old Class Diagram:



In the final project,

The above classes still remain with some changes.



1. 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();

}

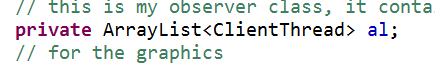
}

}

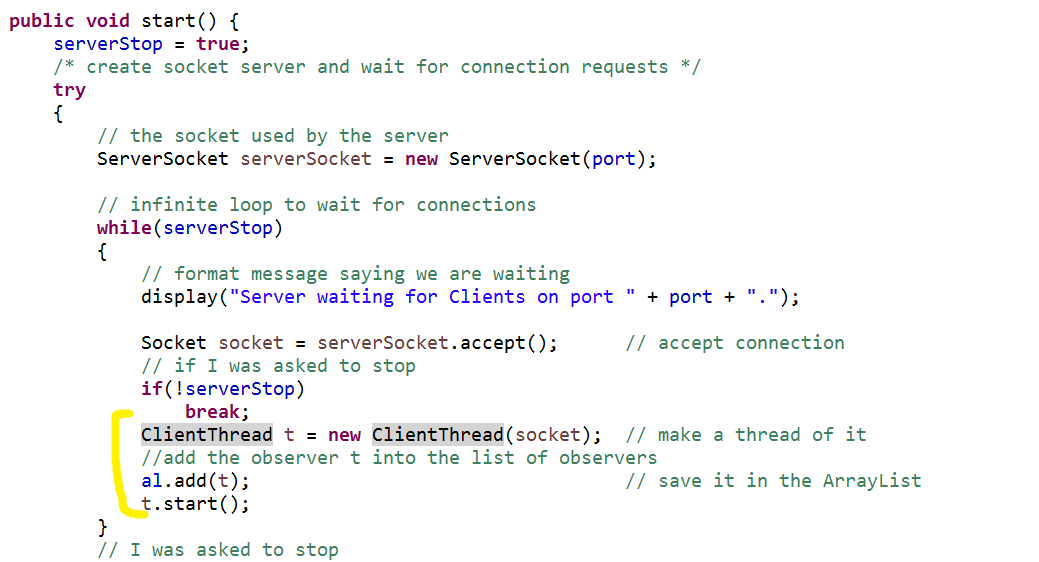
In our code base we have done the above fur steps:

1. The abstract class: ClientThread

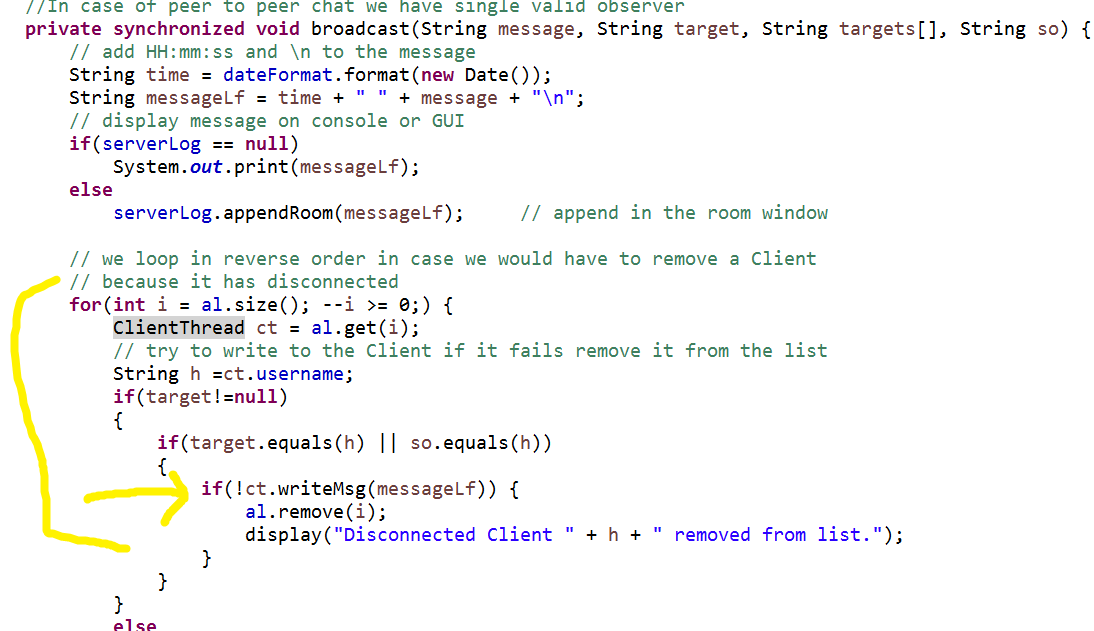
The list of observer/ClientThread class:



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



1. 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.



1. 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.