Kapitel 7: Threads

Einführung

- Threads ermöglichen Nebenläufigkeit in Java
- Sie erlauben gleichzeitige oder quasi-gleichzeitige Aktivitäten in einer Anwendung

Einführung

- Betriebssysteme erlauben i.d.R. *Multitasking*, d.h. mehrere Tasks bzw. Prozesse werden parallel oder quasi-parallel ausgeführt.
- Viele Prozesse teilen sich meist wenige *Ressourcen* z.B. einen Prozessor.
- Quasi-Parallelität entsteht, indem durch *Scheduling* den Prozessen abwechselnd für die Dauer eines *Timeslots* eine Ressource zugewiesen wird.
- Durch kurze Timeslots entsteht für den Benutzer der Eindruck, dass die Prozesse gleichzeitig ausgeführt werden.

Einführung

- Threads werden manchmal als *leichtgewichtige Prozesse* bezeichnet.
- Sie existieren innerhalb von Prozessen.
- Jeder Prozess besitzt mindestens einen Thread.
- Threads eines Prozesses teilen sich Ressourcen desselbigen.

CounterThread

```
class CounterThread extends Thread {
   public void run() {
      for (int i = 0; i < 1000000; i++) {
         Counter.value++;
      }
   }
}
```

Erhöht den Wert der Variable `value` um 1.000.000

Counter

Threads starten, `run()` wird aufgerufen

Warten, bis die Threads beendet sind

Was wird hier ausgegeben? 2.000.000?

```
public class Counter {

   public static int value = 0;

   public static void main(String[] args) {
      Thread thread1 = new CounterThread();
      Thread thread2 = new CounterThread();
      thread1.start();
      thread2.start();

      try {
         thread1.join();
         thread2.join();
      }
      catch (InterruptedException e) {
         System.out.println(e);
      }

      System.out.println(value);
   }
}
```

Problemanalyse

- Der Ergebniswert liegt maximal bei 2.000.000, meist aber darunter!
- Grund: `value++` ist für den Prozessor keine atomare Operation; der Wert wird gelesen, erhöht und zurückgeschrieben

Beispielszenario

Thread 1	**Thread 2**	**value** 0

`Read`
`Increment`
`Write`
		1

	`Read`
	`Increment`
	`Write`
	`Read`
		2

`Read`
`Increment`
`Write`
		3

	`Increment` `Write`	3

Nachdem `value` je Thread zweimal erhöht wurde, ist der Wert am Ende 3, nicht 4!

CounterThread, synchronisiert

```
class CounterThread extends Thread {

   private static Object sema = new Object();

   public void run() {
      for (int i = 0; i < 1000000; i++) {
         synchronized(sema) {
            Counter.value++;
         }
      }
   }
}
```

Der synchronisierte Ablauf ist wesentlich langsamer als der unsynchronisierte.

Thread-Synchronisation

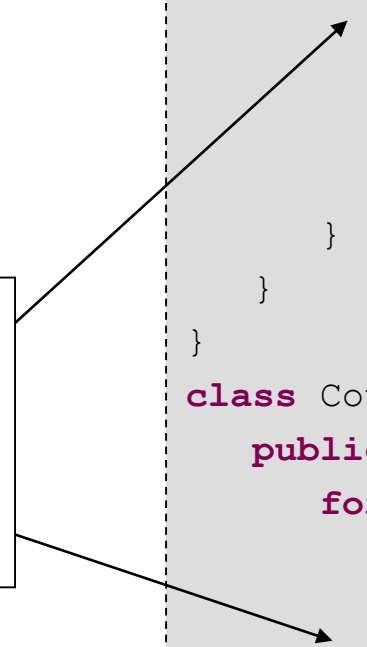
- Synchronisierte Blöcke werden exklusiv für einen Thread reserviert.
- Andere Threads werden bei versuchtem Eintreten eines bereits reservierten Blocks bis zu dessen Freigabe angehalten.
- Auch Methoden als Ganzes können mit dem Schlüsselwort `synchronized` qualifiziert werden.

Deadlocks

- Vorsicht beim Synchronisieren von Threads: Es besteht die Gefahr von Deadlocks!
- Bei einem Deadlock blockieren sich dauerhaft zwei oder mehrere Threads gegenseitig.
- Beispiel folgt…

Deadlocks

Gefahr von Deadlocks durch überkreuztes Sperren von Ressourcen



```
class CounterThread1 extends Thread {
   public void run() {
      for (int i = 0; i < 10000; i++) {
         synchronized (Counter.semaIncr) {
            Counter.value++;
            synchronized (Counter.semaDecr) {
               Counter.value--;
            }
         }
      }
   }
}
class CounterThread2 extends Thread {
   public void run() {
      for (int i = 0; i < 10000; i++) {
         synchronized (Counter.semaDecr) {
            Counter.value--;
            synchronized (Counter.semaIncr) {
               Counter.value++;
            }
         }
      }
   }
}
```

Deadlocks

Programm *hängt*, wenn ein Deadlock eintritt

```
public class Counter {
   public static int value;
   public static Object semaIncr = new Object();
   public static Object semaDecr = new Object();

   public static void main(String[] args) {
      Thread thread1 = new CounterThread1();
      Thread thread2 = new CounterThread2();
      thread1.start();
      thread2.start();
      try {
         thread1.join();
         thread2.join();
      }
      catch (InterruptedException e) {
         System.out.println(e);
      }
      System.out.println(value);
   }
}
```

Threads in Java

- Grundsätzlich gibt es zwei Möglichkeiten, um eigene Threads in Java zu implementieren:
  - Von der Klasse `Thread` ableiten
  - Das Interface `Runnable` (mit ausschließlich der Methode `run()`) implementieren und eine Instanz an ein `Thread`-Objekt übergeben

Die Klasse `Thread`

`Thread()`	Erzeugt ein `Thread`-Objekt
`Thread(Runnable r)`	Erzeugt ein `Thread`-Objekt, wobei beim Ausführen des Threads die `run()`-Methode der `Runnable`-Instanz zum Tragen kommt.
`void interrupt()`	Unterbricht den Thread
`void join()`	Wartet, bis der Thread beendet ist
`void join(` `  long millis)`	Wartet, bis der Thread beendet ist; maximal bis zum definierten Wert von Millisekunden
`void run()`	Diese Methode implementiert den eigentlichen Code des Threads, falls von `Thread` abgeleitet wurde. Falls dem Thread ein `Runnable` übergeben wurde, wird sie durch dessen `run()`-Methode ersetzt.
`void setPriority(` `  int priority)`	Setzt die Priorität des Threads
`static void sleep(` `  long millis)`	Legt den aktuell ausgeführten Thread für einen definierten Wert von Millisekunden *schlafen*
`void start()`	Startet den Thread; die JVM ruft daraufhin die `run()`-Methode auf
`static void` `  yield()`	Lässt den aktuell ausgeführten Thread vorübergehend pausieren, sodass andere Threads zum Zuge kommen

Beispiel Thread-Interrupt

```
public class Counter {
   public static int value = 0;
   public static void main(String[] args) {
      Thread thread = new CounterThread();
      thread.start();
      try {
         Thread.sleep(1000);
         thread.interrupt();
         thread.join();
      }
      catch (InterruptedException e) {
         System.out.println(e);
      }
      System.out.println("Endwert: " + value);
   }
}
```

Warten bis der Thread abgebrochen ist

Beispiel Thread-Interrupt

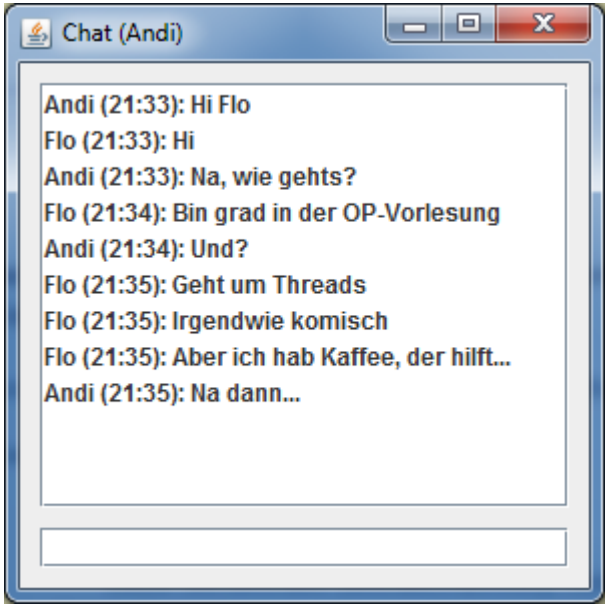
```
class CounterThread extends Thread {
   public void run() {
      try {
         while (true) {
            Counter.value++;
            if (Counter.value % 1000 == 0) {
               System.out.println(Counter.value);
               Thread.sleep(100);
            }
         }
      }
      catch (InterruptedException e) {
         System.out.println(e);
      }
   }
}
```

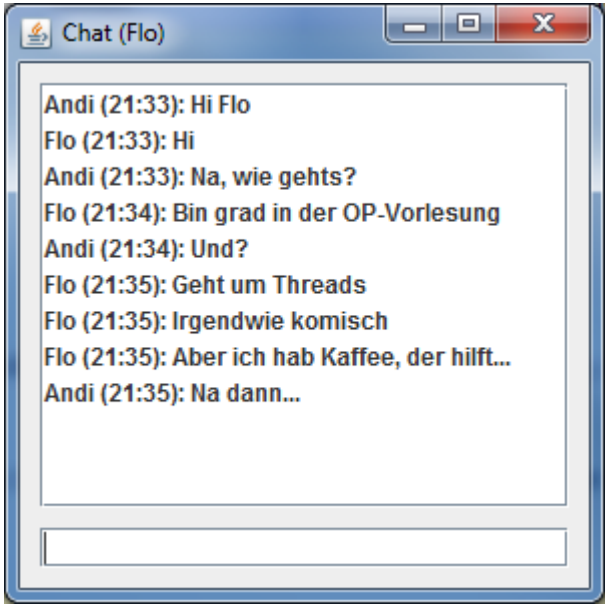
Führt bei `interrupt()` zur Exception

Das Chat-Projekt

- Ziele:
  - ChatServer und ChatClient entwickeln
  - Beliebig viele ChatClients sind möglich
  - Kommunikation im Netzwerk
  - ChatClient bietet GUI mit Swing
  - Komponenten sind stabil, d.h. ein Absturz einzelner Komponenten erfordert keinen Neustart aller Komponenten

Vorschau

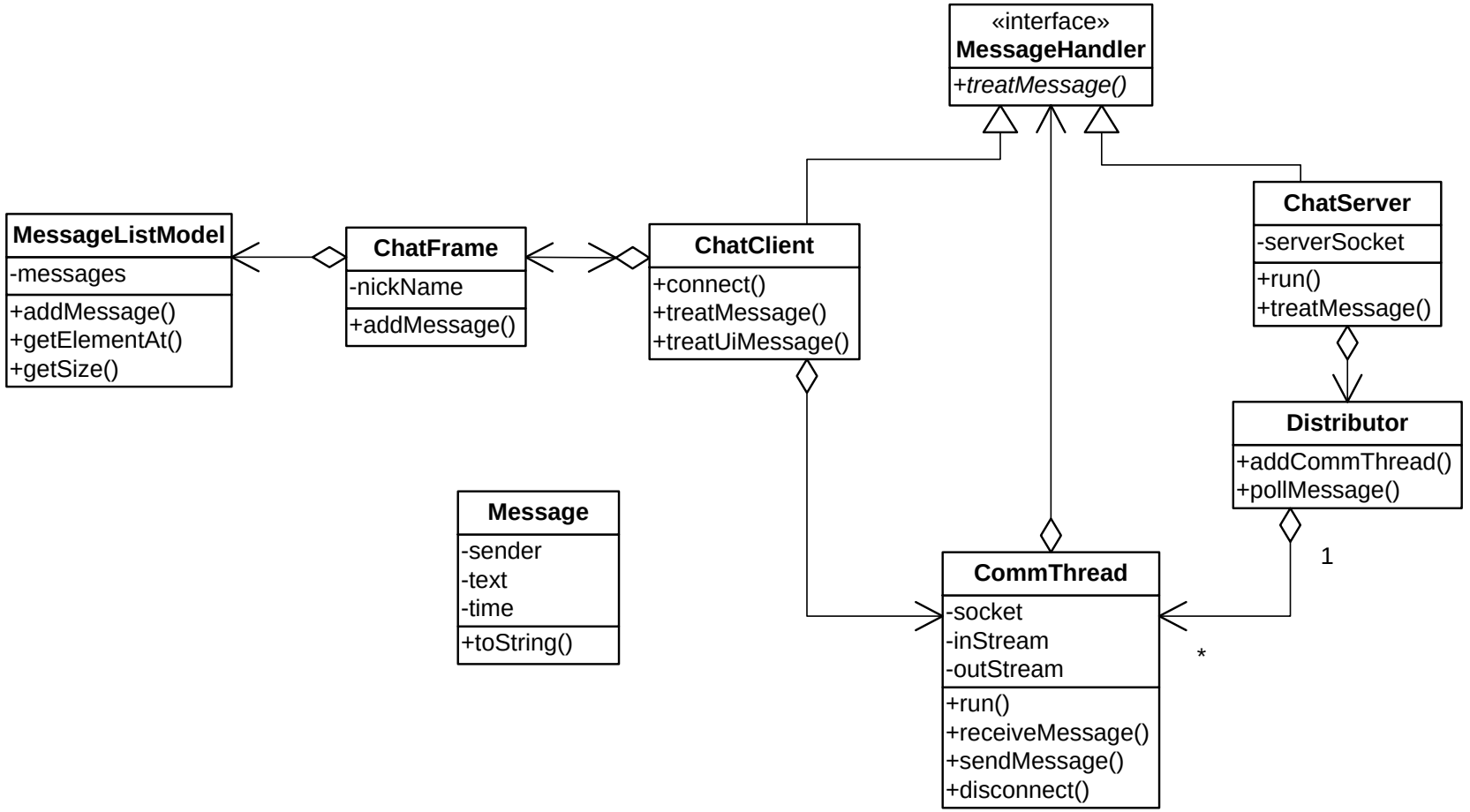

Chat (Andi)
Andi (21:33): Hi Flo
Flo (21:33): Hi
Andi (21:33): Na, wie gehts?
Flo (21:34): Bin grad in der OP-Vorlesung
Andi (21:34): Und?
Flo (21:35): Geht um Threads
Flo (21:35): Irgendwie komisch
Flo (21:35): Aber ich hab Kaffee, der hilft...
Andi (21:35): Na dann...


Chat (Flo)
Andi (21:33): Hi Flo
Flo (21:33): Hi
Andi (21:33): Na, wie gehts?
Flo (21:34): Bin grad in der OP-Vorlesung
Andi (21:34): Und?
Flo (21:35): Geht um Threads
Flo (21:35): Irgendwie komisch
Flo (21:35): Aber ich hab Kaffee, der hilft...
Andi (21:35): Na dann...

Ideen zur Umsetzung

- Wir teilen das Projekt in drei Packages:
  - `chat.client`: Die Client-Anwendung
  - `chat.server`: Die Server-Anwendung
  - `chat.comm`: Gemeinsame Klassen für die Kommunikation zwischen Client und Server
- Zur Kommunikation werden Objekte vom Typ `Message` übertragen.

Klassendiagramm



Die Klasse `Message`

```
package chat.comm; …
public class Message implements Serializable {
   private String sender;
   private String text;
   private Date time;

   public Message(String sender, String text) {
      this.sender = sender;
      this.text = text;
      time = new Date();
   }
   public String getSender() { return sender; }
   public String getText() { return text; }
   public Date getTime() { return time; }
   public String toString() {
      String timestr = DateFormat.getTimeInstance(DateFormat.SHORT).format(time);
      return sender + " (" + timestr + "): " + text;
   }
}
```

Zur Übertragung im Netz muss sie `Serializable` implementieren!

Aktuelle Systemzeit (mit Datum)

Formatierung in `HH:mm`

Zur Ausgabe der Nachricht

Sockets

- Problem: Wie schaffen wir die Kommunikation zwischen Client und Server?
- Lösung: Sockets!

Sockets

- Sockets erlauben eine Client-Server-Verbindung.
- Der Server bietet Dienste an, welche der Client in Anspruch nimmt.
- Sockets sind die Endpunkte der Kommunikation.
- Für eine Socket-Verbindung wird in Java die Klasse `Socket` verwendet.
- Um eine Verbindung zum Server herzustellen, benötigt der Client eine IP-Adresse und einen Port.
- Serverseitig erlaubt die Klasse `ServerSocket`, neue Socket-Verbindungen an einem bestimmten Port anzunehmen (`accept()`)
- Zum Übertragen von Daten über Sockets dienen Streams (`getInputStream()`, `getOutputStream()`)

Organisation der C/S-Kommunikation

- Je ein Thread auf dem Client und auf dem Server kommunizieren per Socket.
- Dies erledigt die Klasse `chat.comm.CommThread`.
- Der Thread wartet auf eingehende Nachrichten und delegiert die Verarbeitung an eine Instanz, die das Interface `MessageHandler` implementiert (Stichwort: *Dependency Injection*)
- `CommThread` erlaubt zudem das Versenden von Nachrichten.
- Scheitert die Kommunikation, wird der Thread beendet.
- Übertragene Objekte im falschen Format (nicht vom Typ `Message`) werden ignoriert.

Das Interface `MessageHandler`

Die implementierende Klasse ist verantwortlich, eingehende Nachrichten zu verarbeiten.

```
package chat.comm;

public interface MessageHandler {

   void treatMessage(Message message);
}
```

Die Klasse `CommThread`

```
package chat.comm; …
public class CommThread extends Thread {
   private Socket socket;
   private ObjectInputStream inStream;
   private ObjectOutputStream outStream;
   private MessageHandler messageHandler;

   public CommThread(Socket socket, MessageHandler mh) throws IOException { … }
   public void run() { … }
   private Message receiveMessage() throws IOException { … }
   public void sendMessage(Message message) throws IOException { … }
   private void disconnect() { … }
   …
}
```

Die Klasse `CommThread`

```
package chat.comm; …
public class CommThread extends Thread {
   private Socket socket;
   private ObjectInputStream inStream;
   private ObjectOutputStream outStream;
   private MessageHandler messageHandler;

   public CommThread(Socket socket, MessageHandler mh) throws IOException {
      this.socket = socket;
      messageHandler = mh;
      outStream = new ObjectOutputStream(socket.getOutputStream());
      outStream.flush();
      inStream = new ObjectInputStream(socket.getInputStream());
   }
   …
}
```

Ein- und Ausgabestrom für Nachrichten

Die Klasse `CommThread`

```
public class CommThread extends Thread {
    …
    public void run() {

        try {
            while (true) {
                Message message = receiveMessage();
                messageHandler.treatMessage(message);
            }
        }
        catch (Exception e) {}
    }
    …
}
```

Warten auf eingehende Nachricht

Delegieren an den `MessageHandler`

Die Klasse `CommThread`

```
public class CommThread extends Thread { …
   private Message receiveMessage() throws IOException {
      Object message = null;
      do {
         try {
            message = inStream.readObject();
         }
         catch (IOException ioe) {
            disconnect();
            throw ioe;
         }
         catch (ClassNotFoundException cne) {}
      }
      while (! (message instanceof Message));
      return (Message)message;
   } …
}
```

Warten auf eingehende Nachricht

`disconnect()` bei `IOException`

Die Klasse `CommThread`

```
public class CommThread extends Thread { …
   public void sendMessage(Message message) throws IOException {
      try {
         outStream.writeObject(message);
         outStream.flush();
      }
      catch(IOException e) {
         disconnect();
         throw e;
      }
   } …
}
```

Ausgabe wird nicht gepuffert, sondern umgehend versendet.

Die Klasse `CommThread`

```
public class CommThread extends Thread { …
   private void disconnect() {
      try { inStream.close(); }
      catch(IOException ex) {}

      try { outStream.close(); }
      catch(IOException ex) {}

      try { socket.close(); }
      catch(IOException ex) {}
   } …
}
```

Der ChatServer

- Die Klasse `ChatServer` implementiert den eigentlichen Server.
- Er nimmt neue Verbindungen entgegen, verwaltet also den `ServerSocket`.
- Pro Socketverbindung erzeugt er einen `CommThread`.
- `ChatServer` implementiert den `MessageHandler`; Eingehende Nachrichten sollen an alle Clients weiterverteilt werden. Diese Aufgabe delegiert er an die Klasse `Distributor`.

Die Klasse `Distributor`

```
package chat.server; …
public class Distributor {
   private List<CommThread> commThreads;
   public Distributor() {
      commThreads = new ArrayList<CommThread>();
   }
   public void addCommThread(CommThread ct) {
      commThreads.add(ct);
   }
   public void pollMessage(Message message) {
      Iterator<CommThread> it = commThreads.iterator();
      while (it.hasNext()) {
         CommThread ct = it.next();
         try { ct.sendMessage(message); }
         catch (IOException e) { it.remove(); }
      }
   }
}
```

Alle `CommThread`s für die Clientverbindungen

Statt der erweiterten for-Schleife nutzen wir den `Iterator` explizit, damit im Fehlerfall der Client per `remove()` entfernt werden kann.

Die Klasse `ChatServer`

```
package chat.server; …
public class ChatServer implements MessageHandler {
   private Object sema = new Object();
   private ServerSocket serverSocket;
   private Distributor distributor;
   public ChatServer() throws IOException {
      serverSocket = new ServerSocket(CommRes.SERVER_PORT);
      distributor = new Distributor();
   }
   public void run() throws IOException {
      while (true) {
         Socket socket = serverSocket.accept();
         CommThread ct = new CommThread(socket, this);
         ct.start();
         synchronized(sema) { distributor.addCommThread(ct); }
      }
   }
   public void treatMessage(Message message) {
      synchronized(sema) { distributor.pollMessage(message); }
   } …
}
```

Warten auf neue Socket-Verbindung

Läuft in verschiedenen Threads!

Die Klasse `ChatServer`

- Es fehlen noch `CommRes`:

```
package chat.comm;
public interface CommRes {
   static final int SERVER_PORT = 98;
}
```

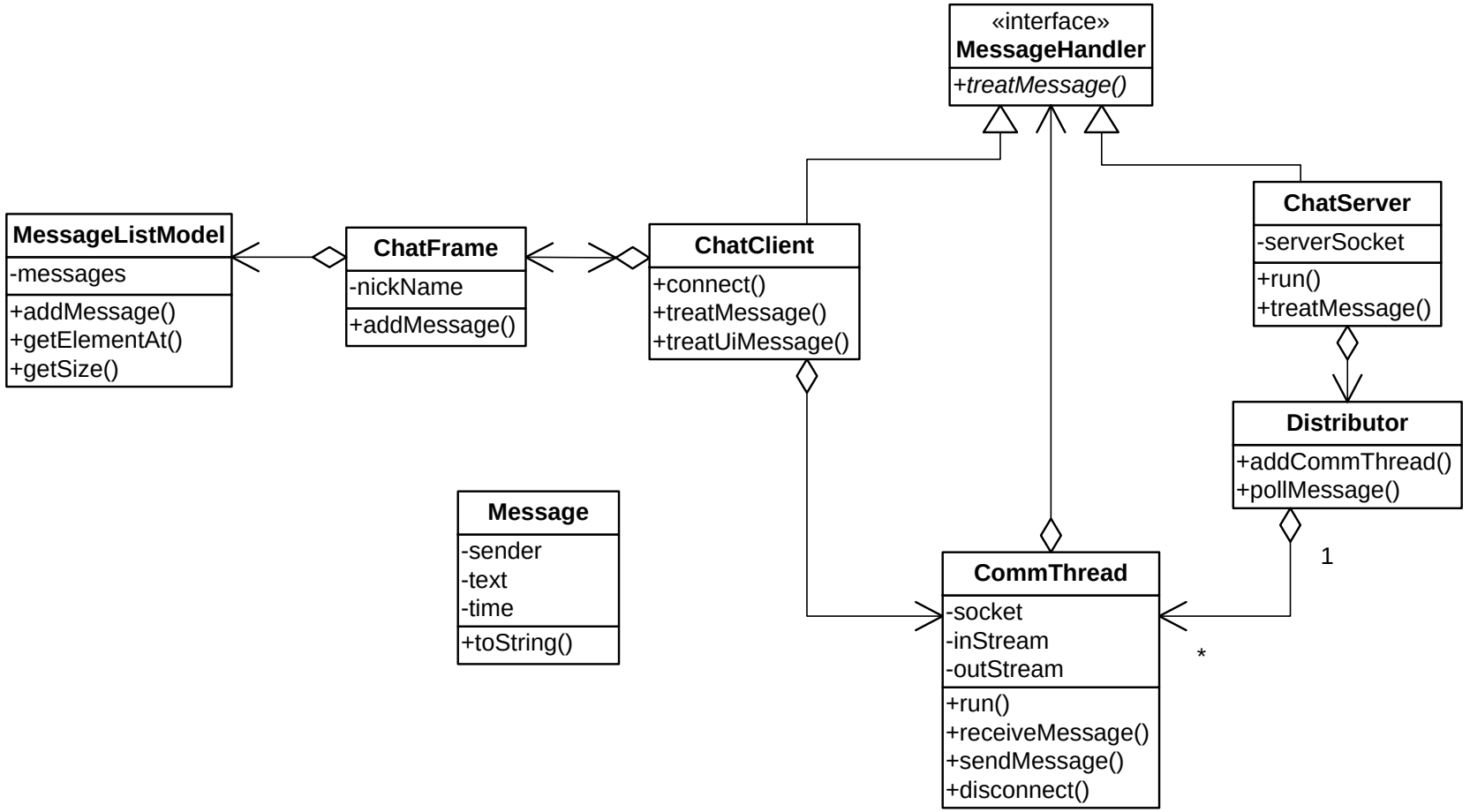
- Und natürlich `main()`:

```
public class ChatServer implements MessageHandler { …
   public static void main(String args[]) throws IOException {
      ChatServer server = new ChatServer();
      server.run();
   }
}
```

Der ChatClient

- Die Klasse `ChatClient` ist die Hauptklasse des Clients.
- `ChatClient` verwendet die bekannte Klasse `CommThread` für die Kommunikation zum Server.
- Die GUI implementiert `ChatFrame` für das eigentliche Chatfenster, `JOptionPane` wird anfangs zum Abfragen des Nicknames verwendet.
- `ChatClient` implementiert `MessageHandler` und übergibt eingehende Nachrichten an `ChatFrame`.
- `ChatFrame` übergibt zu versendende Nachrichten an `ChatClient`, der sie wiederum an `CommThread` zum Versenden weiterreicht.

Klassendiagramm



Die Klasse `ChatClient`

```
package chat.client; …
public class ChatClient implements MessageHandler {
   private static final String SERVER_HOST = "localhost";
   private CommThread commThread;
   private ChatFrame chatFrame;

   public ChatClient() { … }
   private void connect() throws IOException { … }
   public void treatMessage(Message message) { … }
   public void treatUiMessage(Message message) { … }

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

Die Klasse `ChatClient`

```
public class ChatClient implements MessageHandler { …
   public ChatClient() {
      chatFrame = new ChatFrame(this);
      try { connect(); }
      catch (IOException ex) {
         Message errormsg = new Message("System", "Server nicht erreichbar.");
         chatFrame.addMessage(errormsg);
      }
      chatFrame.setVisible(true);
   }
   private void connect() throws IOException {
      Socket socket = new Socket(SERVER_HOST, CommRes.SERVER_PORT);
      commThread = new CommThread(socket, this);
      commThread.start();
   } …
}
```

Erst nach dem Verbindungsversuch setzen

Die Klasse `ChatClient`

```
public class ChatClient implements MessageHandler { …
   public void treatMessage(Message message) {
      chatFrame.addMessage(message);
   }
   public void treatUiMessage(Message message) {
      try {
         if (commThread == null) connect();
         commThread.sendMessage(message);
      }
      catch (IOException ex) {
         commThread = null;
         Message errormsg = new Message("System", "Server nicht erreichbar.");
         chatFrame.addMessage(errormsg);
      }
   } …
}
```

(Eingehende) Nachrichten vom `CommThread`

(Ausgehende) Nachrichten vom `ChatFrame`

Bei Verbindungsfehlern

Verarbeitung eingehender Nachrichten

- …Die Aufgabe von `treatMessage()`:

```
public class ChatClient implements MessageHandler { …
   public void treatMessage(Message message) {
      chatFrame.addMessage(message);
   } …
}
```

- Wir wollen `JList` einen Eintrag hinzufügen.
- Der `CommThread` muss also mit Swing kommunizieren.
- Ein Problem? Bedarf zur Thread-Synchronisierung?

Threads und Swing

- Swing ist bis auf wenige Methoden **nicht** *thread-safe*, d.h. nur diese wenigen Methoden können von jedem Thread aus problemlos aufgerufen werden.
- Ansonsten laufen Swing-Operationen im sog. *event-dispatching thread*

Threads und Swing

- Unsere bisherige Praxis im Umgang mit Swing:
  - Wir haben im *initial thread* (der Hauptthread, der `main()` aufruft) vor dem Aufruf von `setVisible()` die GUI initialisiert.
  - Alles Weitere fand (wie von Swing gefordert) im *event-dispatching thread* statt.

- Beispielsweise werden die Implementierungen von `ActionListener` im *event-dispatching thread* ausgeführt.
- Mit `SwingUtilities.`*`isEventDispatchThread`*`()` kann dies überprüft werden.

Threads und Swing

- Nicht-*thread-safe* Eingriffe außerhalb des *event-dispatching threads* führen meist zum gewünschten Verhalten.
- **Allerdings gilt das nicht immer!**
- Derartige Fehlverhalten ist schwer reproduzierbar.

Threads und Swing

- Unsere neue Situation:
  - Wir haben einen sog. *worker thread*.
  - Dieser empfängt Nachrichten vom `ChatServer`.
  - Die Nachrichten müssen durch Swing ausgegeben werden.

Threads und Swing

- Eine Lösung: der *worker thread* übergibt die notwendigen Swing-Operationen an den *event-dispatching thread* und dieser führt sie dann aus.
- Swing bietet dazu folgende Methoden an:
  - `SwingUtilities.`*`invokeAndWait`*`(Runnable r)`
    `SwingUtilities.`*`invokeLater`*`(Runnable r)`

Threads und Swing

- `SwingUtilities.`*`invokeAndWait`*`(Runnable r)`
  - Erwirkt die synchrone Ausführung von `Runnable.run()` im *event-dispatching thread*
  - Der aufrufende Thread wartet bis zur Abarbeitung der Methode.
- `SwingUtilities.`*`invokeLater`*`(Runnable r)`
  - Erwirkt die asynchrone Ausführung von `Runnable.run()` im *event-dispatching thread*
  - Der aufrufende Thread wartet nicht bis zur Abarbeitung der Methode.
- I.d.R. ist die Methode `invokeLater()` hinreichend.

Threads und Swing

- Weitere Details zum Thema siehe *Concurrency in Swing*:

  http://docs.oracle.com/javase/tutorial/uiswing/concurrency/index.html

Threads und Swing

- So funktioniert es richtig:

```
public class ChatFrame extends JFrame {
   class MessageAdder implements Runnable {
      private Message message;
      public MessageAdder(Message message) { this.message = message; }
      public void run() {
         messageListModel.addMessage(message);
         lstMessages.ensureIndexIsVisible(messageListModel.getSize() - 1);
      }
   }
   …
   public void addMessage(Message message) {
      MessageAdder ma = new MessageAdder(message);
      SwingUtilities.invokeLater(ma);
   }
}
```

Nach unten scrollen…

Threads und Swing

- So funktioniert es **nicht** richtig (manchmal aber schon):

```
public class ChatFrame extends JFrame {
    …
    public void addMessage(Message message) {
        messageListModel.addMessage(message);
        lstMessages.ensureIndexIsVisible(messageListModel.getSize() - 1);
    }
}
```

- Die falsche Implementierung ist daran erkennbar, dass oft nur bis zum vorletzten Eintrag hinuntergescrollt wird.

Die Klasse `ChatClient`

Der Rest der Klasse in Auszügen

```
public class ChatFrame extends JFrame { …
   public ChatFrame(ChatClient chatclient) {
      chatClient = chatclient; …
      nickName = JOptionPane.showInputDialog(this, "Bitte Nickname eingeben:",
                                    "Anmeldung", JOptionPane.QUESTION_MESSAGE);
      if (nickName == null || nickName.trim().equals("")) nickName = "Anonym"; …
      messageListModel = new MessageListModel();
      lstMessages = new JList(messageListModel); …
      txtMessage = new JTextField(); …
      txtMessage.addActionListener(new ActionListener() {
         public void actionPerformed(ActionEvent e) {
            if (txtMessage.getText().equals("")) return;
            Message message = new Message(nickName, txtMessage.getText());
            chatClient.treatUiMessage(message);
            txtMessage.setText("");
         }
      });
      txtMessage.requestFocusInWindow();
   } …
}
```

Kein `setVisible()`!

Die Klasse `MessageListModel`

```
package chat.client; …
public class MessageListModel extends AbstractListModel {
   private List<Message> messages;
   public MessageListModel() {
      messages = new ArrayList<Message>();
   }
   public void addMessage(Message message) {
      messages.add(message);
      fireIntervalAdded(this, messages.size() - 1, messages.size() - 1);
   }
   public Object getElementAt(int index) {
      return messages.get(index);
   }
   public int getSize() {
      return messages.size();
   }
}
```


Nicht vergessen:
View über Änderungen Bescheid geben!

Stabilität des Chats

- Was passiert bei „ungewöhnlichen“ Ereignissen?
- Erhalten wir stets ein kontrolliertes Verhalten?
- Insbesondere zu untersuchende Felder:
  - Client-Server-Kommunikation
  - Kommunikation zwischen Threads
  - Sonstige Ausnahmen
- Kann die Stabilität bewiesen werden? Nein.

Client-Server-Kommunikation

- Serverreaktion auf „Clientabsturz“:
  - `CommThread` beendet die Verbindung
  - `Distributor` entfernt den Client aus der Verteilerliste
- Clientreaktion auf „Serverabsturz“:
  - `CommThread` beendet die Verbindung
  - Bei erneutem Versuch eine Nachricht zu senden, wird versucht, vorab eine neue Serververbindung herzustellen.

Inter-Thread-Kommunikation

- Serverseitig:
  - Synchronisation des Versendens durch den Distributor an Clients und die Aufnahme neuer Clients in den Distributor.
- Clientseitig:
  - GUI wird im *initial thread* vor `setVisible()` initialisiert und danach durch diesen nicht mehr verändert.
  - Eingehende Nachrichten werden per `invokeLater()` an den *event-dispatching thread* von Swing übergeben.

Inter-Thread-Kommunikation

- Bisher unberücksichtigt: Gleichzeitiges Lesen und Schreiben über den Socket:
  - Beim Server senden und empfangen die Client-spezifischen `CommThread`s eben in selbigen Threads Nachrichten.
  - Beim Client schreibt `ChatFrame` unter in Anspruchnahme von `ChatClient` im *event-dispatching thread*, `CommThread` empfängt eben in selbigem Thread Nachrichten.
- Ist gleichzeitiges Lesen und Schreiben über Sockets *thread-safe*?
- Die Antwort aus der Java-Hilfe:

  Socket channels are safe for use by multiple concurrent threads. They support concurrent reading and writing, though at most one thread may be reading and at most one thread may be writing at any given time. …

- Also: Alles in Ordnung!
- Weitere Details siehe: https://docs.oracle.com/javase/7/docs/api/java/nio/channels/SocketChannel.html

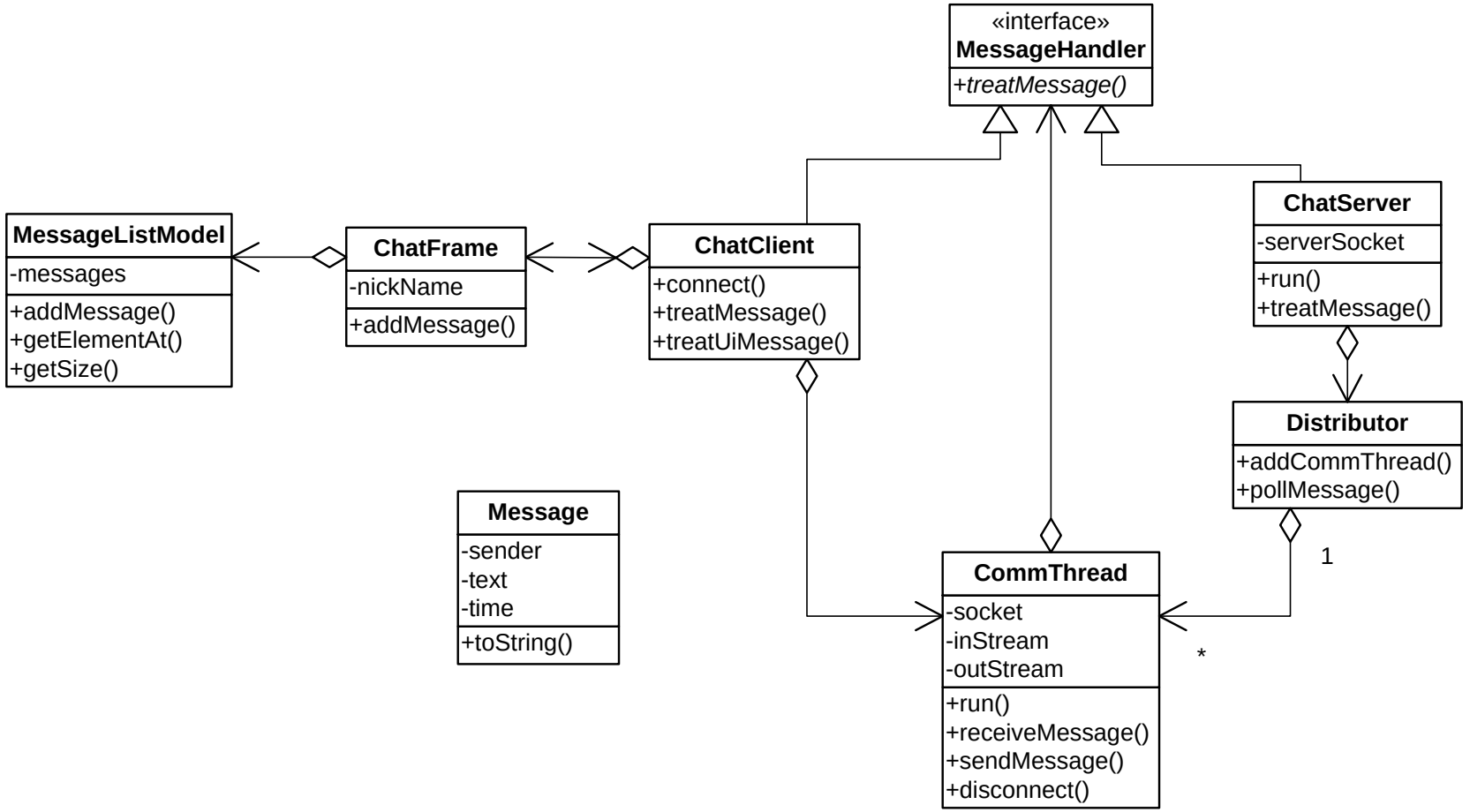
Sonstige Ausnahmen

- Der Server bricht beim Aufbau des `ServerSocket` bei einer `IOException` vollständig ab.
- ➔ OK! - was soll er sonst machen?

- Der Server bricht bei der Annahme einer neuen Clientverbindung bei einer `IOException` vollständig ab.
- ➔ Nachbesserungsbedarf! - er könnte hier lediglich den Client „rausschmeißen“.

- Was ist sonst noch zur berücksichtigen?

Klassendiagramm



Mögliche Erweiterungen

- Natürlich sollte sich ein Client im Normalfall vom Server selbst abmelden…
- Benutzerauthentifizierung
- Benutzerliste im Chatfenster
- Steuer-Messages (mittels verschiedener Message-Typen)
  - Nachricht nur an bestimmte Benutzer senden
  - Anmelden/Abmelden
  - Server herunterfahren
  - …

Aspekte des Softwareengieerings

- Wie kommt man auf diese Struktur?
  - ➔ Erfahrung, Top-Down-Ansatz, agil
- Persönliche Meinung: Eine gesunde Mischung aus Perfektionismus und Pragmatismus schafft langfristig stabile und wartbare Anwendungen.