MV*

Datenhaltung, Benachrichtigung und Binding

Hochschule Rosenheim
University of Applied Sciences

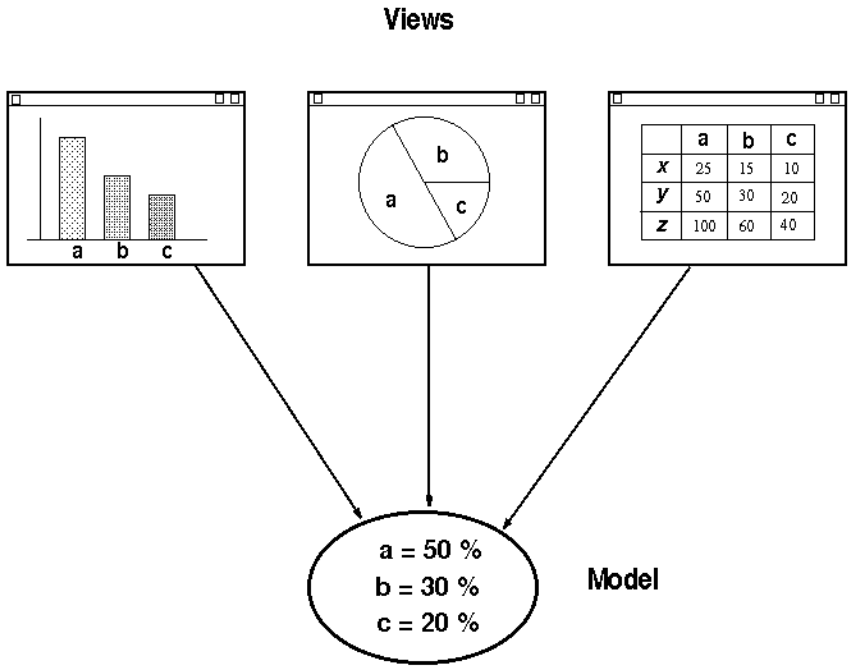
Agenda

1. Überblick und Theorie (20 Minuten)
2. Demo MVP in JavaFX (20 Minuten)
3. Datenmodelle mit JavaFX (20 Minuten)

Hochschule **Rosenheim**
University of Applied Sciences

Datenhaushalt

- Typische Probleme im Datenhaushalt
  - Unterschiedliche Ansichten der gleichen Daten
  - Datenkonsistenz bei Benutzerinteraktion
  - Datenkonsistenz bei Systeminteraktion
  - Trennung von Zuständigkeiten (Separation of Concerns)
  - Erweiterbarkeit neuer Ansichten

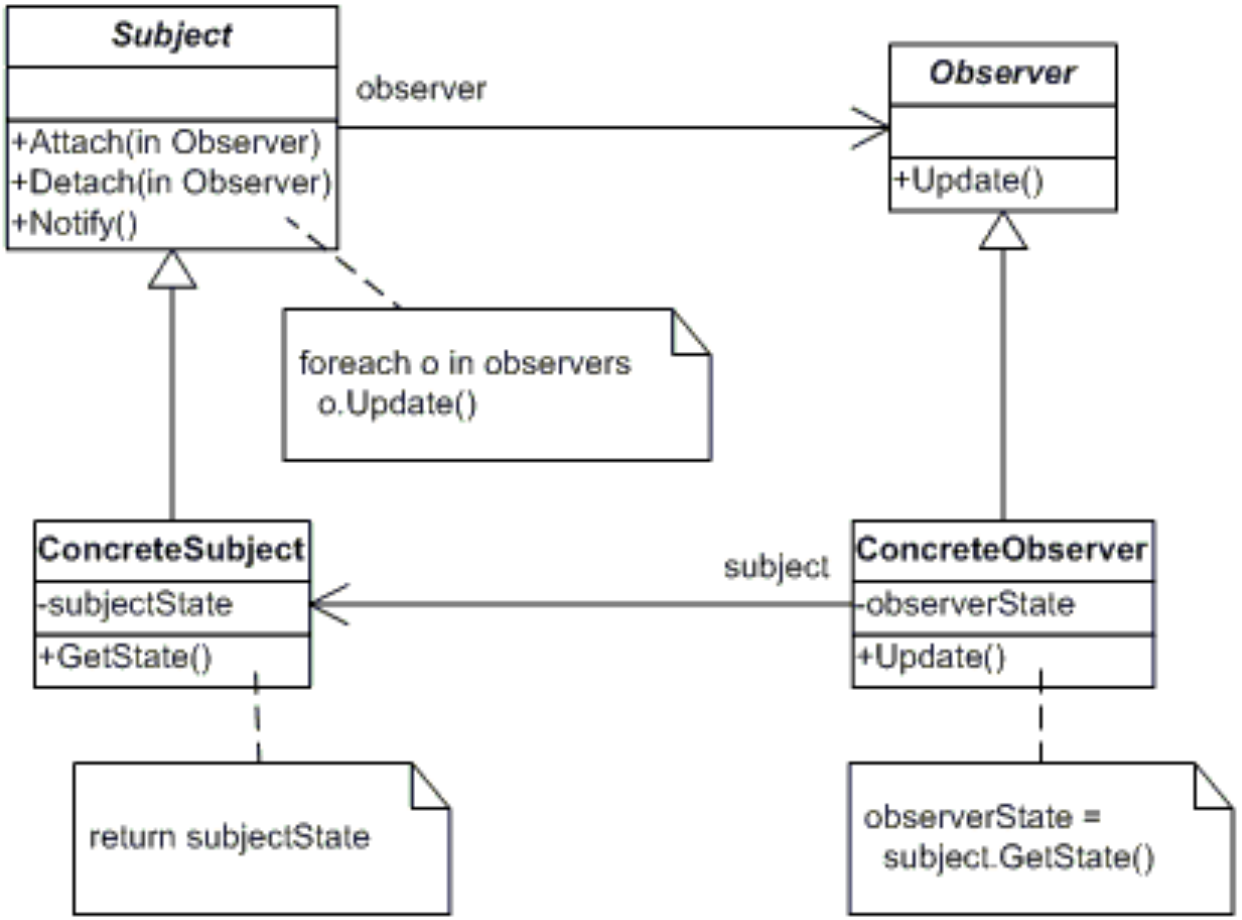


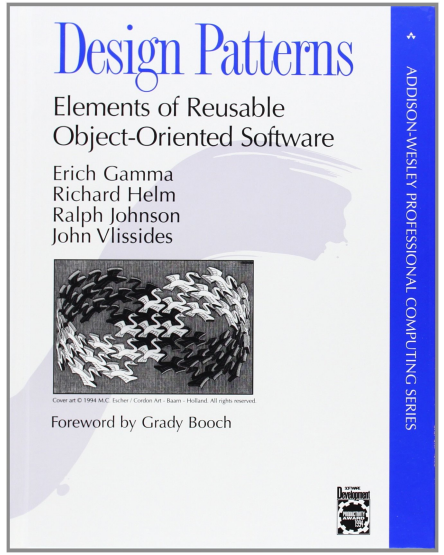
	a	b	c
x	25	15	10
y	50	30	20
z	100	60	40

Grafik aus: E.Gamma et al.: Design Patterns, Addison Wesley, 1995

Hochschule Rosenheim
University of Applied Sciences

Datenkonsistenz basiert auf dem Observer Muster





Hochschule Rosenheim
University of Applied Sciences

Hinweise zum Observer Muster

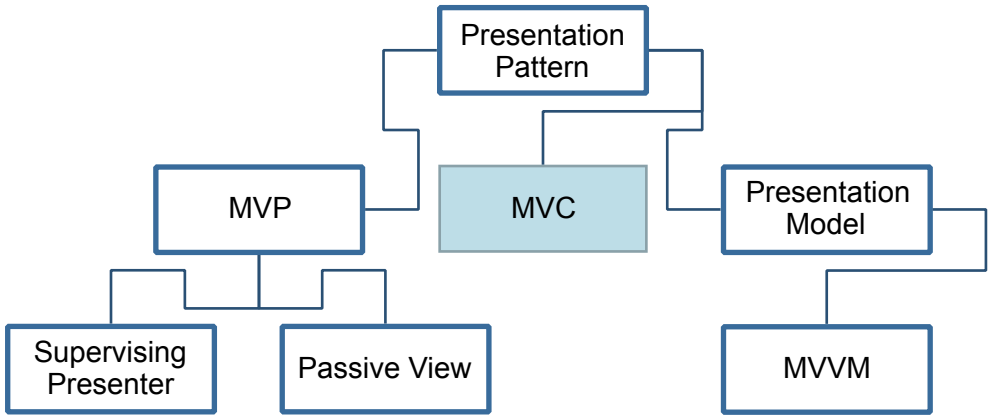
- *Subject* ist eine abstrakte Klasse (mit Implementierung)
- *Observer* ist eine Schnittstelle (ohne Implementierung)
- *Concrete Observer* ist eine Klasse welche die *Observer* Schnittstelle implementiert (z.B. eine Chart-Ansicht)
- *Concrete Subject* ist eine von *Subject* abgeleitete Klasse (z.B. ein SalesModel) das die Methoden von *Subject* benutzt

Hochschule **Rosenheim**
University of Applied Sciences

UI-Patterns helfen den Oberflächen-Code sinnvoll zu strukturieren.

Vorteile

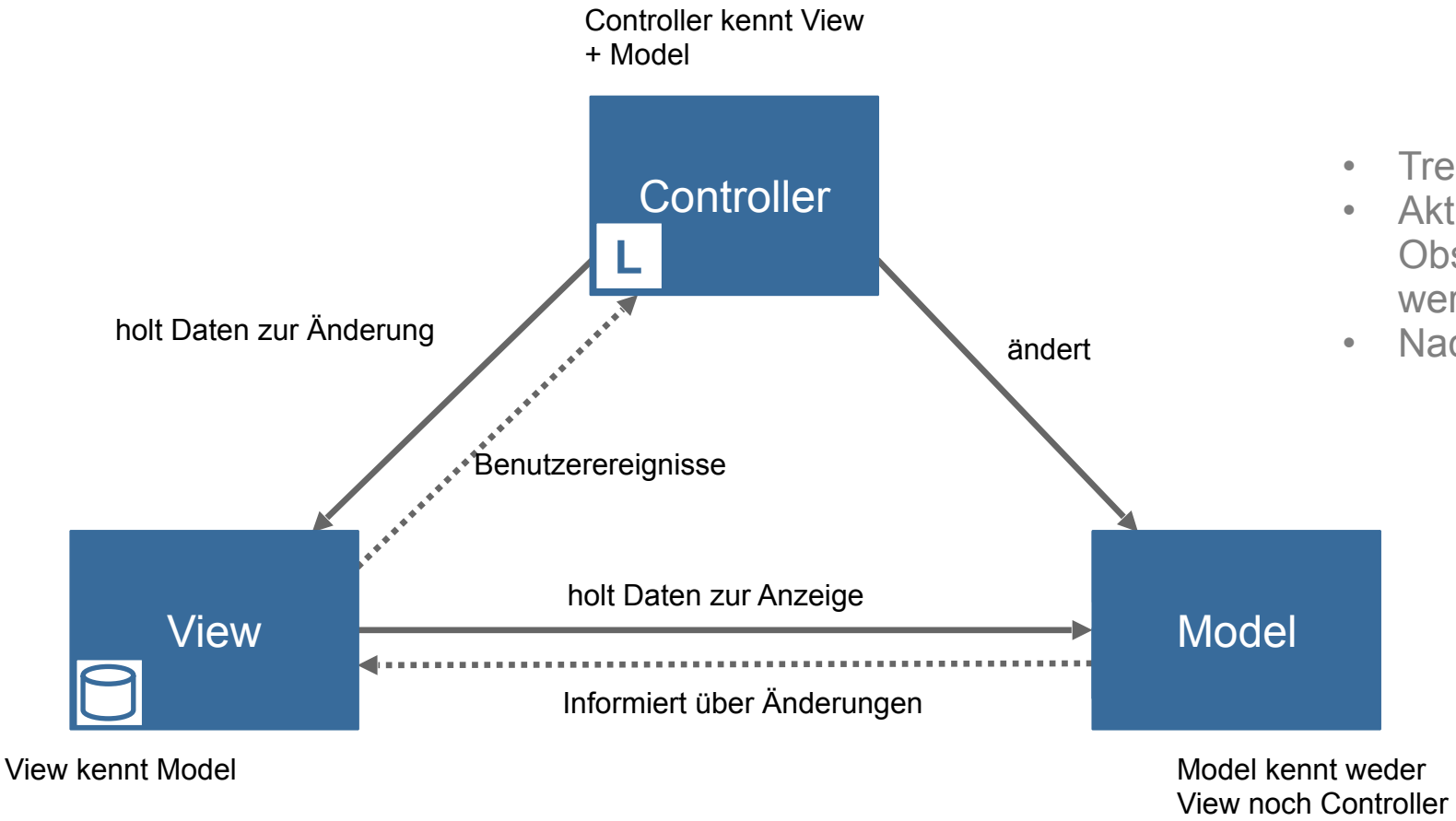
- **Testbarkeit**
  Ermöglichen es UI-Code in Unit Tests unabhängig von Rendering und Benutzer zu testen.
- **Komplexität reduzieren**
  Separation of concerns – Trenne nach technischen Zuständigkeiten, zum Beispiel Darstellung, Daten und Ablauflogik.
- **Modularität**
  Einzelne Teile können durch unterschiedliche Implementierungen ausgetauscht werden, zum Beispiel Dummy- oder Lazy-Implementierung.
- **Wiederverwendung**
  Verwende die gleiche Business-API in verschieden Dialogen.



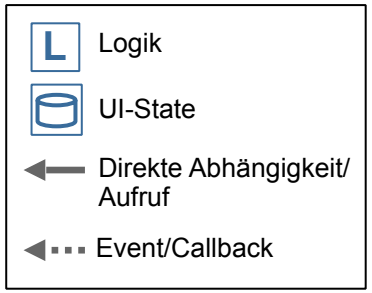
Hochschule Rosenheim
University of Applied Sciences

Classic Model View Controller (MVC)

Smalltalk



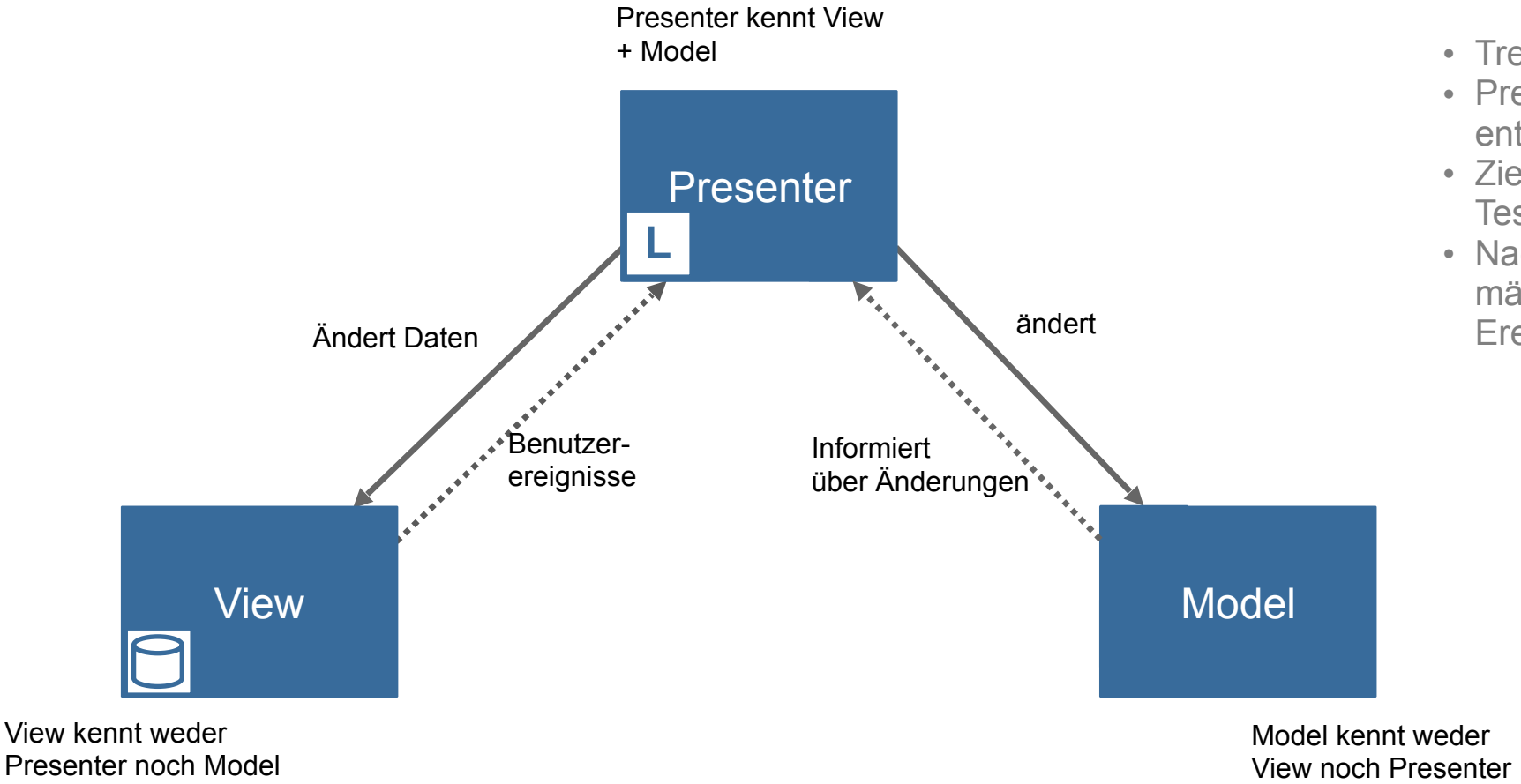
- Trennung von Darstellung, Logik und Model
- Aktualisierungen im Model können über Observer-Pattern an den View gemeldet werden.
- Nachteil: View kennt Model



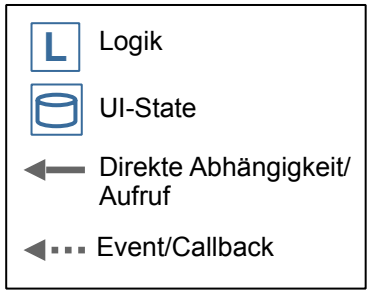
Hochschule Rosenheim
University of Applied Sciences

Classic Model View Presenter (MVP)

OSX



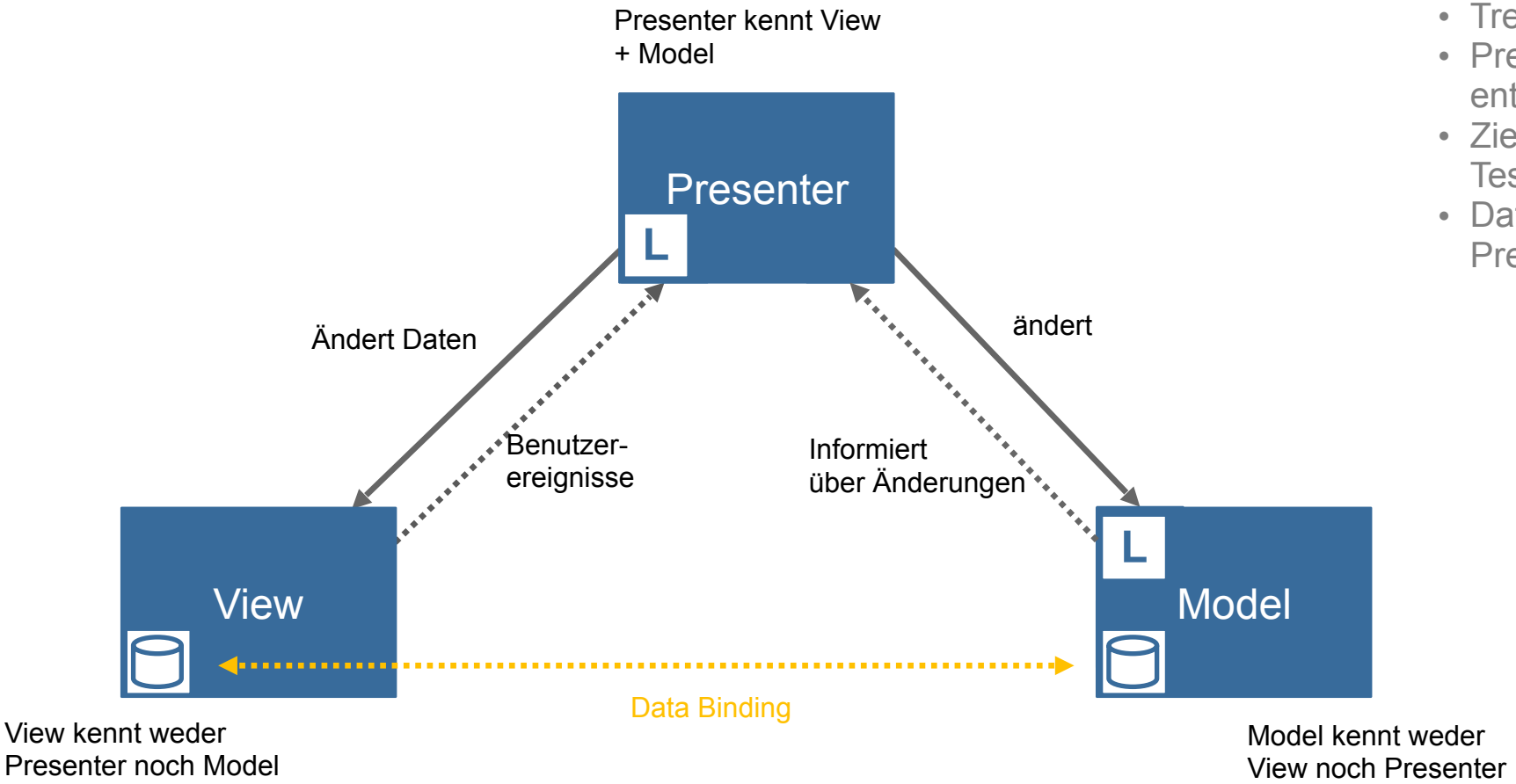
- Trennung von View und Model
- Presenter kann gegen Schnittstellen entwickelt werden
- Ziele: strengere Trennung und gute Testbarkeit
- Nachteil: Presenter wird sehr mächtig und reagiert auf alle Ereignisse.



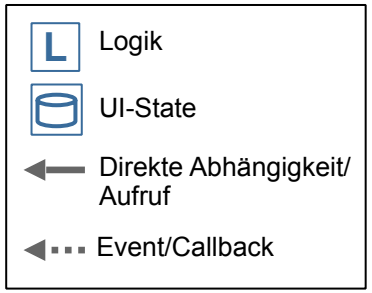
Hochschule Rosenheim
University of Applied Sciences

MVP + Databinding

JavaFX



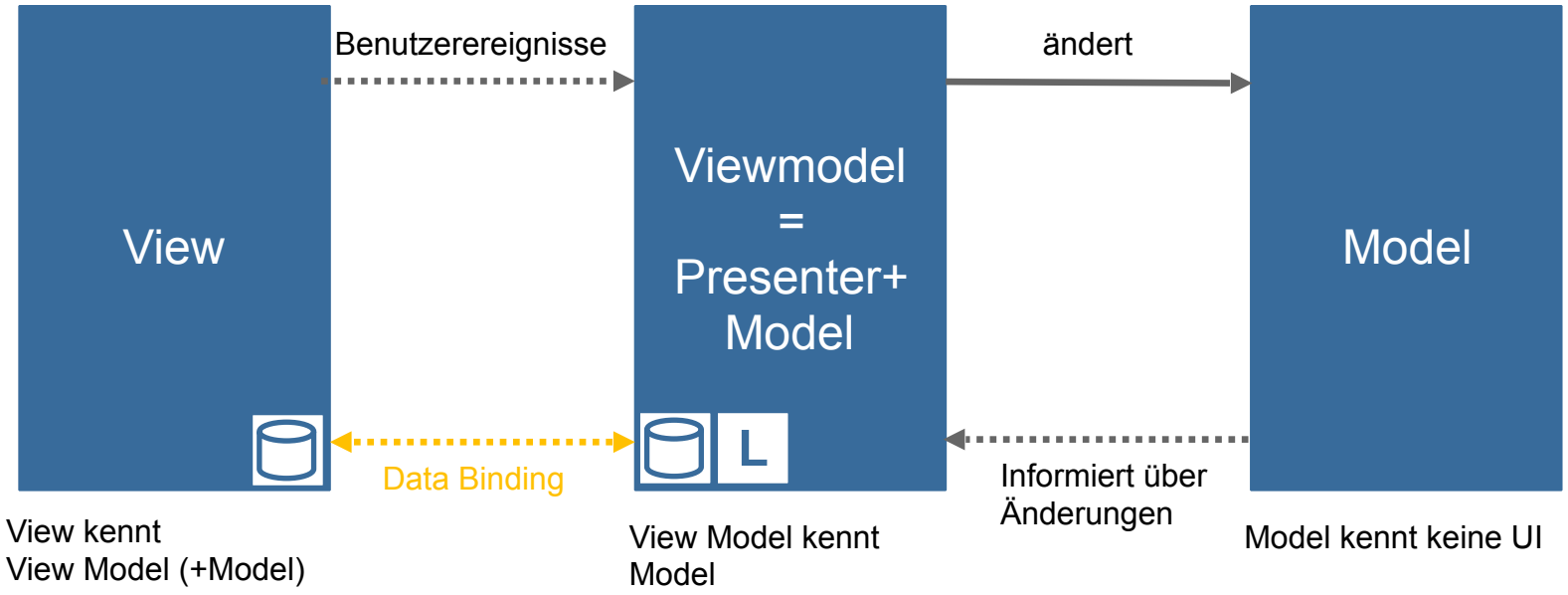
- Trennung von View und Model
- Presenter kann gegen Schnittstellen entwickelt werden
- Ziele: strengere Trennung und gute Testbarkeit
- Databinding minimiert den Presenter-Code

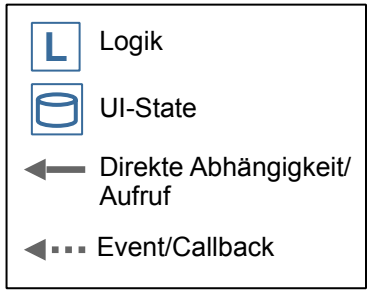


Hochschule Rosenheim
University of Applied Sciences

Microsoft

- **MV(VM) - Model - View - ViewModel**
- **View Model == Presenter + Model in einer Klasse**
- **Fokussiert auf einer klaren Trennung von Business Model und View Model**





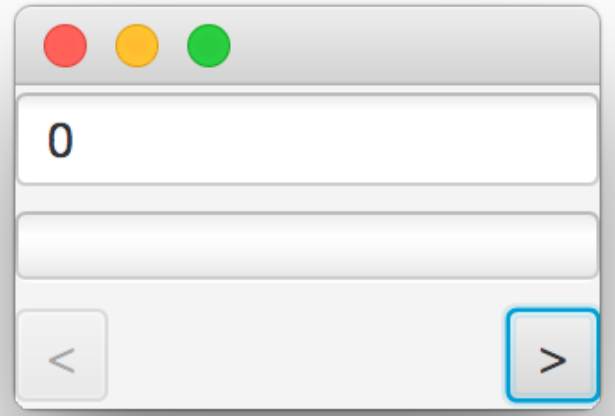
Agenda

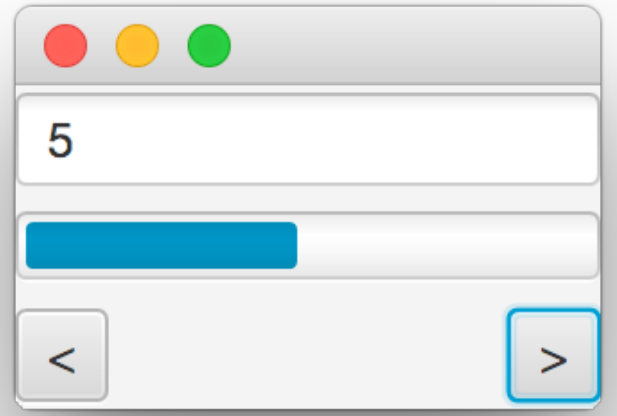
1. Überblick und Theorie (20 Minuten)
2. Demo MVP in JavaFX (20 Minuten)
3. Datenmodelle mit JavaFX (20 Minuten)

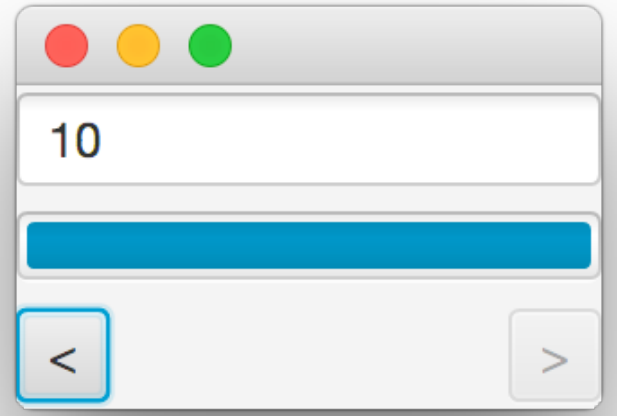
Hochschule **Rosenheim**
University of Applied Sciences

Demo - MVP mit JavaFX

- IntervalModel







Hochschule Rosenheim
University of Applied Sciences

Agenda

1. Überblick und Theorie (20 Minuten)
2. Demo MVP in JavaFX (20 Minuten)
3. **Datenmodelle mit JavaFX (20 Minuten)**

Hochschule **Rosenheim**
University of Applied Sciences

JavaFX Properties sind die Basisbaustein der UI Models

- Getter / Setter müssen final sein
- Für den Aufrufer identisch zu normalen Java Beans
- Zusätzlich: Zugriff auf das Property

```java
package de.fhro.gui.modul5;

import javafx.beans.property.DoubleProperty;
import javafx.beans.property.SimpleDoubleProperty;

public class Bill {

    // Define a variable to store the property
    private DoubleProperty amountDue = new SimpleDoubleProperty();

    // Define a getter for the property's value
    public final double getAmountDue() {
        return amountDue.get();
    }

    // Define a setter for the property's value
    public final void setAmountDue(double value) {
        amountDue.set(value);
    }

    // Define a getter for the property itself
    public DoubleProperty amountDueProperty() {
        return amountDue;
    }
}
```

Properties benachrichtigen bei Änderungen per ChangeListener Interface.

- Properties sind ObservableValues
- Properties sind die Basis für Bindings

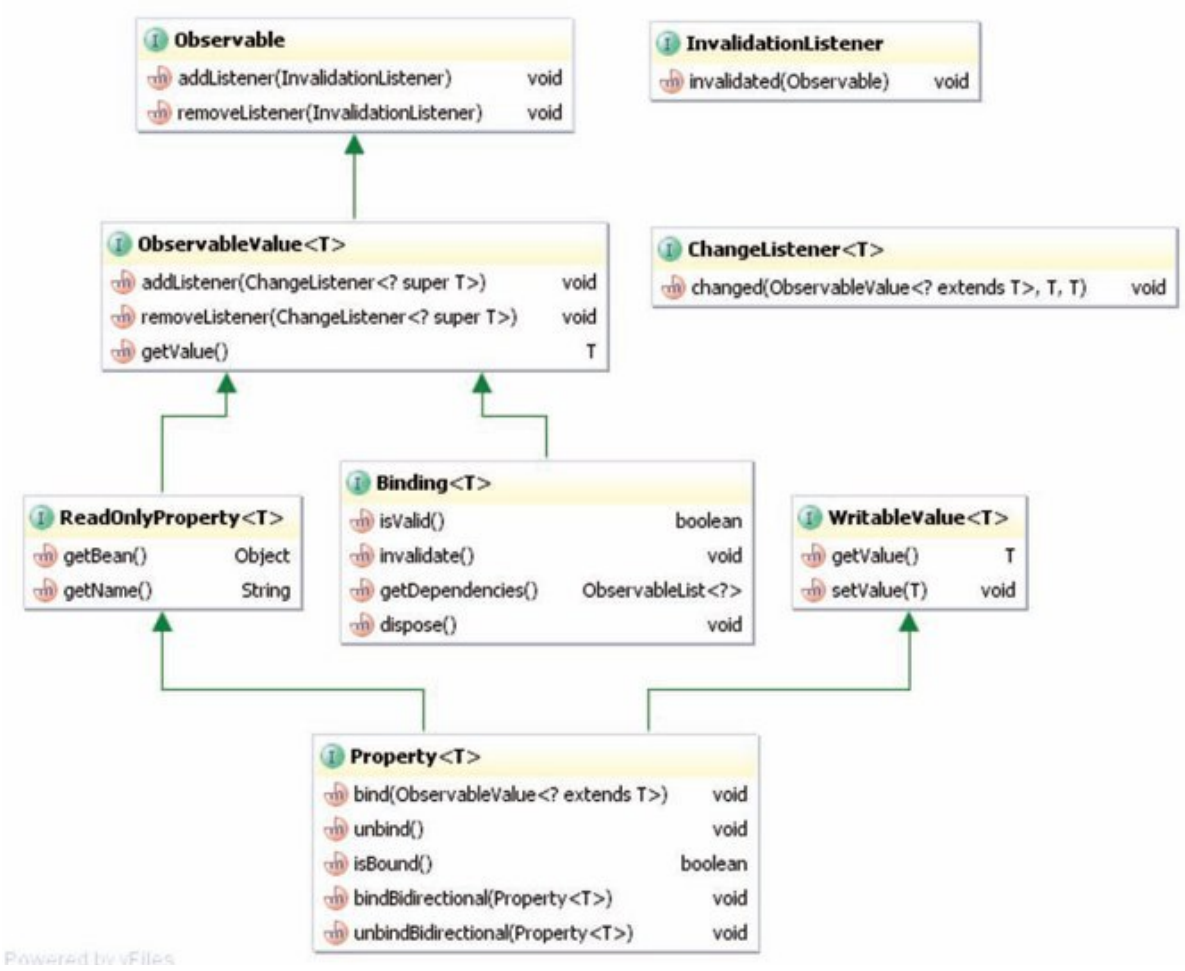
```
Bill bill = new Bill();

bill.amountDueProperty().addListener(new ChangeListener<Number>() {
    public void changed(ObservableValue<? extends Number> observable, Number oldValue, Number newValue) {
        System.out.println("Amount changed: " + newValue);
    }
});

bill.setAmountDue(100f);
```

Hochschule Rosenheim
University of Applied Sciences

ChangeListener / InvalidationListener
Die Trennung ermöglich Lazy Evaluation



Observable wird ungültig -> Neuberechnung notwendig (InvalidationListener werden benachrichtigt)

getValue() – Wenn Neuberechnung notwendig -> Führe Berechnung aus und rufe dann changed() auf.

Hochschule **Rosenheim**
University of Applied Sciences

Was ist Binding?

- Properties lassen sich miteinander verbinden (ähnlich wie in einer Tabellenkalkulaton) – ändert sich ein Wert werden alle anderen automatisch aktualisiert
  - → Binding reduziert die Anzahl von Eventhandlern

```
IntegerProperty num1 = new SimpleIntegerProperty(1);
IntegerProperty num2 = new SimpleIntegerProperty(2);
NumberBinding sum = num1.add(num2);
System.out.println(sum.getValue());
num1.set(2);
System.out.println(sum.getValue());
```

→ 1+2 → 3

→ 2+2 → 4

Hochschule Rosenheim
University of Applied Sciences

Die Hilfklasse „Bindings“ ermöglicht eine flüssige Schreibweise (Fluent-API)

- http://www.martinfowler.com/bliki/FluentInterface.html

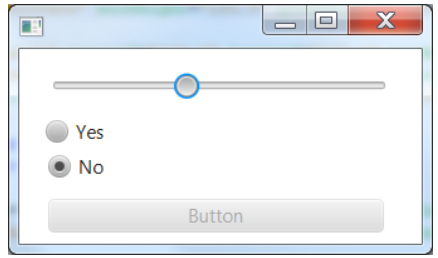
```
private static void bindingFluentSample() {
    IntegerProperty num1 = new SimpleIntegerProperty(1);
    IntegerProperty num2 = new SimpleIntegerProperty(2);
    IntegerProperty num3 = new SimpleIntegerProperty(3);
    IntegerProperty num4 = new SimpleIntegerProperty(4);
    NumberBinding total =
            Bindings.add(num1.multiply(num2), num3.multiply(num4));
    System.out.println(total.getValue());
    num1.setValue(2);
    System.out.println(total.getValue());
}
```

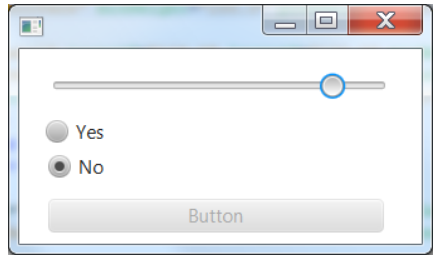

Boolean-Bindings

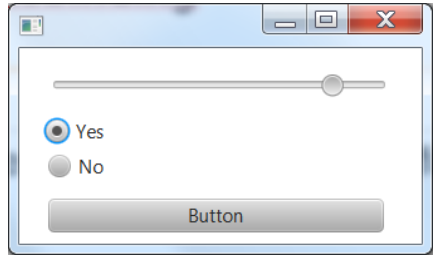
- Button soll „enabled“ sein wenn:
  - A) YES selektiert wurde UND
  - B) Der Slider > 50% ist

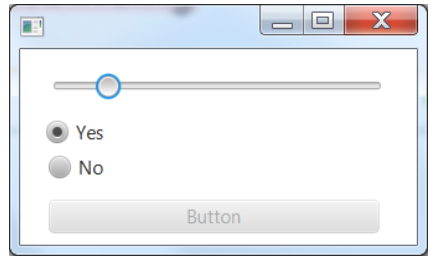
```
// non fluent
button.disableProperty().bind(
        radioYes.selectedProperty().and(
            slider.valueProperty().greaterThan(50d))
        .not());

// fluent: Use Bindings: import static javafx.beans.binding.Bindings.*;
button.disableProperty().bind(
        not(
            and(radioYes.selectedProperty(),
                slider.valueProperty().greaterThan(50d)
            )
        )
);
```









Hochschule Rosenheim
University of Applied Sciences

Zusammenfassung

- Die Basis aller MV* Muster ist das Observer Pattern
- Die MV* Varianten unterscheiden sich in Details
- MVP wird i.R. bei Technologien eingesetzt bei denen der View keinen direkten Zugriff auf das Modell ermöglicht (z.B. FXML)
- Databinding ermöglicht einen generischen Datenaustausch zwischen View und Model. Der Presenter wird dadurch schlanker.