Übung 1: Chatroom mit *socket*

Wir werden alle Technologie, die wir in kvan besprechen werden, in einer *Chat* Applikation einbinden, welche ein Chatroom implementiert. Chat Teilnehmer sollen (1) neue Themen (Topics) anmelden, (2) ein Thema auswählen und (3) kurze Kommentare an alle beim besagten Thema eingeschriebenen Teilnehmer senden können.
Ein Chat Teilnehmer muss sich beim Chatroom (der eigentliche Server) anmelden. Sobald dies erfolgt ist, kann er ein beliebiges Thema auswählen und sofort anfangen, seine Weisheiten an die anderen Teilnehmer zu unterbreiten.

Damit Sie sich voll auf die jeweilige Technologie für die Kommunikation konzentrieren können, haben wir die ganze GUI Umgebung bereits implementiert. Was Sie dann genau implementieren sollen, ist in einer Serie von Interfaces spezifiziert.
(Fast) alle Interface-Methoden deklarieren die *java.io.IOException*, die geworfen wird, wenn bei der Kommunikation Probleme auftreten.

```
public interface ChatDriver {

	/**
	 * Connects to an implementation of a chat room.
	 *
	 * @param host
	 *            host name
	 * @param port
	 *            port number
	 * @throws IOException
	 *             if a remote or communication problem occurs
	 */
	public void connect(String host, int port) throws IOException;

	/**
	 * Disconnects from the chat room server .
	 *
	 * @throws IOException
	 *             if a remote or communication problem occurs
	 */
	public void disconnect() throws IOException;

	/**
	 * Return the chat room which is operated by that server. Before getChatRoom
	 * can be invoked, connect must be called. getChatRoom must have singleton
	 * semantics, i.e. it should return the same instance upon subsequent calls.
	 * After disconnect has been called getChatRoom returns null.
	 *
	 * @see #connect
	 * @see Chat
	 */
	public ChatRoom getChatRoom();
}


public interface ChatRoom {

	/**
	 * Add a participant who joined the chat room.
	 *
	 * @param name
	 *            The name of the participant
	 * @throws IOException
	 *             if a remote or communication problem occurs
	 */
```

```
        public boolean addParticipant(String name) throws IOException;

        /**
         * Remove a participant who left the chat room.
         *
         * @param name
         *            The name of the participant
         * @throws IOException
         *             if a remote or communication problem occurs
         */
        public boolean removeParticipant(String name) throws IOException;



        /**
         * Add a topic to the chat room.
         *
         * @param name
         *            The string defining the topic
         * @throws IOException
         *             if a remote or communication problem occurs
         */
        public boolean addTopic(String topic) throws IOException;

        /**
         * Remove a topic from the chat room.
         *
         * @param name
         *            The string defining the topic
         * @throws IOException
         *             if a remote or communication problem occurs
         */
        public boolean removeTopic(String topic) throws IOException;

        /**
         * Sends a message to the chat room.
         *
         * @param topic
         *            The string defining the topic
         * @param message
         *            The message
         * @throws IOException
         *             if a remote or communication problem occurs
         */
        public boolean sendMessage(String topic, String message) throws IOException;

        /**
         * Get last ten messages from the chat room.
         *
         * @param topic
         *            The string defining the topic
         * @returns String
         *            Last ten messages on that topic
         * @throws IOException
         *             if a remote or communication problem occurs
         */
        public String getMessages(String topic) throws IOException;

        /**
         * Refresh last ten messages from the chat room; refresh participants and
topics, too.
         *
```

```
         * @param topic
         * The string defining the current topic
         * @returns String
         *      Last ten messages on that topic
         * @throws IOException
         *     if a remote or communication problem occurs
         */
        public String refreshMessages(String topic) throws IOException;
}
```

Lösung-Skizze

Der Server erstellt eine Instanz der Klasse `ChatRoomDriver` und ruft ihre Methode `connect()` auf, die ihrerseits eine Instanz der `ChatRoom` Klasse erstellt . Es gibt nur eine Instanz von `ChatRoom`, die alle Klienten, die alle möglichen Gesprächsthemen, und die alle eingehenden Nachrichten verwaltet. Der Server erzeugt einen Server-Socket, um mit seinen Klienten kommunizieren zu können. Zusätzlich erstellt er einen separaten Thread, `ConnectionListener`, das alle aktiven Socket-Verbindungen zu den verschiedenen Klienten verwaltet.
Danach wartet er lediglich, ob sich ein neuer Klient bei ihm anmeldet. Sobald der Server einen neuen Klienten akzeptiert, wird ein neuer Klient-Socket für diesen kreiert. Jeder neue Klient erhält vom Server ein separaten Thread via die `ConnectionHandler` Klasse. Diese Klasse kriegt vom Server als Parameter eine Referenz auf das Socket- und eine andere auf die `ConnectionListener` Instanz.

Der `ConnectionHandler` erstellt zuerst einen `Input-` und einen `OutputStream` (Klassen `In` und `Out` im Verzeichnis `utils`), lauscht dann und bearbeitet die verschiedenen Meldungen von seinem Klienten. Wenn eine Nachricht ankommt, die auf dem „Anschlagsbrett“ aller aktiven Klienten angezeigt werden soll, wird sie vom `ConnectionHandler` zwischengelagert, bis sie vom `ConnectionListener` abgeholt und an alle aktiven Klienten verschickt wird.

Die einzelnen `ConnectionHandler`-Threads werden in der Liste `connections` gespeichert, die ihrerseits vom `ConnectionListener` verwaltet wird. Der `ConnectionListener` durchläuft ständig diese Liste und prüft, ob irgendein `ConnectionHandler`-Thread eine neue Nachricht erhalten hat. Wenn ja, wird diese an alle aktiven Klienten des Servers gesendet. Gleichzeitig wird in jedem `ConnectionHandler`-Thread getestet, ob die Gesprächsthemenliste und /oder die Teilnehmerliste aktualisiert werden soll. Dies geschieht, wenn ein neues Thema von einem Klienten zugefügt oder entfernt wurde, oder wenn sich ein Klient neu eingeloggt oder ausgeloggt hat. In diesen Fällen wird eine entsprechende Nachricht an alle Beteiligten geschickt.

Die `main()` Methode des Client benötigt folgende Argumente: `<Name> < Host> <Port>` (und zwar in dieser Reihenfolge). Er erstellt anhand dieser Informationen den Client Socket mit den dazugehörigen `Input-` und `OutputStream` (Klassen `In` und `Out` im Verzeichnis `utils`). Danach meldet sich der Klient mit seinem Namen beim Server an. Die beigelegte ClientGUI.java vereinfacht die Interaktion zwischen den verschiedenen Klienten und dem Server.

Die Kommunikation zwischen Server und Client kann via Strings, die einfach formatiert sind, erfolgen. Am Anfang jeder Nachrichtenzeichenkette steht ein Schlüsselwort, der die Art der Meldung und deren Inhalt bezeichnet:

Vom Klienten verschickte bzw. beim Server eingehende Zeichenketten:

`"name=<Name des Klienten>"`	Der Klient meldet sich an
`"remove_name=<Name des Klienten>"`	Der Klient meldet sich ab
`"message=<Meldung>"`	Ein Klient schickt diese Meldung
`"add_topic=<Thema>"`	Der Klient schickt die Meldung, dass er dieses Thema hinzufügen möchte
`"remove_topic=<Thema>"`	Der Klient schickt die Meldung, dass er dieses Thema entfernen möchte
`"action=getMessages;topic=<Thema>"`	Der Klient verlangt die zehn letzten Meldungen zum gegebenen Thema
`"action=refreshMessages;topic=<Thema>"`	Der Klient verlangt die zehn letzten Meldungen zum gegebenen Thema

Vom Server ausgehende bzw. beim Klienten eingehende Zeichenketten:

`"message=<Meldung>"`	Der Server verschickt die Meldung eines Klienten an alle Mitglieder
`"add_topic=<Thema>"`	der Server verschickt die Meldung, dass dieses Thema hinzugefügt wurde
"remove_topic=<Thema> "	Der Server verschickt die Meldung, dass dieses Thema entfernt wurde
`"topics=<Thema1;Thema2;Thema3;….;>"`	Der Server verschickt an einen neu-eingeloggten Klienten die Liste aller aktuellen Themen
`"add_participant=<Name des Klienten>"`	Ein neuer Teilnehmer hat sich eingeloggt
`"remove_ participant =< Name des Klienten >"`	Ein Teilnehmer hat ausgeloggt
`"participants=<Name1;Name2;Name3;….;>"`	Der Server verschickt an einen neu-eingeloggten Klienten die Liste aller aktiven Teilnehmer.
`"messages=Meldung1;Meldung2;Meldung3;…;"`	Der Server schickt an den Klienten, der dies verlangt hat, die Liste der letzten zehn Meldungen zu dem gegebenen Thema

Andere Methoden der Verpackung der Information können nach Belieben getestet werden.
Z.B. anstatt eines Schlüsselwortes eine Zahl oder sogar das Ganze in einem Objekt verpacken.

Aufgabe

Sie sollen die Klassen `Server.java`, `client.java` und `ConnectionHandler.java` (für das Starten der Client-Threads im Server) implementieren.
Wie Sie die Meldungen der verschiedenen Clients auf server-Seite bearbeiten möchten, ist es Ihnen freigestellt.
Alle anderen Klassen sind im `ChatSocket.zip`-File auf den AD vorhanden.

Test

Testen Sie Ihr Programm, indem Sie zwei (bzw. mehrere) Klienten starten, die sich mit demselben Server verbinden und prüfen Sie, ob die Änderungen, die sich durch den einen Klienten vornehmen, über alle anderen Klienten sichtbar sind.

Lieferung

Geben Sie neben dem Code eine Beschreibung Ihrer Lösung ab. Daraus soll hervorgehen, wie Sie die Kommunikation Client-Server realisiert haben. Bitte beachten Sie, dass alle von Ihnen implementierten Files in einem einzigen File (zusammen mit der kurzen Beschreibung des Lösungswegs) in **pdf-Format** an carlo.nicola@fhnw.ch senden sollen.

Abgabe: 14. April 2015