LMU LUDWIG-MAXIMILIANS-UNIVERSITÄT MÜNCHEN

mobile and distributed systems group

Javakurs für Fortgeschrittene

Einheit 06: Observer und FXML

Lorenz Schauer
Lehrstuhl für Mobile und Verteilte Systeme

Heutige Agenda

Teil 1: Design-Pattern in JavaFX

- Modell-View-Controller (Wiederholung)
- Observer-Pattern
- Umsetzung in JavaFX

Teil 2: JavaFX mit FXML und Scene Builder

- Motivation für FXML
- FXML Überblick
- Scene Builder installieren und Demo

Praxis:

- MVC in JavaFX
- Observer-Pattern
- FXML mit Scene Builder

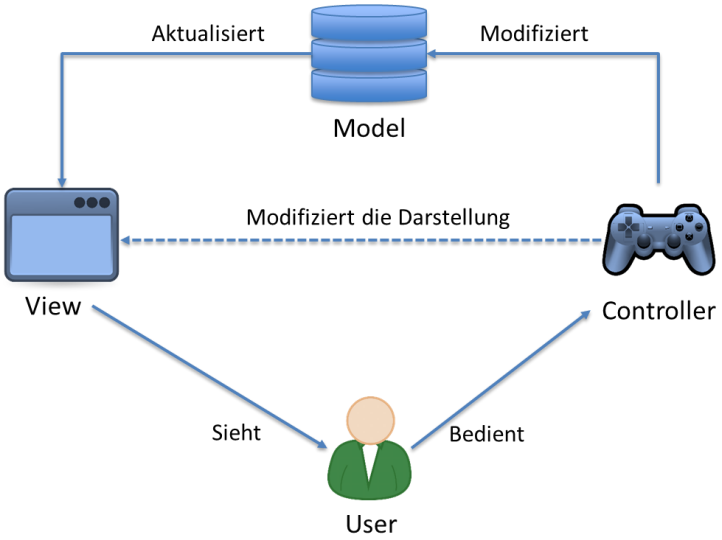
Lernziele

- MVC als Entwurfsmuster für grafische Anwendungen kennenlernen
- Das Observer Pattern zur synchronen Datenanzeige verstehen
- Scene Builder und FXML in JavaFX nutzen können

Wiederholung: MVC

Model:

- Unabhängig von View & Controller
- Aktualisierung der View über Beobachter

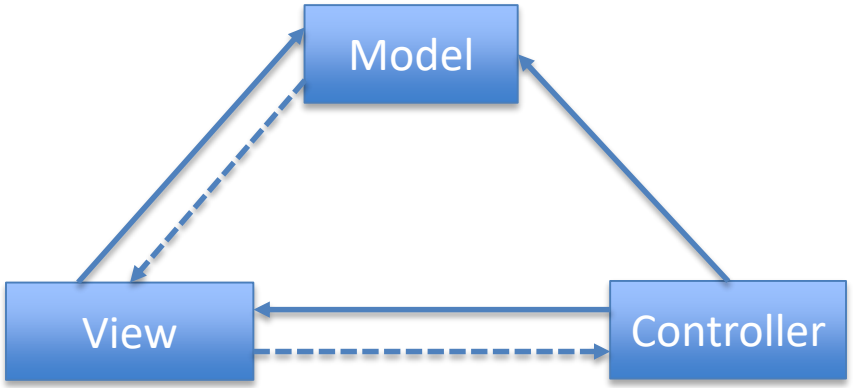


View:

- Kennt ihre Steuerung und ihr Modell
- Wird bei Datenänderung benachrichtigt
- Braucht eine eigene Steuerung

Controller:

- Verwaltet mind. eine View
- Nimmt Interaktionen entgegen
- Bewirkt Änderungen im Model bzw. der View

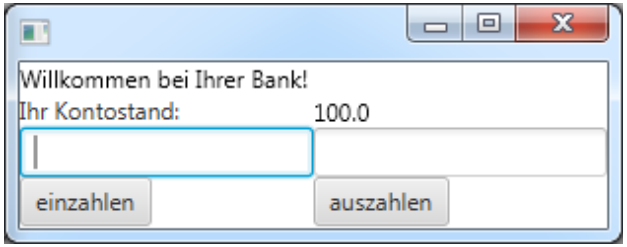


Aufgabe zu MVC (letzte Stunde)

Schreiben Sie eine kleine Anwendung unter Berücksichtigung des MVC Musters:

Erstellen Sie ein neues JavaFX Projekt „MVC“ und generieren Sie 3 Klassen:

- **Anzeige**: Erbt von `Parent` und bildet die Anzeige (und den Controller) für eine einfache Bank-Anwendung, bei welcher der Nutzer Ein- und Auszahlungen mit Hilfe eines Textfeldes und den Buttons tätigen kann.
  - Der aktuelle Kontostand soll darüber angezeigt und bei jeder Ein- bzw. Auszahlung entspr. angepasst werden.



- **Bank**: Ist das eigtl. Model und besitzt eine Instanzvariable für den Kontostand (`double`)
  - Außerdem die Methoden `einzahlen(double Betrag), auszahlen(double Betrag)`
  - Zusätzlich einen Getter für die Instanzvariable des Kontostands
  - Im Konstruktor sollte ein Initialkontostand angegeben werden können.

- **Main**: Die Main-Klasse beinhaltet die `start(Stage primaryStage)` Methode. Sie erzeugt: eine Bankinstanz und die Anzeige, welche die GUI mit dem integrierten Controller enthält

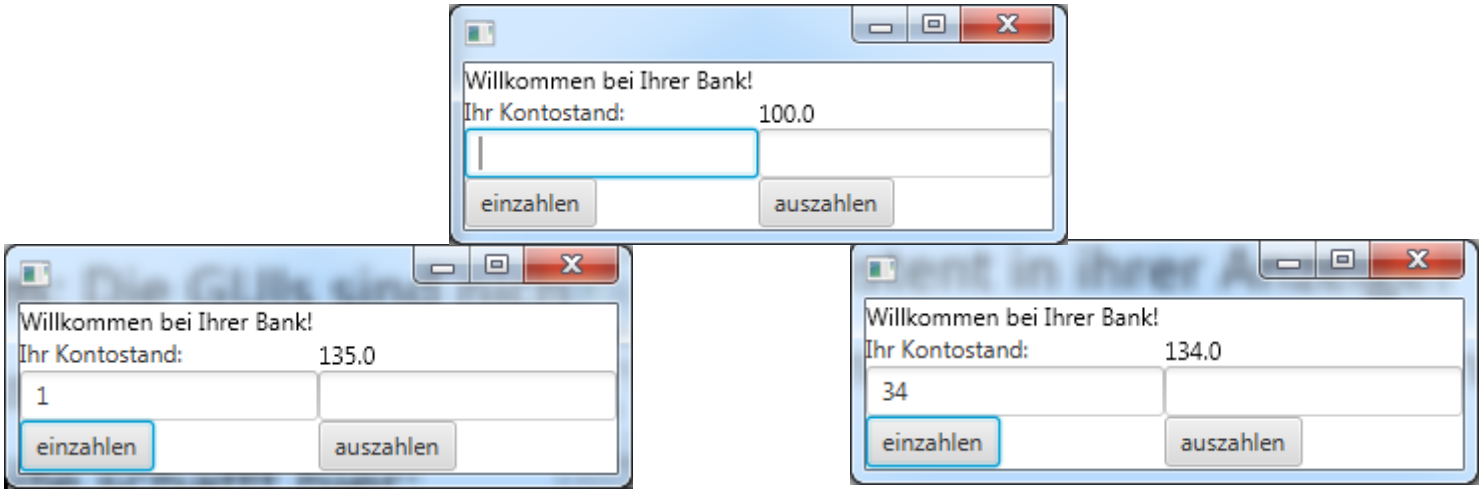
Verknüpfen Sie nun die Teile, so dass Sie über Ihre GUI Ein- und Auszahlungen tätigen können, die sich im Model und in der Anzeige auswirken.

Erweiterung auf 2 oder mehrere Anzeigen ?!?

Was passiert nun, wenn Sie 2 Anzeigen in dem vorherigen Beispiel erzeugen und in der einen Anzeige den Kontostand erhöhen?

Problem: Die GUIs sind nicht konsistent in ihrer Anzeige!

- Müssen also benachrichtigt werden sobald eine Änderung in den Daten erfolgt!
- Abhilfe schafft hier:
  - das Observer Pattern
  - Oder auch Properties (Kommt nächste Stunde)



Das Observer-Pattern

Ein weiteres Entwurfsmuster (Verhaltensmuster), das die Weitergabe von (Daten-)Änderungen an abhängige Objekte strukturiert.

Besteht im Wesentlichen aus 2 Akteure:

- Dem **Subjekt** (Observable):
  - Ist für die Daten zuständig, die beobachtet werden, also bei uns das Model
  - Hat Möglichkeit Beobachter zu Verwalten (hinzufügen, entfernen)
  - Kann die Beobachter über (Daten-)Änderungen informieren
    - Wie ein Push-Notification: `notifyObservers()`
- Dem **Beobachter** (Observer):
  - Implementiert die Aktualisierung, wenn eine Änderung vom Subjekt mitgeteilt wird

Vorteil: Extrem flexibel

- Subjekt und Beobachter lose gekoppelt

Nachteil: Folgen sind nicht immer abzusehen

- Hohe Änderungskosten bei vielen Beobachtern
- Keine Information über das „was“ sich geändert hat

Das Observer Pattern: Konkrete Implementierungsschritte


```
// Subjekt: Unser Model erbt von der Klasse Observable
public class MyModel extends Observable{

   // Wenn sich was ändert, dann Bescheid geben!
   public void aendere_etwas(){
      doSomethigThatChangeSomething();
      setChanged();
      notifyObservers();
   }
}
```

```
// Beobachter: Unsere GUI als Beobachter implementiert das IFace Observer

public class MyGUI implements Observer{

   // Muss die einzige Methode update() implementieren!
   @Override
   public void update(Observable arg0, Object arg1) {

      // Hier kommt die Aktualisierungsstrategie, die ausgeführt wird,
      // wenn sich was ändert  . Bsp.:
      meinTextFeld.setText(""+myModel.getState());

   }
}
```

Damit die Beobachter auch vom Subjekt informiert werden, müssen sie zur Liste der Beobachter hinzugefügt werden:

- `myModel.add(myGUI);`

Aufgabe zu Observer-Pattern

Verwirklichen Sie nun die Prinzipien des Observer-Pattern in Ihrer zuvor erstellten Aufgabe der Bank-Anwendung:

- Lassen Sie zunächst von der start-Methode 2 oder mehrere Ihrer Anzeigen generieren, die alle die selbe Instanz des Models übergeben bekommen.
- Machen Sie nun ihr Model zu einem Observable (mittels Vererbung)
- und die Anzeige zu einem Observer, in dem Sie das entsprechende Interface implementieren.
  - Hier muss die `update()` Methode so gestaltet werden, dass sich der Kontostand automatisch aktualisiert
- Vergessen Sie nicht die Observer bei Ihrem Subjekt anzumelden, damit der Mechanismus funktioniert.

- Testen Sie Ihre Implementierung des Observer-Patterns.
  Der Kontostand sollte sich nun immer auf allen Anzeigen gleichzeitig ändern!

Teil 2: JavaFX mit FXML und Scene Builder

Motivation: FXML

JavaFX bereits einfacher zu implementieren als Swing:

- Einfachere UI-Elemente
- Trennung vom Layout durch Laden von externen CSS-Files

Dennoch:

- GUI Programmierung immer noch aufwendig
- Und eigentlich trivial
  - Folgt immer dem selben Muster

Daher:

- Einführung eines XML-Standards: FXML
  - Formale Beschreibung von Layout und UI-Elementen
  - Können dann als Objekte geladen und im Code verwendet werden
- Vorteil: Einfache Erzeugung durch WYSIWYG-Tools
  - Hier bspw.: Scene Builder

```
<?xml version="1.0" encoding="UTF-8"?>

<?import javafx.geometry.*?>
<?import javafx.scene.text.*?>
<?import javafx.scene.control.*?>
<?import java.lang.*?>
<?import javafx.scene.layout.*?>
<?import javafx.scene.layout.AnchorPane?>

<BorderPane maxHeight="-Infinity" maxWidth="-Infinity" minHeight="-Infinity" minWidth="-Infinity" prefHeight="400.0" prefWidth="600.0" xml
   <center>
      <GridPane prefHeight="338.0" prefWidth="600.0" BorderPane.alignment="CENTER">
        <columnConstraints>
          <ColumnConstraints hgrow="SOMETIMES" maxWidth="294.0" minWidth="10.0" prefWidth="146.0" />
          <ColumnConstraints hgrow="SOMETIMES" maxWidth="532.0" minWidth="10.0" prefWidth="454.0" />
        </columnConstraints>
        <rowConstraints>
          <RowConstraints maxHeight="128.0" minHeight="0.0" prefHeight="12.0" vgrow="SOMETIMES" />
          <RowConstraints maxHeight="319.0" minHeight="10.0" prefHeight="307.0" vgrow="SOMETIMES" />
          <RowConstraints minHeight="10.0" prefHeight="30.0" vgrow="SOMETIMES" />
        </rowConstraints>
         <children>
            <Label text="Counter" />
            <Text fx:id="actionTarget" strokeType="OUTSIDE" strokeWidth="0.0" text="0.0" GridPane.columnIndex="1" />
            <HBox alignment="TOP_CENTER" prefHeight="100.0" prefWidth="200.0" spacing="10.0" GridPane.rowIndex="1">
               <children>
                  <Button mnemonicParsing="false" onAction="#handleIncreaseButton" text="Increase" />
                  <Button mnemonicParsing="false" onAction="#handleDecreaseButton" prefHeight="25.0" prefWidth="66.0" text="Decrease" />
               </children>
               <opaqueInsets>
                  <Insets top="4.0" />
               </opaqueInsets>
            </HBox>
         </children>
      </GridPane>
   </center>
</BorderPane>
```

Beispiel für eine FXML-Datei


```
<?xml version="1.0" encoding="UTF-8"?>

<?import javafx.scene.image.*?>
<?import java.lang.*?>
<?import javafx.scene.control.*?>
<?import javafx.scene.layout.*?>
<?import javafx.scene.layout.AnchorPane?>


<AnchorPane prefHeight="219.0" prefWidth="213.0"
xmlns:fx="http://javafx.com/fxml/1" xmlns="http://javafx.com/javafx/8">
   <children>
      <TitledPane animated="false" layoutX="-5.0" prefHeight="247.0" prefWidth="221.0" text="How is
         it?">
        <content>
          <AnchorPane minHeight="0.0" minWidth="0.0" prefHeight="215.0" prefWidth="200.0">
              <children>
                 <VBox layoutX="53.0" layoutY="11.0" prefHeight="200.0" prefWidth="100.0">
                    <children>
                       <ImageView fitHeight="175.0" fitWidth="83.0" pickOnBounds="true"
                         preserveRatio="true">
                           <image>
                              <Image url="@../../../../Pictures/2000px-Wave.svg.png" />
                           </image>
                       </ImageView>
                       <TextField text="How is it" />
                    </children>
                 </VBox>
              </children>
          </AnchorPane>
        </content>
      </TitledPane>
   </children>
</AnchorPane>
```

„Normaler“ XML-Header

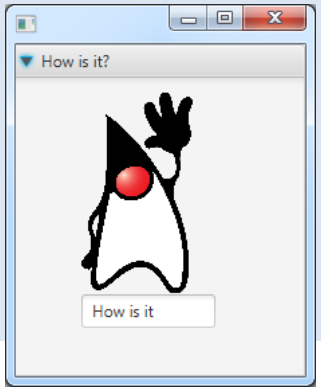
Imports: Wichtig für das Laden der Objekte

Root Node

Definition des XML-Namespaces für fxml und javafx

Kind Knoten

Klassische XML Baumstruktur: Tags schließen!



FXML Datei erzeugen: Scene Builder

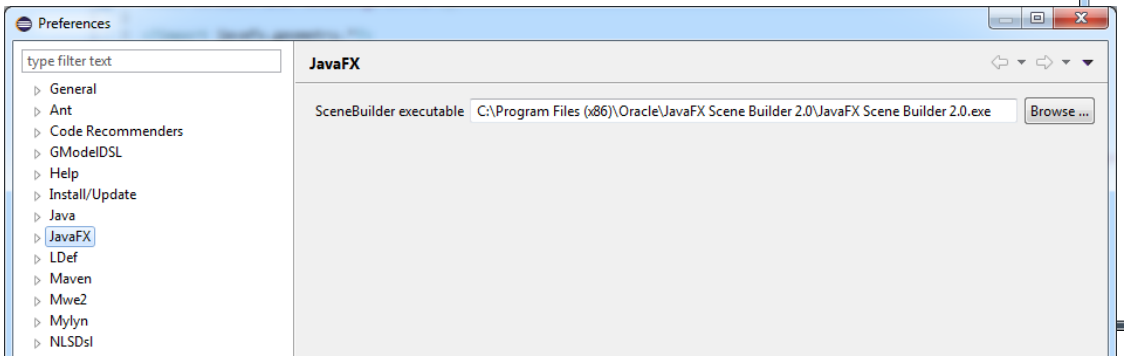
I.d.R. schreibt man eine solche FXML-Datei nicht selbst!

- Wäre sehr umständlich
- Sondern, man lässt sie erzeugen mittels WYSIWYG-Tool: **Scene Builder**

Download bei Oracle unter:

http://www.oracle.com/technetwork/java/javafxscenebuilder-1x-archive-2199384.html#javafx-scenebuilder-2.0-oth-JPR

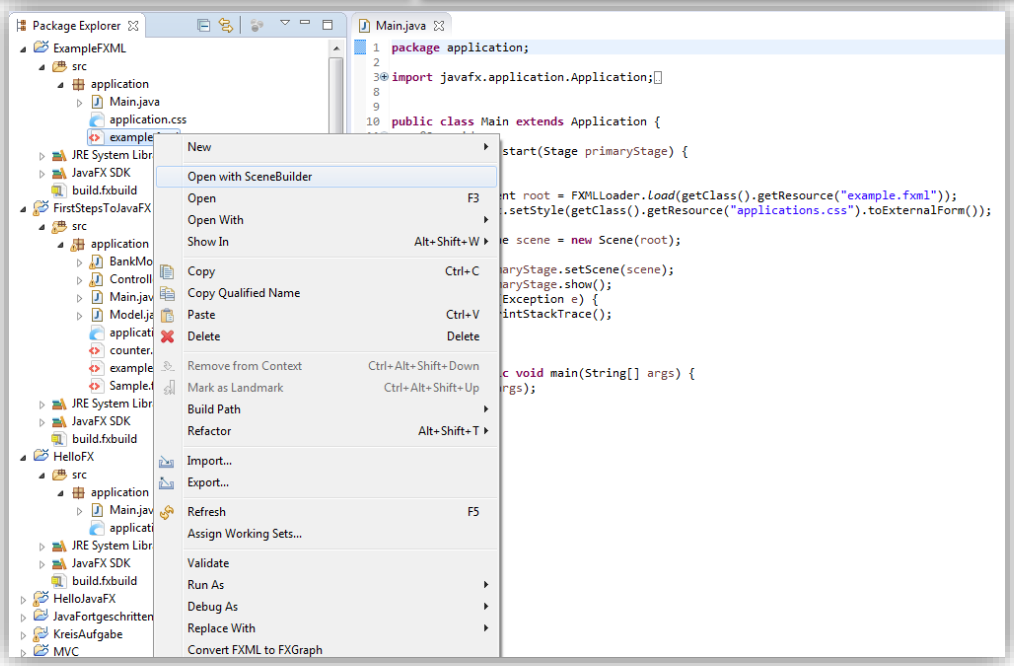
- Installieren und einbinden bei Eclipse:
  - *Window->Preferences->JavaFX:*



Scene Builder: Demo

Neues JavaFX Projekt in Eclipse:

- Wir ändern den vorgeschlagenen Code um die Root Pane
  - FXML-Datei „example.fxml“
- Neue FXML Datei mit entspr. Namen anlegen
  - Mit Scene-Builder gestalten




```
// Im Code FXML-Datei laden:
Parent root = FXMLLoader.load(getClass().getResource("example.fxml"));

// Szene setzen und CSS Datei anbinden:
Scene scene = new Scene(root);
scene.getStylesheets().add(getClass().getResource("application.css").toExternalForm());

// Szene auf die Bühne holen und anzeigen:
primaryStage.setScene(scene);
primaryStage.show();
```

Einen Controller bauen

Jede View (FXML-File) braucht einen Controller

- Den können wir durch eine separate Klasse spezifizieren:
  - Sollte das Interface `Initializable` implementieren
    - Hat die Methode `void initialize(URL location, ResourceBundle resources)`
    - Wird aufgerufen, wenn der Inhalt des FXML-Dokuments komplett geladen wurde
    - Zum Vergleich: Der Konstruktor wird zuerst aufgerufen (vor dem Laden der UI Elemente)

```
// Beispiel: Leerer Controller

public class Controller implements Initializable{

   public Controller(){

      System.out.println("Klasse instanziiert");
   }


   @Override
   public void initialize(URL location, ResourceBundle resources) {

      System.out.println(“Elemente Geladen");

   }
}
```

Ausgabe:
```
Klasse instanziiert
Elemente Geladen
```

Den Controller hinzufügen

Den Controller können wir nun sehr einfach einbinden und in Ihm die Nutzereingaben entgegennehmen:

```
… xmlns:fx="http://javafx.com/fxml/1" fx:controller="application.Controller">
```

- Verbindung zwischen Controller und UI Elementen über IDs und FXML Annotations: @FXML

```
@FXML private Text actionTarget;
```

Im Controller

```
<Text fx:id="actionTarget"
strokeType="OUTSIDE"
strokeWidth="0.0" text="0.0"
GridPane.columnIndex="1" />
```

Im FXML File

- Verbindung zwischen EventHandler und ActionEvent über onAction und FXML Annotation:

```
@FXML protected void handleButton(){
   // Do something…
}
```

Im Controller

```
<Button
onAction="#handleButton"
text="Increase" />
```

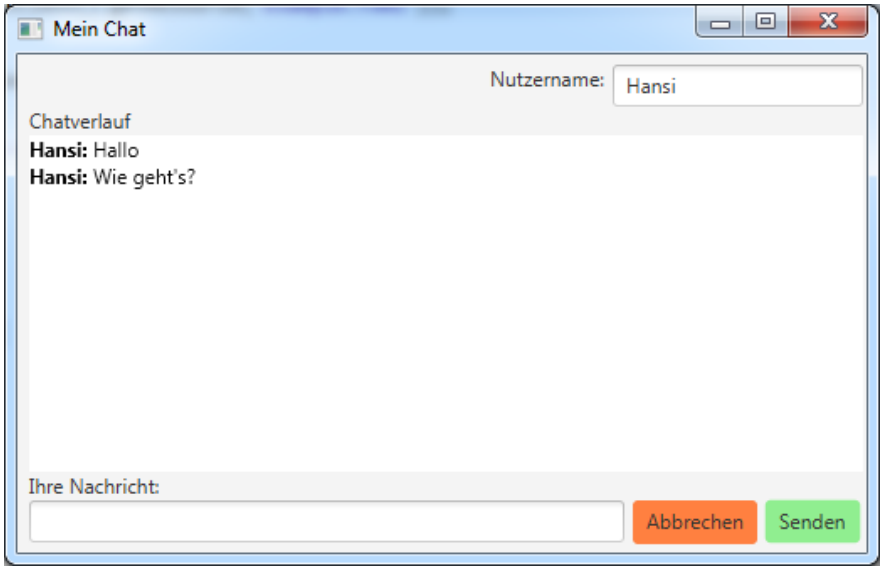
Im FXML File

Beispiel für einen Controller


```
package application;

import java.net.URL;
import java.util.ResourceBundle;

import javafx.fxml.FXML;
import javafx.fxml.Initializable;
import javafx.scene.text.Text;

public class Controller implements Initializable{

    private Model model;

    public Controller(){

        System.out.println("Klasse instanziiert");
    }

    @FXML private Text actionTarget;

    @FXML protected void handleIncreaseButton(){

        model.setCounter(model.getCounter()+1);
        actionTarget.setText(""+model.getCounter());
    }

    @FXML protected void handleDecreaseButton(){
        model.setCounter(model.getCounter()-1);
        actionTarget.setText(""+model.getCounter());
    }

    @Override
    public void initialize(URL location, ResourceBundle resources) {

        System.out.println("Geladen");
        this.model = new Model();
    }
}
```

Aufgabe zu FXML mit Scene Builder

Nutzen Sie nun Ihr erlangtes Wissen über FXML und Scene Builder und gestalten Sie damit eine Chat-GUI, die ungefähr so aussehen könnte:



Schreiben Sie einen geeigneten Controller, der

- Nutzereingaben für Nachrichten und Nutzernamen entgegennimmt
- Beim Drücken des Senden-Buttons diese entsprechend der Grafik im Verlaufsfenster anzeigt.
- Beim Drücken des Abbrechen-Buttons den eingetragenen Text im Nachrichtenfeld wieder löscht

Achten Sie auf eine herkömmliche Usability, d.h.:

- Das Chat-Verlaufsfenster darf nicht direkt editiert werden können
- Beim Absenden einer Nachricht wird automatisch das Nachrichtenfeld geleert.