

# Übung 04

Arbeitsaufwand insgesamt: 8h

## Inhaltsverzeichnis

Teil 1 – Zugriff auf eine Adress-Datenbank mit JDBC .....	3
Lösungsidee: .....	3
UML .....	3
Source-Code .....	4
PersonDaoTest.java .....	4
Person.java .....	6
Dao.java .....	7
PersonDao.java .....	7
AbstractDao.java .....	8
PersonDaoJdbc.java .....	9
Testfälle .....	12
Teil 2 – Implementierung eines Gruppen-Chat-Clients .....	13
Lösungsidee .....	13
UML .....	14
Sequenzdiagramm .....	16
Source-Code .....	17
ChatServer.java .....	17
ChatClient.java .....	19
Client.java .....	22
SharedServerState.java .....	23
Message.java .....	25
MessageBuffer.java .....	25
MessageBufferImpl.java .....	26
MessageSender.java .....	27
MessageSenderImpl.java .....	27
MessageReceiver.java .....	28
MessageReceiverImpl.java .....	28
MessageProcessor.java .....	29
MessageProcessorImpl.java .....	29

MessageHandler.java .....	30
AbstractMessageHandler.java.....	30
ConnectMessageHandler.java.....	31
SendMessageHandler.java .....	32
AliveMessageHandler.java .....	33
QuitMessageHandler.java .....	35
Testfälle .....	36
Parade-Beispiel.....	36
Maximale Client-Anzahl am Server .....	37
Fehlende Antwort auf Alive-Message .....	37
Leere Nachricht .....	38
Zu lange Nachrichten.....	38

## Teil 1 – Zugriff auf eine Adress-Datenbank mit JDBC

In diesem Teil der Übung soll eine Anwendung zum Verwalten von Personendaten in einer Datenbank implementiert werden.

### Lösungsidee:

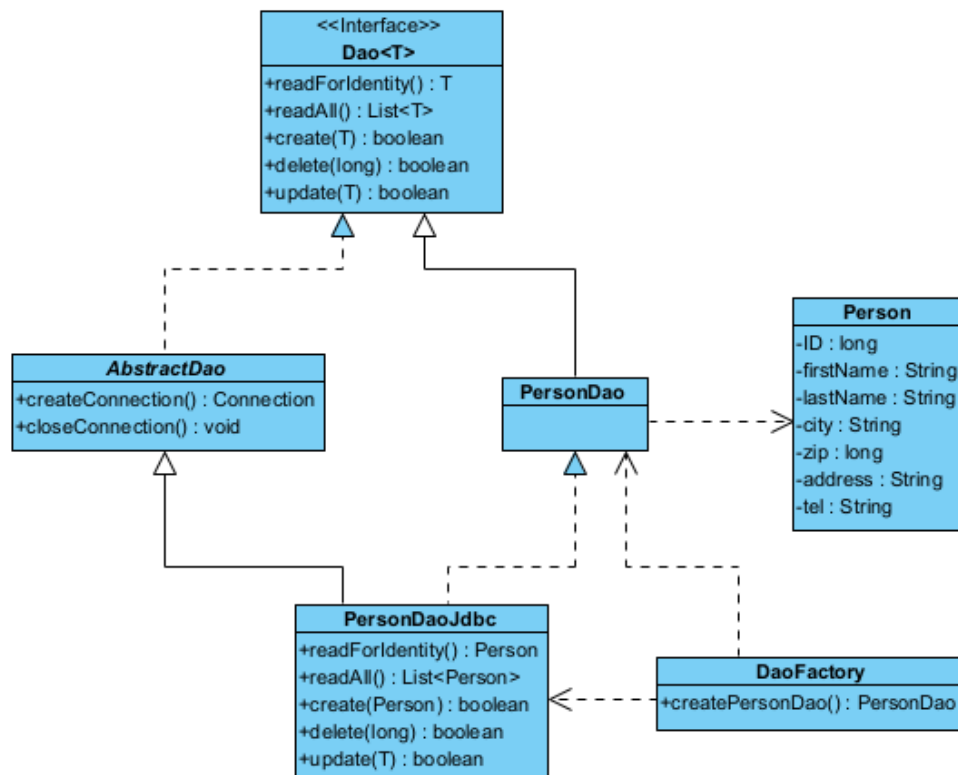
Im Mittelpunkt der Datenbank stehen „Personen“, weshalb ich mit der Kapselung der Informationen Vorname, Nachname, Wohnort, PLZ, Adresse und Telefonnummer als Teil der Klasse „Person“ beginnen würde. Zusätzlich dazu benötigen Personen noch eine ID, um diese eindeutig in der Datenbank zu identifizieren. Dabei würde ich die ID und PLZ als Long speichern und den Rest als Zeichenketten. Das erlaubt es auch bei der Telefonnummer die Ländercodes mit führendem „+“ anzugeben und vermeidet Probleme bei zu langen Telefonnummern.

Für diese Daten müssen dann die Tables in der Apache-Derby Datenbank erstellt werden und die Verbindung eingerichtet werden. Um schlussendlich Daten zu verändern, zu löschen oder diese hinzuzufügen eignet sich die Verwendung des DAO Patterns. Durch die Erstellung einer DAO-Schnittstelle mit den üblichen CRUD Methoden `readForIdentity()`, `readAll()`, `create()`, `update()` und `delete()` wird die Verwaltung der Personen in der Datenbank ermöglicht.

Für jede Methode muss ein SQL-Statement formuliert werden, welche in Statements umgewandelt werden und mit Parametern befüllt werden.

Für das Testen der Datenbank eignen sich Unit-Tests sehr gut, welche für jede Methode implementiert werden müssen.

### UML



## Source-Code

## PersonDaoTest.java

```
package swp4.ue04.part1.test;

import org.testng.Assert;
import org.testng.annotations.*;
import swp4.ue04.part1.dao.Dao;
import swp4.ue04.part1.dao.impl.PersonDaoJdbc;
import swp4.ue04.part1.domain.Person;
import swp4.ue04.part1.util.ScriptRunner;

import java.util.List;

@Test
public class PersonDaoTest {

    private static final String CONNECTION_STR =
"jdbc:derby://localhost:1527/swp4;user=user;password=user";

    // Before every testing situation, reset the database
    @BeforeClass
    public void setup() {
        System.out.println("----- Resetting database -----");
        ScriptRunner.runSqlScript("sql/create_db.sql", CONNECTION_STR);
    }

    @Test
    public void testCreate() {
        // GIVEN
        Dao<Person> personDao = new PersonDaoJdbc();
        System.out.println("----- testCreate -----");
        Person person = new Person("Andrea", "Maschendrahtzaun",
"Knallerbsenstadt", 1337L, "Zaungasse 4", "+43 123456789");

        // WHEN
        boolean result = personDao.create(person);

        // THEN
        Assert.assertNotNull(person);
        Assert.assertEquals(person.getFirstName(), "Andrea");
        Assert.assertEquals(person.getLastName(), "Maschendrahtzaun");
        Assert.assertEquals(person.getCity(), "Knallerbsenstadt");
        Assert.assertEquals((long)person.getZip(), 1337L);
        Assert.assertEquals(person.getAddress(), "Zaungasse 4");
        Assert.assertEquals(person.getTel(), "+43 123456789");
        Assert.assertEquals((long)person.getId(), 1L);

        System.out.println(person);
    }

    @Test(dependsOnMethods = {"testCreate"})
    public void testReadAll() {
        //GIVEN
        Dao<Person> personDao = new PersonDaoJdbc();
        System.out.println("----- testReadAll -----");

        // WHEN
```

```
List<Person> personList = personDao.readAll();

// THEN
Assert.assertNotNull(personList);
Assert.assertEquals(personList.size(), 1);

for(Person person : personList) {
    System.out.println(person);
}
}

@Test(dependsOnMethods = {"testCreate"})
public void testReadForIdentity() {
    // GIVEN
    Dao<Person> personDao = new PersonDaoJdbc();
    System.out.println("----- testReadForIdentity -----");

    // WHEN
    Person person = personDao.readForIdentity(1L);

    // THEN
    Assert.assertNotNull(person);
    System.out.println(person);
}

@Test(dependsOnMethods = {"testCreate"})
public void testUpdate() {
    // GIVEN
    Dao<Person> personDao = new PersonDaoJdbc();
    System.out.println("----- testUpdate -----");
    Person person = new Person("Peter", "Maschendrahtzaun",
"Knallerbsenstadt", 1337L, "Zaungasse 4", "+43 123456789");
    person.setId(1L);

    // WHEN
    boolean result = personDao.update(person);

    // THEN
    Assert.assertTrue(result);
    System.out.println(person);
}

@Test(dependsOnMethods = {"testCreate", "testUpdate"})
public void testDelete() {
    // GIVEN
    Dao<Person> personDao = new PersonDaoJdbc();
    System.out.println("----- testDelete -----");

    // WHEN
    boolean result = personDao.delete(1L);

    // THEN
    Assert.assertTrue(result);
}
}
```

## Person.java

```
package swp4.ue04.part1.domain;

public class Person {
    private Long id;
    private String firstName;
    private String lastName;
    private String city;
    private Long zip;
    private String address;
    private String tel;

    public Person(String firstName, String lastName, String city, Long zip,
String address, String tel) {
        this.firstName = firstName;
        this.lastName = lastName;
        this.city = city;
        this.zip = zip;
        this.address = address;
        this.tel = tel;
    }

    public Long getId() {
        return id;
    }

    public void setId(Long id) {
        this.id = id;
    }

    public String getFirstName() {
        return firstName;
    }

    public String getLastName() {
        return lastName;
    }

    public String getCity() {
        return city;
    }

    public Long getZip() {
        return zip;
    }

    public String getAddress() {
        return address;
    }

    public String getTel() {
        return tel;
    }

    @Override
    public String toString() {
        return "Person{" +
            "id=" + id +
```

```
        ", firstName='" + firstName + '\\'' +  
        ", lastName='" + lastName + '\\'' +  
        ", city='" + city + '\\'' +  
        ", zip='" + zip +  
        ", address='" + address + '\\'' +  
        ", tel='" + tel + '\\'' +  
        '}'';  
    }  
}
```

### Dao.java

```
package swp4.ue04.part1.dao;  
  
import java.util.List;  
  
public interface Dao<T> {  
    T readForIdentity(Long identity);  
    List<T> readAll();  
    boolean create(T entity);  
  
    boolean delete(long identity);  
  
    boolean update(T entity);  
}
```

### PersonDao.java

```
package swp4.ue04.part1.dao;  
  
import swp4.ue04.part1.domain.Person;  
  
public interface PersonDao extends Dao<Person> {  
}
```

## AbstractDao.java

```
package swp4.ue04.part1.dao.impl;

import swp4.ue04.part1.dao.Dao;

import java.lang.reflect.InvocationTargetException;
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;

public abstract class AbstractDao<T> implements Dao<T> {
    // set database connection string
    protected final String DATABASE_URL =
        "jdbc:derby://localhost:1527/swp4;user=user;password=user";
    private Connection connection;

    // Check if driver is installed
    static {
        try {
            Class.forName("org.apache.derby.jdbc.ClientDriver").getConstructor(new
            Class[]{}).newInstance();
        } catch (ClassNotFoundException
            | InvocationTargetException
            | InstantiationException
            | IllegalAccessException
            | NoSuchMethodException e) {
            e.printStackTrace();
        }
    }

    // returns a new connection using the drivermanager
    protected Connection createConnection() throws SQLException {
        connection = DriverManager.getConnection(DATABASE_URL);
        return connection;
    }

    protected void closeConnection() {
        try {
            if(connection != null) {
                connection.close();
                connection = null;
            }
        } catch(SQLException e) {
            e.printStackTrace();
        }
    }
}
```



## PersonDaoJdbc.java

```
package swp4.ue04.part1.dao.impl;

import swp4.ue04.part1.dao.PersonDao;
import swp4.ue04.part1.domain.Person;

import java.sql.*;
import java.util.ArrayList;
import java.util.List;

public class PersonDaoJdbc extends AbstractDao<Person> implements PersonDao {
    private static final String TABLE_NAME = "person";
    @Override
    public Person readForIdentity(Long identity) {
        String selectSql = "SELECT * FROM "+TABLE_NAME+" WHERE id = ?";
        try(Connection connection = createConnection();
            // create statement, add parameter and execute it
            PreparedStatement statement =
connection.prepareStatement(selectSql)) {
            statement.setLong(1, identity);
            ResultSet resultSet = statement.executeQuery();
            if(resultSet.next()) {
                // if the resultset is filled, fetch all columns...
                long id = resultSet.getLong(1);
                String firstName = resultSet.getString(2);
                String lastName = resultSet.getString(3);
                String city = resultSet.getString(4);
                long zip = resultSet.getLong(5);
                String address = resultSet.getString(6);
                String tel = resultSet.getString(7);

                // and create a person
                Person person = new Person(firstName, lastName, city, zip,
address, tel);
                person.setId(id);
                return person;
            }
        } catch (SQLException throwables) {
            System.err.println("Failed to fetch person with id = "+identity);
            throwables.printStackTrace();
            return null;
        }
        return null;
    }

    @Override
    public List<Person> readAll() {
        // Similar to read by identity, but with multiple Persons
        String selectSql = "SELECT * FROM "+TABLE_NAME;
        try(Connection connection = createConnection();
            Statement statement = connection.createStatement();
            ResultSet resultSet = statement.executeQuery(selectSql)) {

            List<Person> personList = new ArrayList<>();
            while(resultSet.next()){
                long id = resultSet.getLong(1);
                String firstName = resultSet.getString(2);
                String lastName = resultSet.getString(3);
```

```
        String city = resultSet.getString(4);
        long zip = resultSet.getLong(5);
        String address = resultSet.getString(6);
        String tel = resultSet.getString(7);

        // instead of returning the person, create new persons and add
them to the list
        Person person = new Person(firstName, lastName, city, zip,
address, tel);
        person.setId(id);
        personList.add(person);
    }

    return personList;
} catch (SQLException throwables) {
    System.err.println("Failed to fetch all persons.");
    throwables.printStackTrace();
    return null;
}
}

@Override
public boolean create(Person entity) {
    // Create sql with parameters for user creation
    String createSql = "INSERT INTO "+TABLE_NAME+"(firstname, lastname,
city, zip, address, tel) VALUES(?, ?, ?, ?, ?, ?)";
    PreparedStatement ps = null;
    try {
        Connection connection = createConnection();
        // prepare a statement and tell the database to return the
generated keys
        ps = connection.prepareStatement(createSql,
Statement.RETURN_GENERATED_KEYS);

        // fill parameters
        ps.setString(1, entity.getFirstName());
        ps.setString(2, entity.getLastName());
        ps.setString(3, entity.getCity());
        ps.setLong(4, entity.getZip());
        ps.setString(5, entity.getAddress());
        ps.setString(6, entity.getTel());

        // execute the update
        int result = ps.executeUpdate();

        // check if keys have been generated (successful creation)
        try(ResultSet generatedKeys = ps.getGeneratedKeys()) {
            if(generatedKeys.next()) {
                entity.setId(generatedKeys.getLong(1));
            } else {
                throw new SQLException("Setting person.id failed, no ID
contained.");
            }
        }

        return result > 0;
    } catch (SQLException throwables) {
        System.err.println("Failed to create new person.");
    }
}
```

```

        throwables.printStackTrace();
        return false;
    }
}

@Override
public boolean delete(long identity) {
    String deleteSql = "DELETE FROM "+TABLE_NAME+" WHERE ID = ?";
    try(Connection connection = createConnection();
        PreparedStatement preparedStatement =
connection.prepareStatement(deleteSql)) {
        preparedStatement.setLong(1, identity);

        int result = preparedStatement.executeUpdate();

        // check if something has been deleted
        return result > 0;
    } catch (SQLException throwables) {
        System.err.println("Failed to remove person with id = "+identity);
        throwables.printStackTrace();
        return false;
    }
}

@Override
public boolean update(Person entity) {
    // similar to create
    String updateSql = "UPDATE "+TABLE_NAME+" SET FIRSTNAME=?, LASTNAME=?,
CITY=?, ZIP=?, ADDRESS=?, TEL=? WHERE ID=?";
    try(Connection connection = createConnection();
        PreparedStatement preparedStatement =
connection.prepareStatement(updateSql)) {
        // again, set all the parameters
        preparedStatement.setString(1, entity.getFirstName());
        preparedStatement.setString(2, entity.getLastName());
        preparedStatement.setString(3, entity.getCity());
        preparedStatement.setLong(4, entity.getZip());
        preparedStatement.setString(5, entity.getAddress());
        preparedStatement.setString(6, entity.getTel());
        preparedStatement.setLong(7, entity.getId());

        // but this time execute an update and check how many rows have
        // been altered
        int result = preparedStatement.executeUpdate();
        return result > 0;
    } catch (SQLException throwables) {
        System.err.println("Failed to update person with id =
"+entity.getId());
        throwables.printStackTrace();
        return false;
    }
}
}

```

## Testfälle

Die Testfälle wurden für diese Übung als Unit-Tests realisiert.

Der Aufruf der Testklasse zeigt folgenden Output:

```

----- Resetting database -----
Executing SQL statement -> DROP TABLE person
Executing SQL statement -> CREATE TABLE person (
    id INTEGER NOT NULL GENERATED ALWAYS AS IDENTITY,
    firstname VARCHAR(50),
    lastname VARCHAR(50),
    city VARCHAR(50),
    zip INTEGER,
    address VARCHAR(50),
    tel VARCHAR(50),
    CONSTRAINT primary_key PRIMARY KEY (id)
)
----- testCreate -----
Person{id=1, firstName='Andrea', lastName='Maschendrahtzaun', city='Knallerbsenstadt',
zip=1337, address='Zaungasse 4', tel='+43 123456789'}
----- testReadAll -----
Person{id=1, firstName='Andrea', lastName='Maschendrahtzaun', city='Knallerbsenstadt',
zip=1337, address='Zaungasse 4', tel='+43 123456789'}
----- testReadForIdentity -----
Person{id=1, firstName='Andrea', lastName='Maschendrahtzaun', city='Knallerbsenstadt',
zip=1337, address='Zaungasse 4', tel='+43 123456789'}
----- testUpdate -----
Person{id=1, firstName='Peter', lastName='Maschendrahtzaun', city='Knallerbsenstadt',
zip=1337, address='Zaungasse 4', tel='+43 123456789'}
----- testDelete -----

=====
Default Suite
Total tests run: 5, Passes: 5, Failures: 0, Skips: 0
=====

```

Oder auch als Bild:

▼ ✓ Default Suite	2 s 57 ms
▼ ✓ SWP4_Pritz_UE04	2 s 57 ms
▼ ✓ PersonDaoTest	2 s 57 ms
✓ testCreate	92 ms
✓ testReadAll	9 ms
✓ testReadForIdentity	12 ms
✓ testUpdate	18 ms
✓ testDelete	10 ms

## Teil 2 – Implementierung eines Gruppen-Chat-Clients

In diesem Teil der Übung geht es um die Implementierung eines gruppenbasierten Chat-Clients, bei dem sich User über das Netzwerk Nachrichten schicken können.

### Lösungsidee

Der Großteil der Übung wurde bereits in der Übung fertiggestellt, vollständigshalber wird aber, zusätzlich zu selbst entwickelten Klassen und Funktionen, die grobe Funktionsweise beschrieben.

Bei dieser Anwendung gibt es grundsätzlich zwei Seiten: Die Client- und die Serverseite. Der Server, welcher zuerst gestartet werden muss, erhält alle Anfragen vom Client und ist über einen Datagramsocket erreichbar. Im Herzen der Implementierung steht das Producer- und Consumer-Pattern. Clients senden („produzieren“) Nachrichten an den Server, welche dort in einem Puffer landen. Spezielle „Handler“-Klassen überprüfen regelmäßig, ob neue Nachrichten angekommen sind und führen, je nach Nachricht, andere Aktionen auf der Server-Seite aus („konsumieren“ also die Nachrichten). Nach dem Auswerten der Nachrichten werden die „Handler“-Klassen selbst zum Producer und generieren notwendigenfalls Nachrichten, welche zurück an den Client gesendet werden.

Zu Beginn des selbstständigen Ausarbeitens war das Senden und Empfangen von Nachrichten schon möglich, und Clients konnten sich bereits am Server anmelden. Der Hauptteil dieser Übung besteht somit aus dem Implementieren der Handler für jeden Nachrichtentyp und der Abwicklung bzw. Anzeige der Nachrichten am Client selbst.

Es sind noch folgende Handler zu erstellen:

- Senden bzw. Verteilen von empfangenen Nachrichten
- „Ausloggen“ aus dem Server
- Monitoring der Aktivität der Clients
- Fehlermeldungen

Beim Senden der Nachricht mit Command „s“, muss die Gruppe bzw. der Chatroom des Senders bestimmt werden und eine Nachricht mit Command „m“ an jeden Client in derselben Gruppe adressiert werden. Falls der User nicht eingeloggt ist, könnte gleich eine Fehlermeldung geschrieben werden. Zusätzlich dazu würde ich verlangen, dass die Nachricht einem Muster „<Name>:<Nachricht>“ folgen muss, also zumindest einen Doppelpunkt enthalten muss.

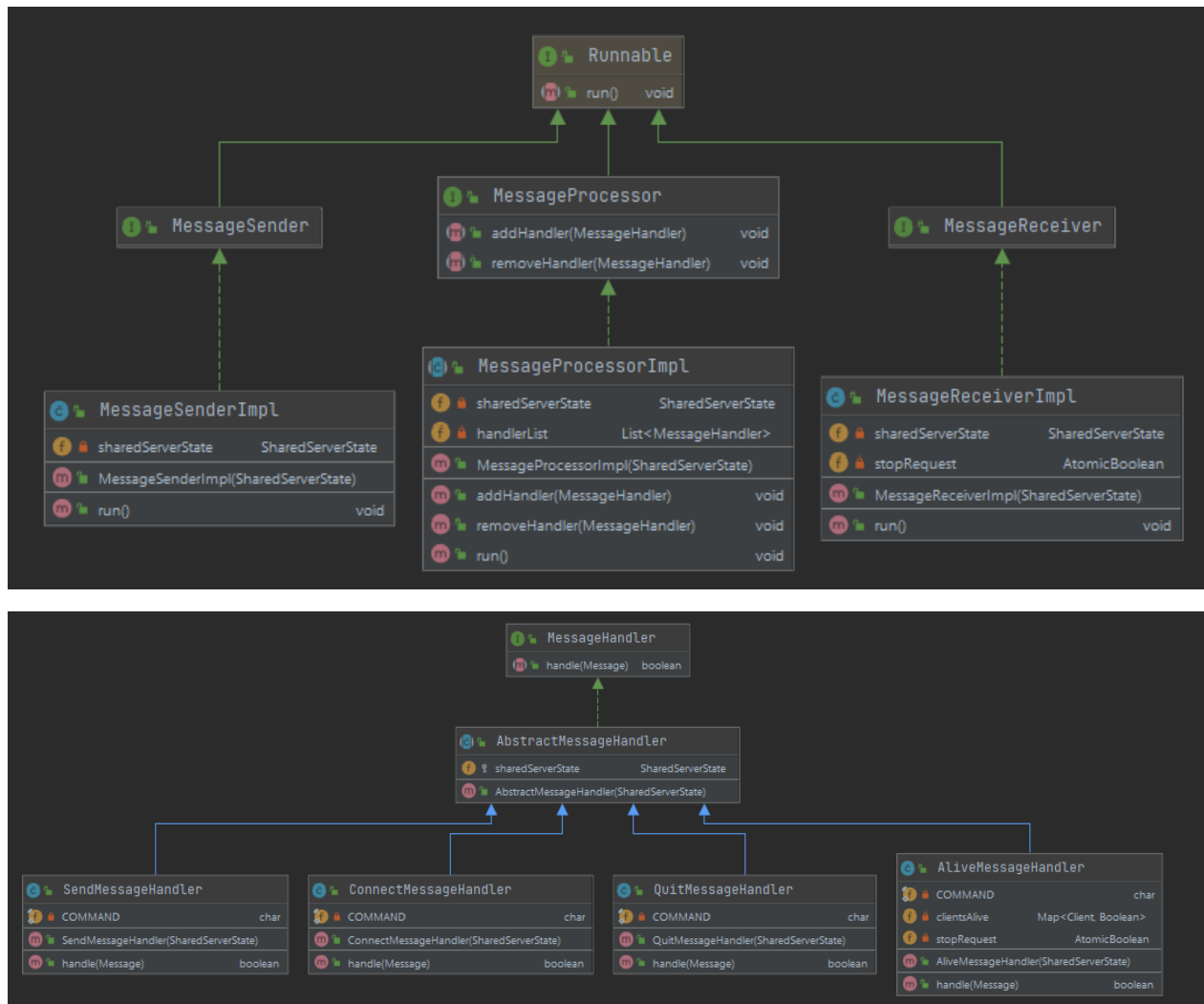
Beim Ausloggen aus dem Server wird einfach eine Message mit „q“ als Command an den Server geschickt und die Client-Anwendung beendet. Der Server entfernt zusätzlich den Client aus dem ServerState.

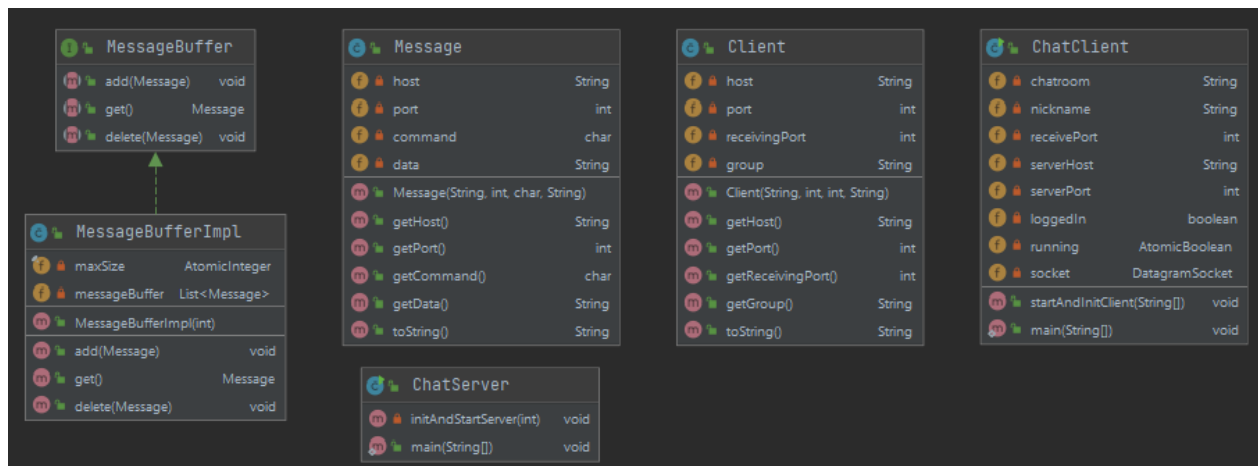
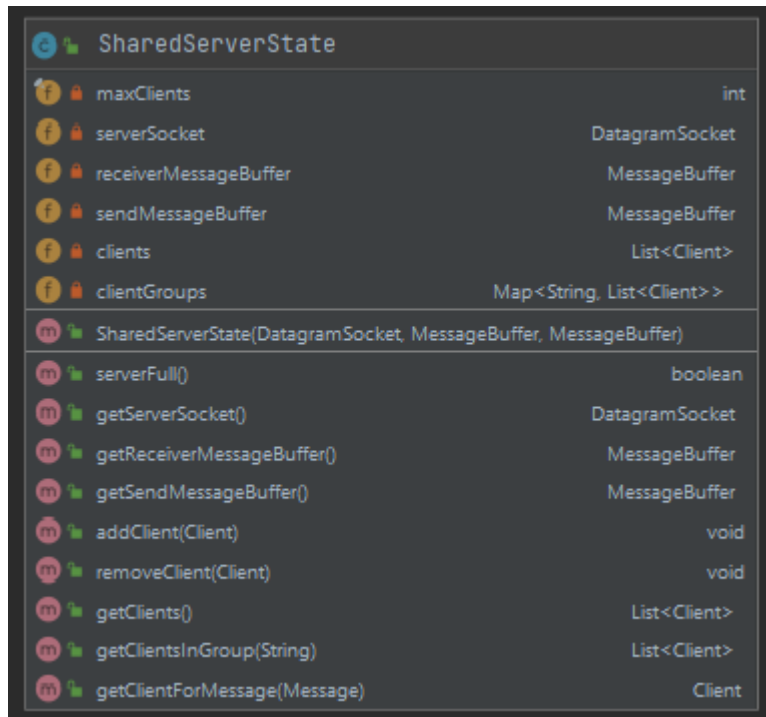
Für das Monitoring der Aktivität würde ich einerseits einen eigenen Thread starten, der alle 120 Sekunden ein Paket mit dem „a“ Command ausschickt und nach 60 Sekunden prüft, welche Clients mit „o“ geantwortet haben. Die Differenz dieser Clients und der eingeloggten Clients wird dann aus den Clients entfernt und den entsprechenden Clients eine Fehlermeldung geschickt, sodass diese über das Ausloggen informiert werden.

Die Client-Anwendung muss die Messages je nach Command anders abarbeiten. Zum Beispiel interessiert es den Anwender nicht, dass der Server „Alive“-Messages in regelmäßigen Abständen schickt. Das ermöglicht auch eine visuelle Aufbereitung des Inhalts.

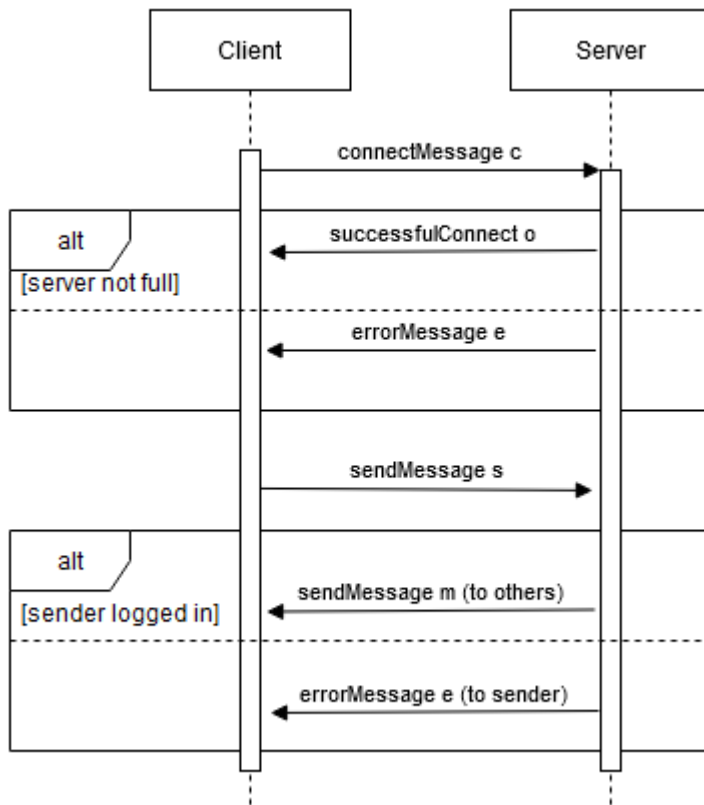
## UML

Dadurch, dass die meisten Klassen bereits gegeben waren, die Anzahl der Klassen sehr hoch ist habe ich das Diagramm direkt mit IntelliJ IDEA generiert:





## Sequenzdiagramm





## Source-Code

## ChatServer.java

```
package swp4.ue04.part2;

import swp4.ue04.part2.buffer.MessageBuffer;
import swp4.ue04.part2.buffer.impl.MessageBufferImpl;
import swp4.ue04.part2.handler.impl.AliveMessageHandler;
import swp4.ue04.part2.handler.impl.ConnectMessageHandler;
import swp4.ue04.part2.handler.impl.QuitMessageHandler;
import swp4.ue04.part2.handler.impl.SendMessageHandler;
import swp4.ue04.part2.process.MessageProcessor;
import swp4.ue04.part2.process.impl.MessageProcessorImpl;
import swp4.ue04.part2.receiver.MessageReceiver;
import swp4.ue04.part2.receiver.impl.MessageReceiverImpl;
import swp4.ue04.part2.sender.MessageSender;
import swp4.ue04.part2.sender.impl.MessageSenderImpl;

import java.net.DatagramSocket;
import java.net.SocketException;

public class ChatServer {

    private void initAndStartServer( int serverPort ) {
        try {
            // create a serversocket to receive and send messages from
            DatagramSocket serverSocket = new DatagramSocket( serverPort );

            // then create messagebuffers to intermediately store the messages
            // before processing or sending them
            MessageBuffer receiveBuffer = new MessageBufferImpl( 100 );
            MessageBuffer sendBuffer = new MessageBufferImpl( 100 );

            // create a sharedserverstate to manage clients and buffers
            // centrally
            SharedServerState sharedServerState = new SharedServerState(
                serverSocket, receiveBuffer, sendBuffer );

            // create instances to receive, send and process messages
            MessageReceiver receiver = new MessageReceiverImpl(
                sharedServerState );
            MessageSender sender = new MessageSenderImpl( sharedServerState );
            MessageProcessor processor = new MessageProcessorImpl(
                sharedServerState );

            // add all the different message handlers for different message
            // types
            processor.addHandler( new ConnectMessageHandler(
                sharedServerState ) );
            processor.addHandler( new SendMessageHandler( sharedServerState ) );
            processor.addHandler( new QuitMessageHandler( sharedServerState ) );
            processor.addHandler( new AliveMessageHandler( sharedServerState ) );

            // create and start threads for receiving, sending and processing
```

```
packets
    Thread receiverThread = new Thread( receiver );
    Thread senderThread = new Thread( sender );
    Thread processorThread = new Thread( processor );
    processorThread.start();
    senderThread.start();
    receiverThread.start();

    // before ending, wait for all threads to end.
    receiverThread.join();
} catch( InterruptedException | SocketException e ) {
    System.err.println( "Unable to start ChatServer, verify if port is
already in use: " + serverPort );
}

}

public static void main(String[] args) {
    new ChatServer().initAndStartServer( 9999 );
}

}
```

## ChatClient.java

```
package swp4.ue04.part2;

import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.net.*;
import java.util.concurrent.atomic.AtomicBoolean;

public class ChatClient {

    private String chatroom;
    private String nickname;
    private int receivePort;
    private String serverHost;
    private int serverPort;
    private boolean loggedIn = false;
    private AtomicBoolean running = new AtomicBoolean( true );
    private DatagramSocket socket;

    public void startAndInitClient( String[] args )    {
        try {
            // take argument from console
            chatroom = args[0];
            nickname = args[1];
            receivePort = Integer.valueOf(args[2]);
            serverHost = args[3];
            serverPort = Integer.valueOf(args[4]);
            socket = new DatagramSocket();

            // create a new thread for the blocking method "receive" of the
            socket

            Thread thread = new Thread(() -> {
                try (DatagramSocket receiver = new
                DatagramSocket(receivePort)) {
                    while (running.get()) {
                        byte[] buffer = new byte[2048];
                        DatagramPacket pkt = new DatagramPacket(buffer,
                        buffer.length);

                        //System.out.print("Client waiting for incoming
                        messages in group: " + chatroom + " - ");
                        receiver.receive(pkt);

                        // depending on the message start, process the
                        messages differently

                        String data = new String(buffer).trim();
                        switch(data.substring(0,1)) {
                            case "o":
                                loggedIn = true;
                                System.out.println("Connected to "+chatroom+"
                                (Local: "+data.substring(1)+")");
                                break;
                            case "m":
                                System.out.println(data.substring(1));
                                break;
                            case "e":
                                System.err.println("Error:
                                "+data.substring(1));

```

```

// error 1: connection limit, error 2: logged
out because of missing alive
        if(data.charAt(1)=='1' || data.charAt(1)=='2')
{
    loggedIn = false;
}
break;
case "a":
    //System.out.println("Are you still alive?");
    byte[] m = ("o").getBytes();
    socket.send(new DatagramPacket(m, m.length,
InetAddress.getByName(serverHost), serverPort));
    break;
}

}
} catch (IOException e) {
    System.err.println( "Failed to receive Paket from
server..." );
}

});
// make the thread a daemon, so it is being terminated along with
the main thread.
thread.setDaemon(true);
thread.start();

// first of all, send a connect message to the server
String ConnPacket = "c " + receivePort + " " + chatroom;
InetAddress addr = InetAddress.getByName(serverHost);
byte[] ConnMessage = ConnPacket.getBytes();
DatagramPacket ConnPkt = new DatagramPacket(ConnMessage,
ConnMessage.length, addr, serverPort);
socket.send(ConnPkt);
Thread.sleep(1000);

// after connecting, continuously read from the console
BufferedReader reader = new BufferedReader(new
InputStreamReader(System.in));
while(running.get()) {
    String userdata = reader.readLine().trim();
    if(loggedIn) {
        byte[] m;
        // if the sent message is the letter q, quit the client
application
        if (userdata.equals("q")) {
            m = ("q").getBytes();
            running.set(false);
            System.out.println("You've been logged out.");
        } else {
            // if it isn't q, send a full message and the nickname
            m = ("s <" + nickname + ">: " + userdata).getBytes();
        }
        if(userdata.length() > 0) {
            DatagramPacket SendPkt = new DatagramPacket(m,
m.length, addr, serverPort);
            socket.send(SendPkt);
        }
    }
}

```

```
        } else { // if the user isn't logged in and tries to send a
message, he gets an error message and the program terminates
        System.err.println("You are not logged in. Please try
reconnecting.");
        running.set(false);
    }
}

    } catch( SocketException e ) {
        running.set( false );
        System.err.println( "The client could not be started for server "
+ serverHost
        + ":" + serverPort + ". Is the server running?" );
        e.printStackTrace();

    } catch( UnknownHostException e ) {
        running.set( false );
        System.err.println( "The host " + serverHost + " could not be
found!" );
        e.printStackTrace();
    } catch( InterruptedException | IOException e ) {
        running.set( false );
        e.printStackTrace();
    }
}

    public static void main(String[] args) throws Exception {
        new ChatClient( ).startAndInitClient( args );
    }
}
```

## Client.java

```
package swp4.ue04.part2;

public class Client {
    private String host;
    private int port;
    private int receivingPort;
    private String group;

    public Client(String host, int port, int receivingPort, String group) {
        this.host = host;
        this.port = port;
        this.receivingPort = receivingPort;
        this.group = group;
    }

    public String getHost() {
        return host;
    }

    public int getPort() {
        return port;
    }

    public int getReceivingPort() {
        return receivingPort;
    }

    public String getGroup() {
        return group;
    }

    @Override
    public String toString() {
        return "Client{" +
            "host='" + host + '\'' +
            ", port=" + port +
            ", receivingPort=" + receivingPort +
            ", group='" + group + '\'' +
            '}';
    }
}
```

## SharedServerState.java

```
package swp4.ue04.part2;

import swp4.ue04.part2.buffer.MessageBuffer;
import swp4.ue04.part2.message.Message;

import java.net.DatagramSocket;
import java.util.*;

public class SharedServerState {

    private final int maxClients = 10;

    private DatagramSocket serverSocket;
    private MessageBuffer receiverMessageBuffer;
    private MessageBuffer sendMessageBuffer;
    private List<Client> clients = Collections.synchronizedList(new
ArrayList<>());
    private Map<String, List<Client>> clientGroups =
Collections.synchronizedMap(new HashMap<>());

    public SharedServerState(DatagramSocket serverSocket, MessageBuffer
receiverMessageBuffer, MessageBuffer sendMessageBuffer) {
        // initialize the serverstate using parameters
        this.serverSocket = serverSocket;
        this.receiverMessageBuffer = receiverMessageBuffer;
        this.sendMessageBuffer = sendMessageBuffer;
    }

    public synchronized boolean serverFull() {
        return clients.size() == maxClients;
    }

    public synchronized DatagramSocket getServerSocket() {
        return serverSocket;
    }

    public synchronized MessageBuffer getReceiverMessageBuffer() {
        return receiverMessageBuffer;
    }

    public synchronized MessageBuffer getSendMessageBuffer() {
        return sendMessageBuffer;
    }

    public synchronized void addClient(Client client) {
        // add clients only if the server has capacity
        if(!serverFull()) {
            this.clients.add(client);
            this.clientGroups.computeIfAbsent(client.getGroup(), tmp -> new
ArrayList<>());
            this.clientGroups.get(client.getGroup()).add(client);
        }
    }

    public synchronized void removeClient(Client client) {
        // if disconnecting client is the last one in the group, remove the
group
    }
```

```
        if (getClientsInGroup(client.getGroup()).size() == 1) {
            clientGroups.remove(client.getGroup());
        }
        clients.remove(client);
    }

    public List<Client> getClients() {
        return clients;
    }

    public synchronized List<Client> getClientsInGroup(String group) {
        return this.clientGroups.get(group);
    }

    public synchronized Client getClientForMessage(Message message) {
        for (Client client : clients) {
            if (client.getPort() == message.getPort() &&
client.getHost().equals(message.getHost())) {
                return client;
            }
        }

        return null;
    }
}
```



## Message.java

```

package swp4.ue04.part2.message;

public class Message {
    private String host;
    private int port;
    private char command;
    private String data;

    public Message(String host, int port, char command, String data) {
        this.host = host;
        this.port = port;
        this.command = command;
        this.data = data;
    }

    public String getHost() {
        return host;
    }
    public int getPort() {
        return port;
    }
    public char getCommand() {
        return command;
    }
    public String getData() {
        return data;
    }

    @Override
    public String toString() {
        return "Message{" +
            "host='" + host + '\'' +
            ", port=" + port +
            ", command=" + command +
            ", data='" + (data != null ? data.trim() : "<NO DATA>") + '\'' +
            '}';
    }
}

```

## MessageBuffer.java

```

package swp4.ue04.part2.buffer;

import swp4.ue04.part2.message.Message;

public interface MessageBuffer {
    // adds a message to the buffer
    void add(Message message);
    // retrieves a message from the buffer
    Message get();
    // deletes a message from the buffer
    void delete(Message message);
}

```

## MessageBufferImpl.java

```
package swp4.ue04.part2.buffer.impl;

import swp4.ue04.part2.buffer.MessageBuffer;
import swp4.ue04.part2.message.Message;
import java.util.ArrayList;
import java.util.Collections;
import java.util.List;
import java.util.concurrent.atomic.AtomicInteger;

public class MessageBufferImpl implements MessageBuffer {

    private final AtomicInteger maxSize;
    // due to multithreading we need a synchronizedList
    private List<Message> messageBuffer = Collections.synchronizedList(new
ArrayList<>());

    public MessageBufferImpl(int maxSize) {
        this.maxSize = new AtomicInteger(maxSize);
    }

    @Override
    public synchronized void add(Message message) {
        // if the list is full, wait till something is removed
        while(messageBuffer.size() == maxSize.get()) {
            try {
                wait(200);
            } catch (InterruptedException e) {

            }
        }

        // afterwards, remove add the message
        messageBuffer.add(message);
        notifyAll();
    }

    @Override
    public synchronized Message get() {
        // if the messageBuffer is empty, wait till there is an element
        while(messageBuffer.size() < 1) {
            try {
                wait(200);
            } catch (InterruptedException e) {

            }
        }
        return messageBuffer.get(0);
    }

    @Override
    public synchronized void delete(Message message) {
        // remove a message from the List
        messageBuffer.remove(message);
        notifyAll();
    }
}
```

## MessageSender.java

```
package swp4.ue04.part2.sender;

public interface MessageSender extends Runnable {
}
```

## MessageSenderImpl.java

```
package swp4.ue04.part2.sender.impl;

import swp4.ue04.part2.SharedServerState;
import swp4.ue04.part2.message.Message;
import swp4.ue04.part2.sender.MessageSender;

import java.io.IOException;
import java.net.DatagramPacket;
import java.net.InetAddress;

public class MessageSenderImpl implements MessageSender {

    private SharedServerState sharedServerState;

    public MessageSenderImpl(SharedServerState sharedServerState) {
        this.sharedServerState = sharedServerState;
    }

    @Override
    public void run() {
        // implement Consumer logic
        Message message;

        // periodically check for new messages in the buffer
        while( ( message = sharedServerState.getSendMessageBuffer().get() ) !=
null ) {
            try {
                sharedServerState.getSendMessageBuffer().delete(message);
                byte[] messageRaw = (message.getCommand() +
message.getData()).getBytes();
                DatagramPacket pkt = new DatagramPacket(
                    messageRaw,
                    messageRaw.length,
                    InetAddress.getByName(message.getHost()),
                    message.getPort()
                );

                // if message has been found, send the packets from the buffer
                sharedServerState.getServerSocket().send(pkt);
            } catch( IOException e ) {
                System.err.println( "Failed to send message: " + message );
            }
        }
    }
}
```

## MessageReceiver.java

```
package swp4.ue04.part2.receiver;

public interface MessageReceiver extends Runnable {
}
```

## MessageReceiverImpl.java

```
package swp4.ue04.part2.receiver.impl;

import swp4.ue04.part2.SharedServerState;
import swp4.ue04.part2.message.Message;
import swp4.ue04.part2.receiver.MessageReceiver;
import java.io.IOException;
import java.net.DatagramPacket;
import java.util.concurrent.atomic.AtomicBoolean;

public class MessageReceiverImpl implements MessageReceiver {

    private SharedServerState sharedServerState;
    private AtomicBoolean stopRequest = new AtomicBoolean(false);

    public MessageReceiverImpl(SharedServerState sharedServerState) {
        this.sharedServerState = sharedServerState;
    }

    @Override
    public void run() {

        DatagramPacket pkt;
        // while there has not been a stopRequest, wait for messages from
clients
        while(!stopRequest.get()) {
            try {
                byte[] buffer = new byte[2048];
                pkt = new DatagramPacket(buffer, buffer.length);
                sharedServerState.getServerSocket().receive(pkt);
                String message = new String(buffer, 0 , buffer.length);

                // if as message has been received, check if a command is
present and add
it to the buffer
                if(message.length()>0) {
                    char command = message.charAt(0);
                    if(message.length() > 1) {
                        sharedServerState.getReceiverMessageBuffer().add(
                            new Message(pkt.getAddress().getHostName(),
pkt.getPort(), command, message.substring(1).trim())
                        );
                    }
                }
            } catch(IOException e) {
                System.err.println("Received an error processing the buffer.
"+e.getMessage());
            }
        }
    }
}
```

## MessageProcessor.java

```
package swp4.ue04.part2.process;

import swp4.ue04.part2.handler.MessageHandler;

public interface MessageProcessor extends Runnable{

    void addHandler(MessageHandler handler);
    void removeHandler(MessageHandler handler);
}
```

## MessageProcessorImpl.java

```
package swp4.ue04.part2.process.impl;

import swp4.ue04.part2.SharedServerState;
import swp4.ue04.part2.handler.MessageHandler;
import swp4.ue04.part2.message.Message;
import swp4.ue04.part2.process.MessageProcessor;

import java.util.ArrayList;
import java.util.Collections;
import java.util.List;

public class MessageProcessorImpl implements MessageProcessor {

    private SharedServerState sharedServerState;
    private List<MessageHandler> handlerList =
Collections.synchronizedList(new ArrayList<>());

    public MessageProcessorImpl(SharedServerState sharedServerState) {
        this.sharedServerState = sharedServerState;
    }

    @Override
    public void addHandler(MessageHandler handler) {
        handlerList.add(handler);
    }

    @Override
    public void removeHandler(MessageHandler handler) {
        handlerList.remove(handler);
    }

    @Override
    public void run() {
        Message message;
        // check periodically if there are new messages
        while((message = sharedServerState.getReceiverMessageBuffer().get())
!= null) {
            try {
                sharedServerState.getReceiverMessageBuffer().delete(message);
                System.out.println("MessageProcessor received message:
"+message);

                // try to process it with every messageHandler until the right
```

```

one has been found
        for(MessageHandler handler : handlerList) {
            if(handler.handle(message)) {
                System.out.println("Handler "+handler+" handled
message: "+message);
            }
        }

        Thread.sleep(200);
    } catch (InterruptedException e) {}
}
}
}

```

### MessageHandler.java

```

package swp4.ue04.part2.handler;

import swp4.ue04.part2.message.Message;

public interface MessageHandler {
    public boolean handle(Message message);
}

```

### AbstractMessageHandler.java

```

package swp4.ue04.part2.handler;

import swp4.ue04.part2.SharedServerState;

public abstract class AbstractMessageHandler implements MessageHandler {

    protected SharedServerState sharedServerState;

    // every MessageHandler needs the serverstate to interact with clients
    public AbstractMessageHandler(SharedServerState sharedServerState) {
        this.sharedServerState = sharedServerState;
    }
}

```

## ConnectMessageHandler.java

```
package swp4.ue04.part2.handler.impl;

import swp4.ue04.part2.Client;
import swp4.ue04.part2.SharedServerState;
import swp4.ue04.part2.handler.AbstractMessageHandler;
import swp4.ue04.part2.message.Message;

public class ConnectMessageHandler extends AbstractMessageHandler {

    private static final char COMMAND = 'c';

    public ConnectMessageHandler(SharedServerState sharedServerState) {
        super(sharedServerState);
    }

    @Override
    public boolean handle(Message message) {
        if(COMMAND == message.getCommand()) {
            // message.getData() 1234 group_1
            String[] data = message.getData().split(" ");
            // connect message consists of port and group. both are needed,
            // else we have an error.
            if(data.length == 2) {

                Client client = new Client(message.getHost(),
                    message.getPort(), Integer.parseInt(data[0]), data[1]);

                // add the client to the server (login) only if the server
                // still has capacity.
                if(!sharedServerState.serverFull()) {
                    sharedServerState.addClient(client);

                    // send an "o" message back to the client if they have
                    // been accepted
                    sharedServerState.getSendMessageBuffer().add(
                        new Message(client.getHost(),
                            client.getReceivingPort(), 'o', client.getHost() + ":" + client.getPort())
                    );
                } else {
                    // if too many people are on the server, send an error
                    // message
                    sharedServerState.getSendMessageBuffer().add(
                        new Message(client.getHost(),
                            client.getReceivingPort(), 'e', "1: Server capacity reached.")
                    );
                }
                return true;
            } else {
                System.err.println("Invalid Connect message from client: " +
                    message);
            }
        }

        return false;
    }
}
```

## SendMessageHandler.java

```

package swp4.ue04.part2.handler.impl;

import swp4.ue04.part2.Client;
import swp4.ue04.part2.SharedServerState;
import swp4.ue04.part2.handler.AbstractMessageHandler;
import swp4.ue04.part2.message.Message;

public class SendMessageHandler extends AbstractMessageHandler {

    private static final char COMMAND = 's';

    public SendMessageHandler(SharedServerState sharedServerState) {
        super(sharedServerState);
    }

    @Override
    public boolean handle(Message message) {
        if (COMMAND == message.getCommand()) {
            String data = message.getData().trim();
            // every message needs to contain a colon
            if (data.contains(":")) {
                Client sender =
sharedServerState.getClientForMessage(message);

                // if client is logged in
                if (sender != null) {
                    // distribute the message to all clients in the same group
as the sender
                    for (Client receiver :
sharedServerState.getClientsInGroup(sender.getGroup())) {
                        if (sender != receiver) {
                            sharedServerState.getSendMessageBuffer().add(
                                new Message(receiver.getHost(),
receiver.getReceivingPort(), 'm', data)
                            );
                        }
                    }
                } else {
                    // if the user isnt logged in, send an error message
                    sharedServerState.getSendMessageBuffer().add(
                        new Message(message.getHost(), message.getPort(),
'e', "2: You are not logged in.")
                    );
                }

                return true;
            } else {
                System.err.println("Invalid Send message from client: " +
message);
            }
        }

        return false;
    }
}

```



## AliveMessageHandler.java

```
package swp4.ue04.part2.handler.impl;

import swp4.ue04.part2.Client;
import swp4.ue04.part2.SharedServerState;
import swp4.ue04.part2.handler.AbstractMessageHandler;
import swp4.ue04.part2.message.Message;

import java.util.Collections;
import java.util.HashMap;
import java.util.Map;
import java.util.concurrent.atomic.AtomicBoolean;

public class AliveMessageHandler extends AbstractMessageHandler {

    private static final char COMMAND = 'o';

    // List to manage active clients and detect absent clients
    private Map<Client, Boolean> clientsAlive =
Collections.synchronizedMap(new HashMap<>());
    private AtomicBoolean stopRequest = new AtomicBoolean(false);

    public AliveMessageHandler(SharedServerState sharedServerState) {
        super(sharedServerState);
    }

    // create a new thread to monitor clients and remove inactive ones
    Thread aliveThread = new Thread( () -> {
        try {
            while(!stopRequest.get()) {
                // first, copy a list of all clients to the map and set
                // their activity to inactive (false)
                // and also send each client a message to ask about their
                // activity
                for(Client curClient : sharedServerState.getClients()) {
                    sharedServerState.getSendMessageBuffer().add(
                        new Message(curClient.getHost(),
curClient.getReceivingPort(), 'a', ""))
                    );
                    clientsAlive.put(curClient, false);
                }
                // Then, wait 60 seconds and afterwards check how many
                // clients responded
                Thread.sleep(5000);

                // if clients havent responded, their activity is still
                // inactive (false)
                // they will then be removed and notified with an error
                for(Map.Entry<Client, Boolean> clientInfo :
clientsAlive.entrySet()) {
                    if(!(clientInfo.getValue())) {
sharedServerState.removeClient(clientInfo.getKey());
                        sharedServerState.getSendMessageBuffer().add(
                            new Message(clientInfo.getKey().getHost(),
clientInfo.getKey().getReceivingPort(), 'e', "2: You've been disconnected")
                        );
                    }
                }
            }
        } catch (InterruptedException e) {
            // Thread interrupted
        }
    });
    aliveThread.start();
}
```

```
        clientsAlive.remove(clientInfo.getKey());
    }
}
// afterwards, wait another 60 seconds to have an interval
of 120s
Thread.sleep(5000);
}
} catch (InterruptedException e) {
    e.printStackTrace();
}
});
aliveThread.start();
}

@Override
public boolean handle(Message message) {
    if(COMMAND == message.getCommand()) {
        // if the server receives a message with command "o" from the
client,
        // get the right client corresponding to the message
        // then set its activity to active in the list (= true)
        clientsAlive.put(sharedServerState.getClientForMessage(message),
true);
        return true;
    }
    return false;
}
}
```

## QuitMessageHandler.java

```
package swp4.ue04.part2.handler.impl;

import swp4.ue04.part2.SharedServerState;
import swp4.ue04.part2.handler.AbstractMessageHandler;
import swp4.ue04.part2.message.Message;

public class QuitMessageHandler extends AbstractMessageHandler {

    private static final char COMMAND = 'q';

    public QuitMessageHandler(SharedServerState sharedServerState) {
        super(sharedServerState);
    }

    @Override
    public boolean handle(Message message) {
        if(COMMAND == message.getCommand()) {
            // quit message has no data, just remove the client
            sharedServerState.removeClient(
                sharedServerState.getClientForMessage(message)
            );
        }
        return false;
    }
}
```

## Testfälle

Da die Testfälle schwer zu streamlinen sind, wurde dieser Teil nur mittels Beschreibungen und zusätzlichen Screenshots/Ausgaben dokumentiert.

## Parade-Beispiel

Startet man den Server und zwei Clients mit demselben Chatroom als Argument, so können diese direkt miteinander schreiben und sich auch logischerweise mit der Message „q“ ausloggen.

```
Connected to SchwupsiDupsi-Lounge (Local: 127.0.0.1:51347)
Servus, noch jemand wach?
<Jochen>: Jo, sicher. Programmieraufgabe schon fertig?
Ja, danke der Nachfrage. Heute sogar vor 23:59! ;)
q
You've been logged out.

Process finished with exit code 0
```

Das Gegenüber sieht natürlich genau das umgekehrte, was die Farben und Namen betrifft.

```
Connected to SchwupsiDupsi-Lounge (Local: 127.0.0.1:51348)
<Sebastian>: Servus, noch jemand wach?
Jo, sicher. Programmieraufgabe schon fertig?
<Sebastian>: Ja, danke der Nachfrage. Heute sogar vor 23:59! ;)
q
You've been logged out.

Process finished with exit code 0
```

### Maximale Client-Anzahl am Server

Sind bereits zu viele Clients am Server eingeloggt, so wird ein Error zurückgeschickt und nicht eingeloggt. Für diesen Test wurde die maximale Anzahl auf 1 gesetzt.

Während sich der erste Client anmelden kann, erhält der zweite Client folgenden Fehler:

```
Error: 1: Server capacity reached.  
Was soll das? Es ist doch nur Sebastian eingeloggt!!!  
You are not logged in. Please try reconnecting.  
  
Process finished with exit code 0
```

Der erste Client bekommt von allem nichts mit und erhält auch keine Nachricht.

```
Connected to SchwupsiDupsi-Lounge (Local: 127.0.0.1:56860)  
|
```

### Fehlende Antwort auf Alive-Message

Melden sich die Clients nicht innerhalb des Rahmens für Alive-Messages, so erhalten diese eine Fehlermeldung (falls sie diese überhaupt empfangen können und nicht wirklich einen Disconnect haben) und werden vom Server abgemeldet. Dieser Prozess wurde durch das Auskommentieren der „o“-Antwort-Messages getestet.

```
Connected to SchwupsiDupsi-Lounge (Local: 127.0.0.1:55467)  
Error: 2: You've been disconnected  
Warum wurde ich disconnected??? :(  
You are not logged in. Please try reconnecting.  
  
Process finished with exit code 0
```

Leere Nachricht

Schickt der User eine „leere Nachricht“ so fängt die Client-Anwendung das ab und die gegenüberliegende Seite bekommt keine Nachricht.

```
C:\Users\basti\.jdk8\corretto-15.0.2\bin\java.exe ...  
Connected to SchwupsiDupsi-Lounge (Local: 127.0.0.1:62245)  
|
```

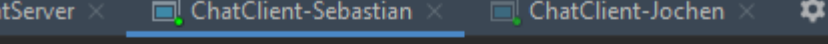
```
C:\Users\basti\.jdk\corretto-15.0.2\bin\java.exe ...  
Connected to SchwupsiDupsi-Lounge (Local: 127.0.0.1:62246)
```

## Zu lange Nachrichten

Wird ein String mit zu großer Länge (größer als der Buffer) verschickt, so kommen nur die ersten 2048 Bytes des Pakets an.

[illegible]

Sebastian sieht jedoch das letzte b nicht, da der String abgeschnitten wurde.



The screenshot shows a Java IDE with three tabs: 'ChatServer', 'ChatClient-Sebastian', and 'ChatClient-Jochen'. The 'ChatClient-Sebastian' tab is active and displays a long string of 'a' characters, indicating a successful connection and data reception. The 'ChatClient-Jochen' tab is also visible, showing a similar string of 'a' characters. The 'ChatServer' tab is partially visible on the left.