



LUDWIG-
MAXIMILIANS-
UNIVERSITÄT
MÜNCHEN



 mobile and
distributed systems group



Javakurs für Fortgeschrittene

Einheit 06: Observer Pattern und Scene Builder

Kyrill Schmid

Lehrstuhl für Mobile und Verteilte Systeme



Teil 1: Design-Pattern

- Observer-Pattern

Teil 2: JavaFX mit FXML und Scene Builder

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

Praxis:

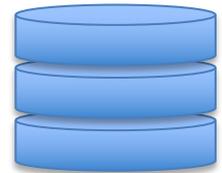
- Observer-Pattern
- FXML mit Scene Builder

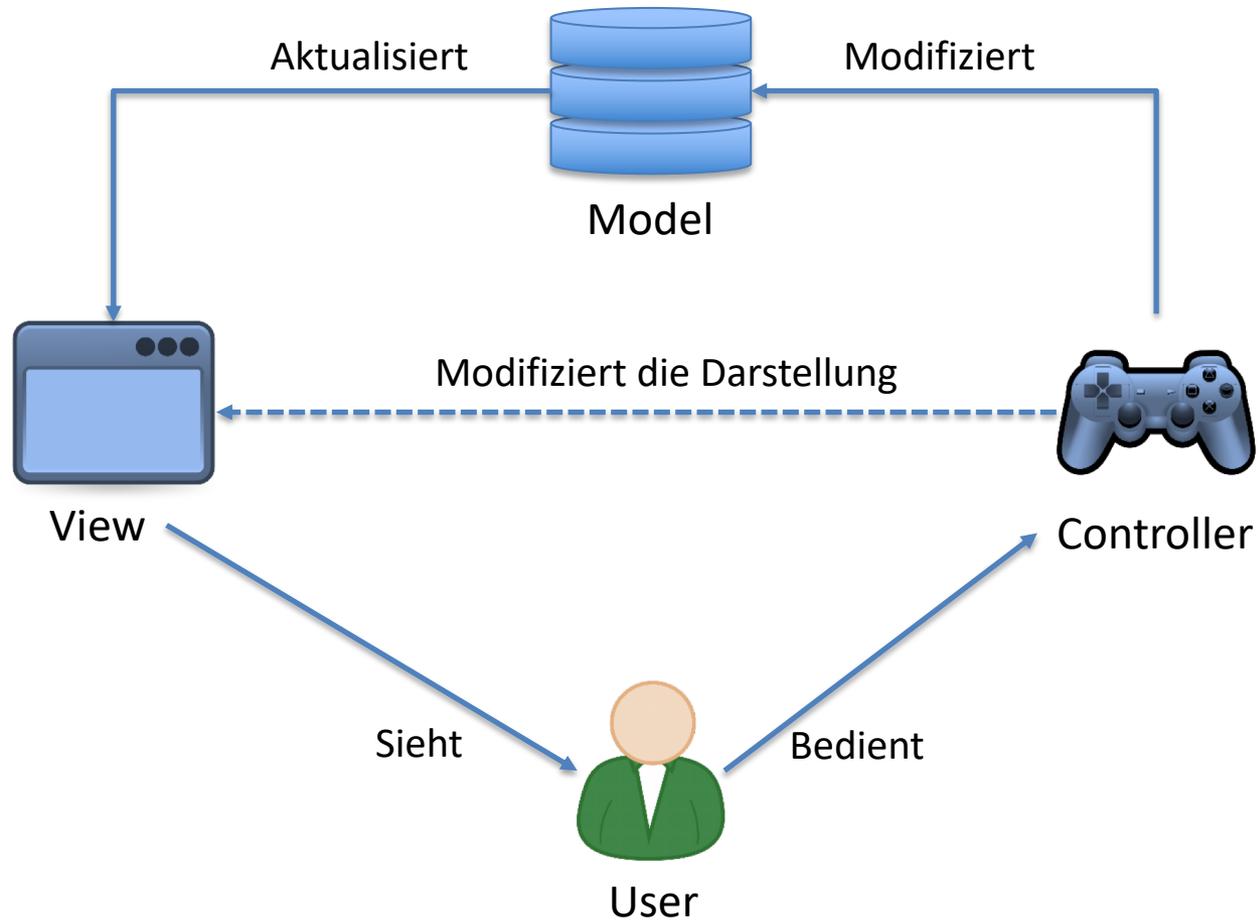
Lernziele

- Das Observer Pattern zur synchronen Datenanzeige verstehen
- Scene Builder und FXML in JavaFX nutzen können

Bei der Programmierung einer grafischen Anwendung haben wir meistens 3 größere Bereiche zu implementieren:

- Das Modell (**M**odel):
 - Ist zuständig für die gesamte Datenhaltung, also dem Zustand des Systems
 - Können über Nutzereingaben (Interaktionen) verändert werden
 - Auch sind hier häufig Regeln und Logik verankert (Backend)
- Die Ansicht (**V**iew):
 - Das ist das, was der Anwender letztendlich sieht (Frontend)
 - also Buttons, Textfelder, Fenster, usw.
 - Mehrere Anzeigen möglich:
 - Engl. vs. deutsche Ansicht, verschiedene Graphen für gleiche Daten, usw.
- Die Steuerung (**C**ontroller):
 - Über die Steuerung kann der Nutzer Einfluss auf die Daten im Model nehmen
 - Bspw.: Anzeigen oder manipulieren
 - Gängige Elemente der Steuerung sind **EventHandler** und **ActionListener**





Model:

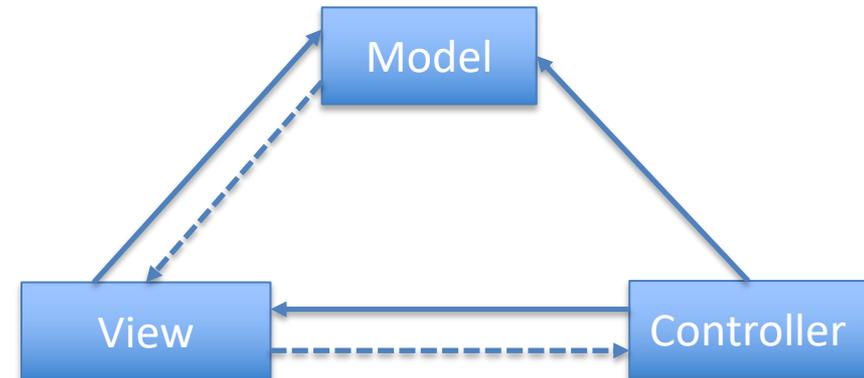
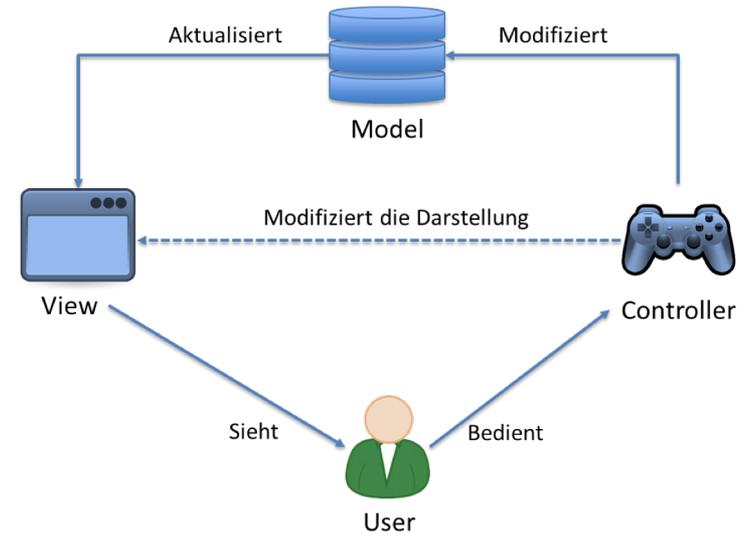
- Unabhängig von View & Controller
- Aktualisierung der View über Beobachter

View:

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

Controller:

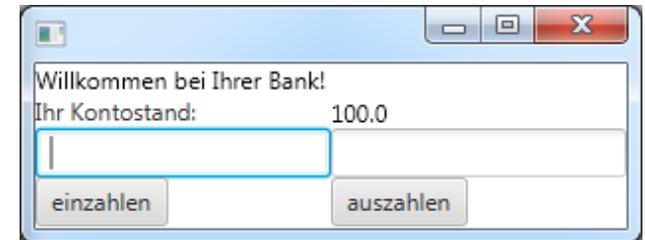
- Verwaltet mind. eine View
- Nimmt Interaktionen entgegen
- Bewirkt Änderungen im Model bzw. der View



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.



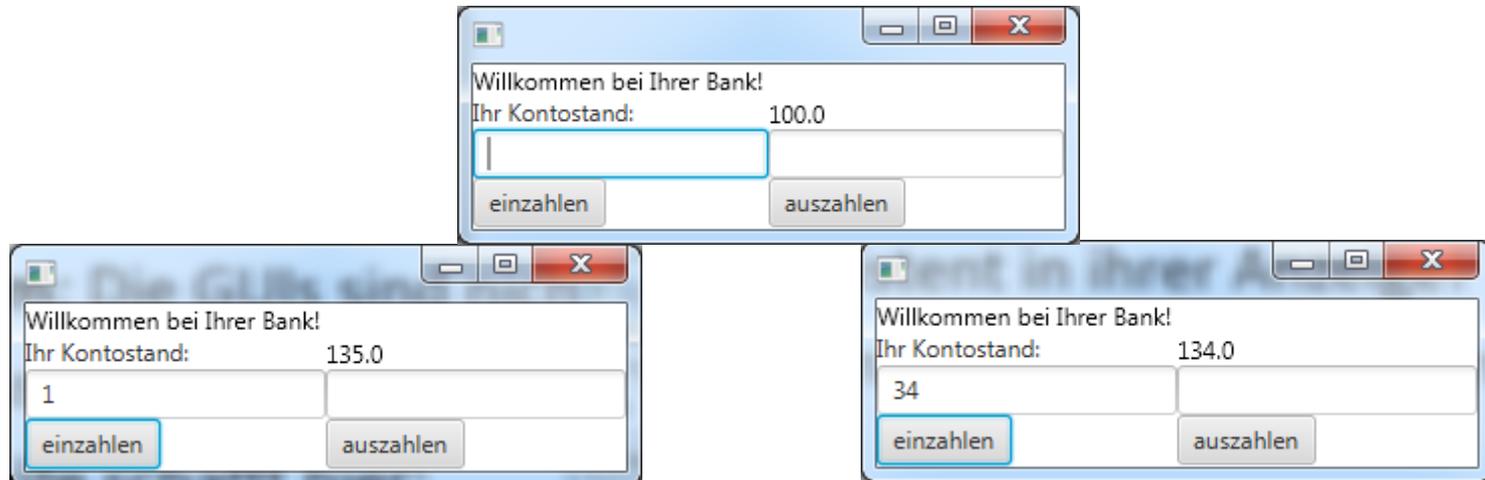
- **Bank:** Ist das eigtl. Model und besitzt eine Instanzvariable für den Kontostand (`double`)
 - Außerdem die Methoden `einzahlen(double Betrag)`, `auszahlen(double Betrag)`
 - Zusätzlich einen Getter für die Instanzvariable des Kontostands
 - Im Konstruktor sollte ein Initialkontostand angegeben werden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

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)



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

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

```
    // Wenn sich was ändert, dann Bescheid geben!  
    public void aendere_etwas(){  
        doSomethingThatChangeSomething();  
        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);`

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!



LUDWIG-
MAXIMILIANS-
UNIVERSITÄT
MÜNCHEN



Teil 2: JavaFX mit FXML und Scene Builder



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

```

1 <?xml version="1.0" encoding="UTF-8"?>
2
3 <?import javafx.geometry.*?>
4 <?import javafx.scene.text.*?>
5 <?import javafx.scene.control.*?>
6 <?import java.lang.*?>
7 <?import javafx.scene.layout.*?>
8 <?import javafx.scene.layout.AnchorPane?>
9
10 <BorderPane maxHeight="-Infinity" maxWidth="-Infinity" minHeight="-Infinity" minWidth="-Infinity" prefHeight="400.0" prefWidth="600.0" xml:lang="en" >
11   <center>
12     <GridPane prefHeight="338.0" prefWidth="600.0" BorderPane.alignment="CENTER">
13       <columnConstraints>
14         <ColumnConstraints hgrow="SOMETIMES" maxWidth="294.0" minWidth="10.0" prefWidth="146.0" />
15         <ColumnConstraints hgrow="SOMETIMES" maxWidth="532.0" minWidth="10.0" prefWidth="454.0" />
16       </columnConstraints>
17       <rowConstraints>
18         <RowConstraints maxHeight="128.0" minHeight="0.0" prefHeight="12.0" vgrow="SOMETIMES" />
19         <RowConstraints maxHeight="319.0" minHeight="10.0" prefHeight="307.0" vgrow="SOMETIMES" />
20         <RowConstraints minHeight="10.0" prefHeight="30.0" vgrow="SOMETIMES" />
21       </rowConstraints>
22       <children>
23         <Label text="Counter" />
24         <Text fx:id="actionTarget" strokeType="OUTSIDE" strokeWidth="0.0" text="0.0" GridPane.columnIndex="1" />
25         <HBox alignment="TOP_CENTER" prefHeight="100.0" prefWidth="200.0" spacing="10.0" GridPane.rowIndex="1">
26           <children>
27             <Button mnemonicParsing="false" onAction="#handleIncreaseButton" text="Increase" />
28             <Button mnemonicParsing="false" onAction="#handleDecreaseButton" prefHeight="25.0" prefWidth="66.0" text="Decrease" />
29           </children>
30         <opaqueInsets />
31         <insets top="4.0" />
32       </opaqueInsets>
33     </HBox>
34   </children>
35 </GridPane>
36 </center>
37 </BorderPane>
38

```

```
<?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

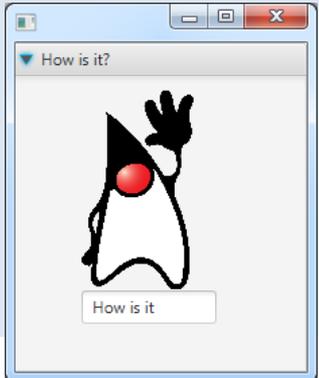
Imports: Wichtig für das Laden der Objekte

Root Node

Definition des XML-Namespaces für fxml und javafx

Kind Knoten

Klassische XML Baumstruktur:
Tags schließen!



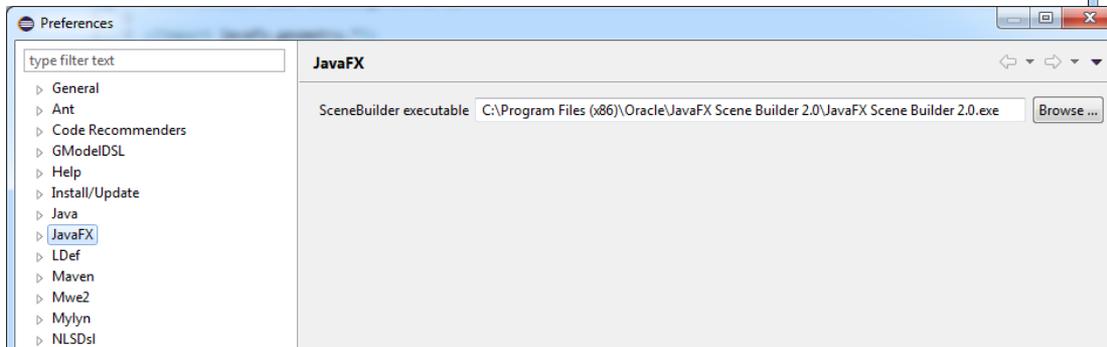
I.d.R. schreibt man eine solche FXML-Datei nicht selbst!

- 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>

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

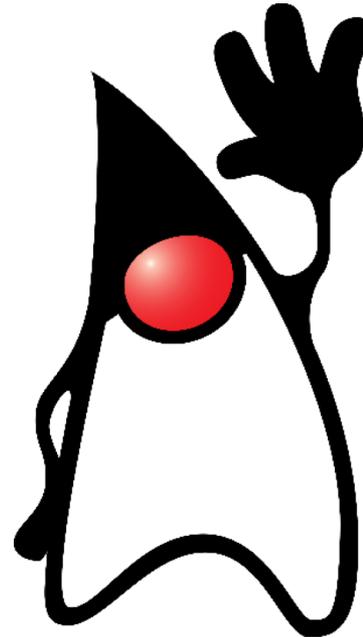


Umgang mit:

- Scene Builder
- FXML und Controller

Was uns dann noch fehlt:

- Properties und Bindings!
- MVC mit JavaFX und FXML
- Weitere Übungen...



Bis nächste Woche...