

Grafische Benutzeroberfläche

Teil 2: Eingabemasken

Björn Näf
Dozent

bjoern.naef@edu.teko.ch



Grafische Benutzeroberfläche – Teil 2

▲ Grafische Benutzeroberfläche – Fortsetzung

- Verschiedene Eingabe- und Auswahlfelder einsetzen
- Daten-Bindung
- Mehrere Ansichten dynamisch anzeigen / wechseln
- CRUD-Operationen im UI
- Varia



Neue View erstellen

Ausgangslage: Endresultat aus Aufgabe 5.2

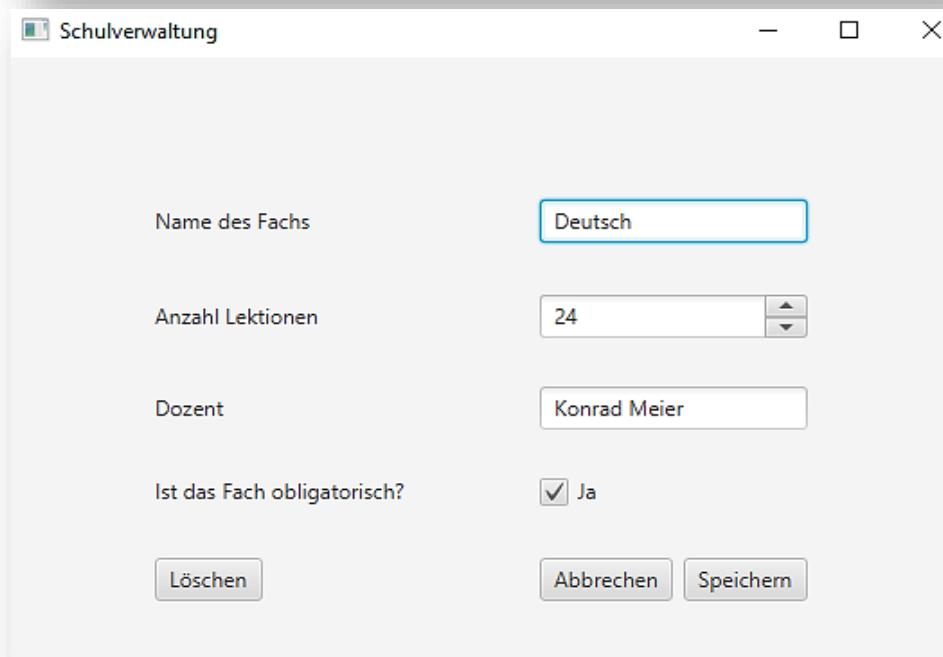
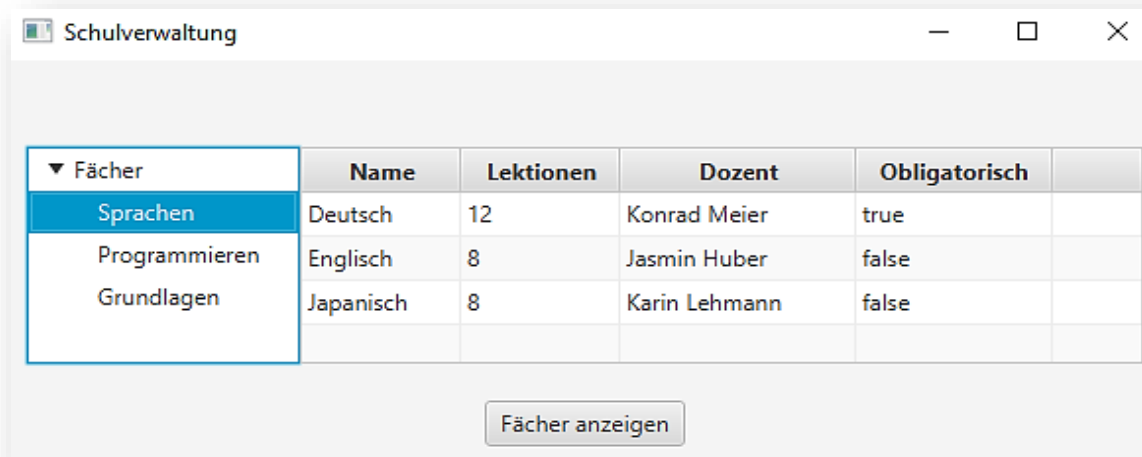
UI einer (einfachen) Schulverwaltungssoftware

▲ Bisher:

- TreeView-Komponente
- TableView-Komponente
- Ereignisverwaltung

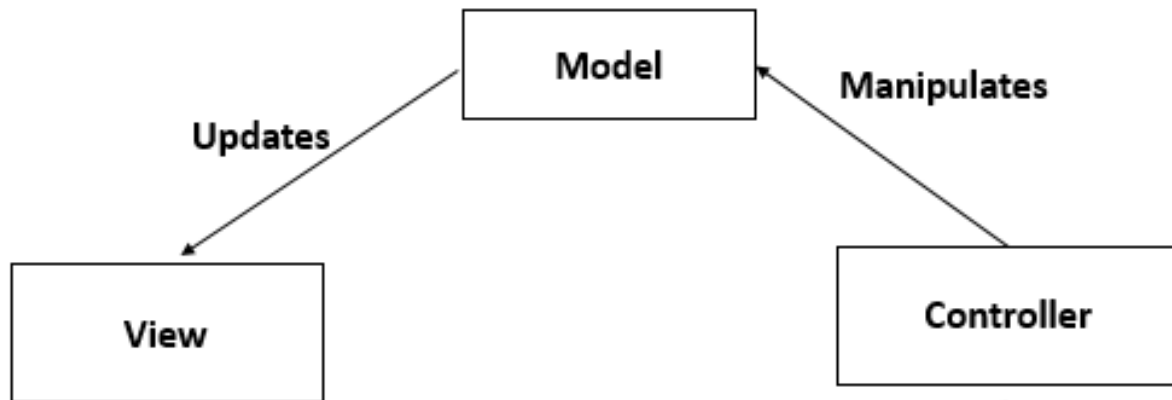
▲ Fächer sollen erstellt, bearbeitet und gelöscht werden können (CRUD-Operationen)

→ Neue (Teil-) View erstellen



Mehrere Views (=Masken) verwenden

Weiterführung des MVC-Konzepts



- ▲ Grundstruktur besteht aus 3 Komponenten
 - ▲ Jede der Komponenten kann mehrfach vorkommen
 - ▲ Mischen / Mehrfachverwendung ist möglich, aber nicht immer sinnvoll, z.B.
 - C1 verarbeitet Inputs von V1 und V2
 - V1 zeigt Daten von M1, M2 und M3 an
 - C2 und C3 verändern Daten von M1
 - etc.
- Achtung: Wartbarkeit wird schnell schwierig!
- Einfacher: 1 View + 1 Controller + n Modell(s)

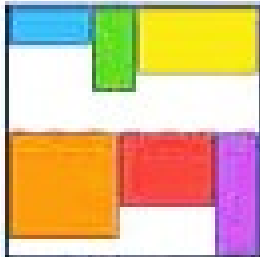
Vorgehen

Die Erstellung und Anbindung einer neuen UI-Maske erfolgt in mehreren Schritten, welche nachfolgenden beschrieben werden:

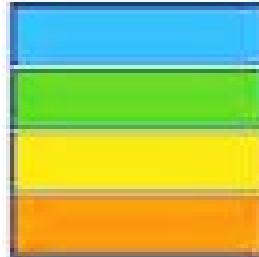
1. Basis-Layout anpassen – neue Teilansicht erstellen
2. UI-Elemente („Controls“) für „Fach“ hinzufügen
3. Model für Datenbindung anpassen
4. Controller erweitern – Neue UI-Elemente und Model referenzieren
5. Editieren (d.h. View umschalten) via Event Handler kontrollieren
6. UI-Elemente an Daten binden
7. Ansichten umschalten

1. Basis-Layout anpassen – neue Teilansicht erstellen

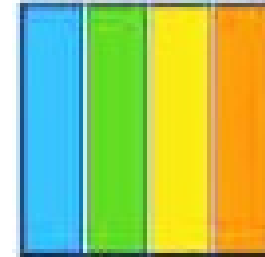
- ▲ Um es einfach zu halten, erweitern wir die bestehende View, anstatt eine neue zu erstellen
- ▲ Dazu legen wir die Teil-Ansichten „übereinander“, blenden aber nur jeweils eine davon ein
- ▲ Damit das darstellerisch funktioniert, müssen wir das Layout unserer App anpassen



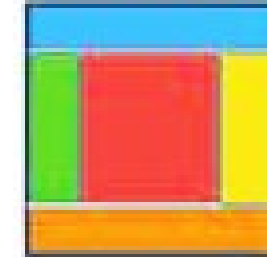
FlowPane



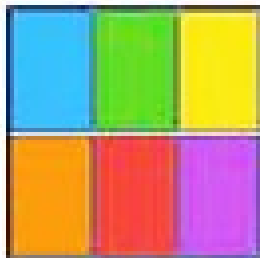
VBox



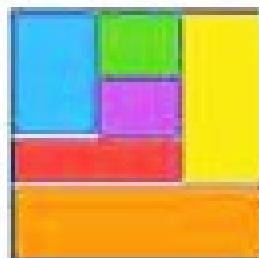
HBox



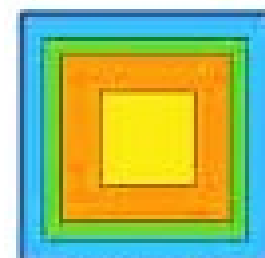
BorderPane



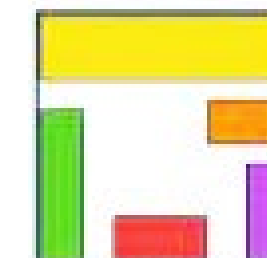
TilePane



GridPane



StackPane



AnchorPane

Neue View erstellen

1. Basis-Layout anpassen – neue Teilansicht erstellen

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

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

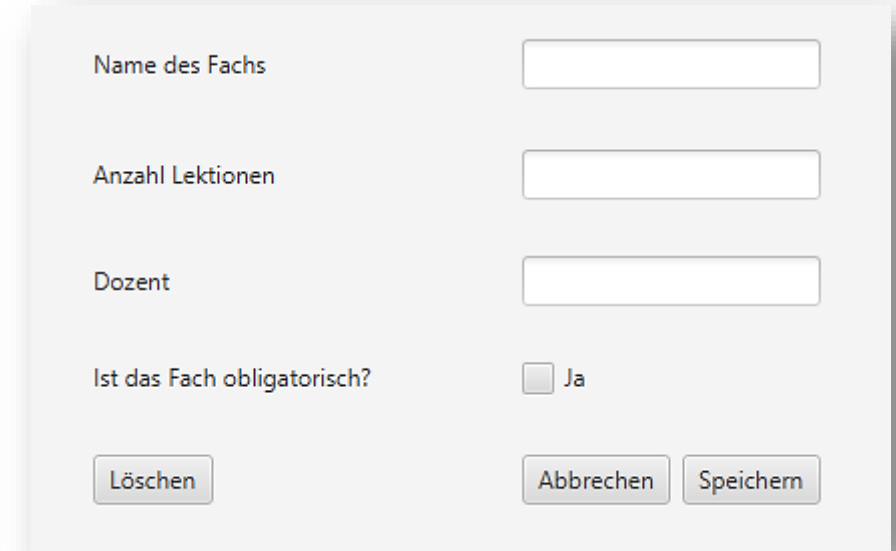
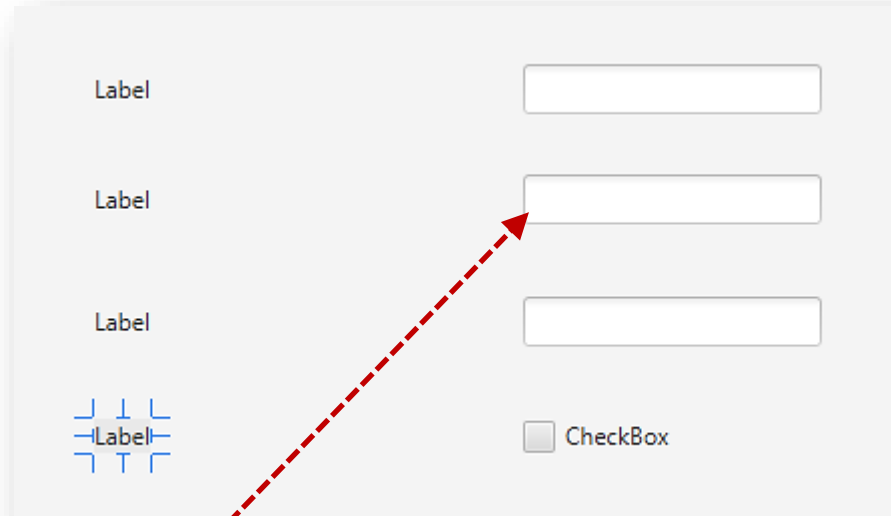
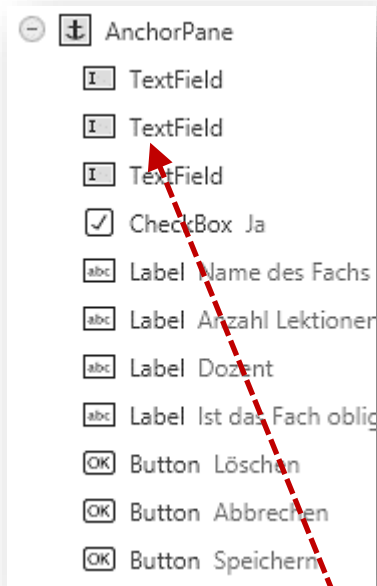
<StackPane xmlns="http://javafx.com/javafx"
  xmlns:fx="http://javafx.com/fxml"
  fx:controller="com.example.demo2.HelloController"
  prefHeight="400.0" prefWidth="600.0">

  <VBox fx:id="listPane">
    <Label fx:id="welcomeText" />
    <HBox prefHeight="200.0" prefWidth="600.0">
      <children>
        <TreeView fx:id="tree" onMouseClicked="#treeMouseClicked" />
        <TableView fx:id="table" onMouseClicked="#tableMouseClicked">
          ...
        </TableView>
      </children>
    </HBox>
    <Button onAction="#onHelloButtonClick" text="Fächer anzeigen" />
  </VBox>

  <AnchorPane fx:id="editPane" visible="false">
    ...
  </AnchorPane>
</StackPane>
```


Neue View erstellen

2. UI-Elemente („Controls“) für „Fach“ hinzufügen (im Markup oder mit Scene Builder)



Hier kann auch z.B. ein
«Spinner» verwendet werden

Achtung: Zur UI-Gestaltung die **Sichtbarkeit** umschalten (Attribut „**visible**“ false / true)

```
<AnchorPane prefHeight="400.0" prefWidth="600.0" visible="true">
  <children>
    <TextField fx:id="fachNameField" layoutX="286.0" layoutY="72.0" />
    <TextField fx:id="fachLektionenField" layoutX="286.0" layoutY="127.0" />
    <TextField fx:id="fachDozentField" layoutX="286.0" layoutY="180.0" />
    <CheckBox fx:id="fachObligatorischField" layoutX="286.0" layoutY="232.0" mnemonicParsing="false" text="Ja" />
    <Label layoutX="72.0" layoutY="76.0" text="Name des Fachs" />
    <Label layoutX="72.0" layoutY="131.0" text="Anzahl Lektionen" />
    <Label layoutX="72.0" layoutY="184.0" text="Dozent" />
    <Label layoutX="72.0" layoutY="232.0" text="Ist das Fach obligatorisch?" />
    <Button layoutX="72.0" layoutY="279.0" mnemonicParsing="false" text="Löschen" />
    <Button layoutX="286.0" layoutY="279.0" mnemonicParsing="false" text="Abbrechen" />
    <Button layoutX="366.0" layoutY="279.0" mnemonicParsing="false" text="Speichern" />
  </children>
</AnchorPane>
```

- ▲ Repetition: Statische Elemente kommen ins **FXML**, dynamische in Controller-Klassen
- ▲ **Wahl des Layouts** ist entscheidend für Flexibilität (Erweiterbarkeit, Portabilität) des UI
- ▲ **Wahl des UI-Elementes** auf Basis der darzustellenden Daten(typen) → nächste Slide
- ▲ Für Datenbindung muss **ID (fx:id) vergeben** werden (Scene Builder: Rechter Bereich „Code“)

UI-Elemente und zugehörige Daten(typen)

Daten	Control	1-N
Textausgabe/-anzeige ohne Eingabefunktion	Label	1
Einfache kurze Texte wie Bezeichnungen etc.	TextField	1
Lange Texte wie Beschreibungen	TextArea	1
Passwörter (versteckte Eingabe)	PasswordField	1
Zahlenwerte	TextField, Spinner , Slider	1
Optionen, zweiwertig („ja“ / „nein“)	Checkbox , ToggleButton, RadioButton, ComboBox	1
Optionen, mehrwertig exklusiv (z.B. S, M, L)	RadioButton, ComboBox, ChoiceBox	1
Optionen, mehrwertig inklusiv (z.B. Beilagen)	Checkboxes, ListView	1
Datum	TextField, DatePicker	1
Menge / Liste von Daten gleicher Struktur	ListView, TreeView, TableView, TreeTableView	N

3. Model für Datenbindung anpassen

- ▲ Damit die Eigenschaften des Models im View ausgelesen und gesetzt werden können, dürfen diese nicht als „private“ markiert sein. Deshalb müssen wir die Klasse anpassen
- ▲ Für UI-Elemente vom Typ „Spinner“ (für Anzahl Lektionen) wird ein anderer Datentyp als Property benötigt – auch dies muss entsprechend in der Klasse Fach angepasst werden:

```
class Fach (name : String, lektionen : Int, dozent : String, obligatorisch : Boolean)
{
    var name: SimpleStringProperty
    var lektionen: ObjectProperty<Int>
    var dozent: SimpleStringProperty
    var obligatorisch: SimpleBooleanProperty
    ...
}
```

- ▲ Achtung: Weiter unten in der Klasse werden weitere zugehörige Anpassungen benötigt.

4. Controller erweitern – Neue UI-Elemente und Model referenzieren

▲ Jetzt muss der Controller um die neuen UI-Elemente und das Modell erweitert werden

```
lateinit var model : Fach
```

```
@FXML
lateinit var listPane : VBox
@FXML
lateinit var editPane : AnchorPane
@FXML
lateinit var fachNameField : TextField
@FXML
lateinit var fachLektionenField : Spinner<Int>
@FXML
lateinit var fachDozentField : TextField
@FXML
lateinit var fachObligatorischField : CheckBox
```

5. Editieren (d.h. View umschalten) via Event Handler kontrollieren

- ▲ Unsere Edit-Maske existiert bisher nur im „Hintergrund“ (d.h. ist am Anfang ausgeblendet)
- ▲ Sie soll anstelle der Listenansicht eingeblendet werden, sobald ein Fach angeklickt wird
- ▲ Dies ist eine dynamische Aktion, die vom Benutzer initiiert wird → Event Handler
- ▲ FXML – Eventhandlermethode deklarieren:

```
<TableView fx:id="table" prefHeight="200.0" prefWidth="440.0" onMouseClicked="#tableMouseClicked">
```

- ▲ Controller – Eventhandlermethode implementieren:

```
fun tableMouseClicked(event: MouseEvent) {  
    val node: Node = event.pickResult.intersectedNode  
    // Accept clicks only on node cells, and not on empty spaces of the TreeView  
    if (node is Text || node is TableCell<*,*> && (node as TableCell<*,*>).text != null) {  
        ...  
    }  
}
```

6. UI-Elemente an Daten binden

- ▲ Damit **Daten** (aus dem Model) im entsprechenden **UI-Element** (z.B. Textfeld) angezeigt und ggf. bearbeitet werden können, müssen UI und Daten „**aneinander gebunden**“ werden
 - Funktion **bind()** bindet UI und Daten unidirektional: Daten werden nur angezeigt (read-only)
 - Funktion **bindBidirectional()** bindet bidirektional: Daten werden angezeigt, können geändert werden
- ▲ Datenbindung funktioniert nur mit den unterdessen bereits bekannten **JavaFX-Properties**
- ▲ Je nach Datentyp des Properties und der zu bindenden Daten ist das Vorgehen leicht anders

6. UI-Elemente an Daten binden

▲ Text (z.B. TextField)

- UiElementDas**Text**Enthält.*textProperty()* .bind (ModelFeldDas**Text**Enthält)

▲ Bool'sche Auswahl-Option (z.B. Checkbox)

- UiElementDas**Bool**Enthält. *selectedProperty()* .bind (ModelFeldDas**Bool**Enthält)

▲ Zahlenwert (z.B. Slider)

- UiElementDas**Zahl**Enthält.*valueProperty()* .bind (ModelFeldDas**Zahl**Enthält)

▲ Zahlenwert in Textfeld (z.B. TextField mit Int-Wert)

- UiElementDas**Text**Enthält.*textProperty()* .bind (ModelFeldDas**Zahl**Enthält, *NumberStringConverter()*)

▲ Zahlenwert in „komplexem“ Feld (z.B. Spinner)

- KomplexesUiElementDas**Zahl**Enthält . *valueFactory*. *valueProperty()* .bind (ModelFeldDas**Zahl**Enthält)

6. UI-Elemente an Daten binden

- ▲ Eine neue Controller-Funktion setzt das „Model“ (=Fach) und richtet die Datenbindung ein
Dies wird benötigt, damit die Daten des gewählten Fachs in der Editmaske erscheinen

```
fun setFach(fach : Fach) {  
  
    model = fach  
  
    fachNameField.textProperty().bindBidirectional(model.name);  
    fachLektionenField.valueFactory.valueProperty().bindBidirectional(model.lektionen);  
    fachDozentField.textProperty().bindBidirectional(model.dozent);  
    fachObligatorischField.selectedProperty().bindBidirectional(model.obligatorisch);  
  
}
```

7. Ansichten umschalten

- ▶ Nach dem Klick auf einen Tabelleneintrag lesen wir das daran gebundene Fach aus
- ▶ Dieses übergeben wir der zuvor erstellten neuen „setFach“-Funktion
- ▶ Schliesslich schalten wir die Ansichten um, d.h. Liste = verstecken, Editmaske = anzeigen

```
fun tableMouseClicked(event: MouseEvent) {  
    val node: Node = event.pickResult.intersectedNode  
    // Accept clicks only on node cells, and not on empty spaces of the TreeView  
    if (node is Text || node is TableCell<*,*> && (node as TableCell<*,*>).text != null) {  
        val fach = table.getSelectionModel().getSelectedItem() as Fach  
        val name = fach.getName() as String  
        println("Table Cell click: $name")  
        setFach(fach)  
        listPane.isVisible = false  
        editPane.isVisible = true  
    }  
}
```

Was noch fehlt...

- ▲ Soweit haben wir eine neue View erstellt, an Daten gebunden und diese angezeigt
- ▲ Die neue Edit-Maske funktioniert aber noch nicht – es fehlen noch die CRUD-Operationen
 - Bearbeite Daten speichern und Tabellenansicht (Liste) entsprechend aktualisieren
 - Bearbeitung abbrechen, ohne zu speichern
 - Datensatz (Fach) löschen
 - Neuen Datensatz anlegen
- ▲ Zudem funktioniert die Datenbindung in der Tabelle noch nicht (immer) korrekt
 - Muss noch verbessert werden

→ Folgt

Aufgabe 5.3

« EditView »

- ▲ Erweitern Sie Ihr Projekt aus Aufgabe 5.2 um die Edit-Ansicht
- ▲ Unterstützen Sie Ihre Klassenkamerad:innen bei Schwierigkeiten
- ▲ Die Umsetzung funktioniert korrekt, wenn beim Anklicken eines Eintrags (Fachs) im TableView die Edit-View inkl. Daten erscheint:

The screenshot shows a window titled 'Schulverwaltung' with the following form elements:

- Name des Fachs:** A text input field containing 'Deutsch'.
- Anzahl Lektionen:** A numeric input field with a spinner, showing '24'.
- Dozent:** A text input field containing 'Konrad Meier'.
- Ist das Fach obligatorisch?:** A checkbox labeled 'Ja' which is checked.
- Buttons:** At the bottom, there are three buttons: 'Löschen' (Delete), 'Abbrechen' (Cancel), and 'Speichern' (Save).

The background of the slide features a series of overlapping, wavy lines in shades of red, orange, and green. These lines create a sense of depth and movement, resembling a stylized landscape or a data visualization. A bright, circular light source is positioned in the upper center, casting a soft glow across the scene. The overall color palette is warm and vibrant, with a gradient from light blue at the top to a darker purple at the bottom.

Neue View erweitern – CRUD-Operationen einbauen

Vorgehen

Die zuvor genannten fehlenden Funktionen sollen jetzt Schritt für Schritt eingebaut werden:

1. Datenbindung in Modell-Klasse verbessern
2. Event Handler für Buttons hinzufügen
3. CRUD: Event Handler fürs Speichern implementieren
4. CRUD: Event Handler fürs Abbrechen implementieren
5. CRUD: Event Handler fürs Löschen implementieren
6. Refactoring
7. CRUD: Button fürs Erstellen neuer Fächer hinzufügen
8. CRUD: Event Handler fürs Erstellen neuer Fächer implementieren

1. Datenbindung in Modell-Klasse verbessern

- Damit die Daten in der **Tabelle korrekt angezeigt und aktualisiert** werden, muss die Modell-Klasse („Fach“) um **zusätzliche Getter-Funktionen** erweitert werden (die bisher verwendete Variante war eine Vereinfachung, die nicht korrekt funktioniert)

```
fun nameProperty() : StringProperty {  
    return name  
}  
  
fun lektionenProperty() : ObjectProperty<Int> {  
    return lektionen  
}  
  
fun dozentProperty() : StringProperty {  
    return dozent  
}  
  
fun obligatorischProperty() : BooleanProperty {  
    return obligatorisch  
}
```

- Anmerkung: Dies ist JavaFX-Logik – besser nicht hinterfragen, sondern einfach gemäss Konvention umsetzen

2. Event Handler für Buttons hinzufügen

- ▲ FXML: Wie bereits bekannt, onAction-Event mit (neuer) Funktion ausstatten
- ▲ Noch fehlende EventHandler-Funktionen durch IDE generieren lassen (rote Glühbirne)

```
<Button layoutX="72.0" layoutY="279.0" mnemonicParsing="false" text="Löschen" onAction="#onDeleteClick" />  
<Button layoutX="286.0" layoutY="279.0" mnemonicParsing="false" text="Abbrechen" onAction="#onCancelClick" />  
<Button layoutX="366.0" layoutY="279.0" mnemonicParsing="false" text="Speichern" onAction="#onSaveClick" />
```

- ▲ Controller: (Noch) leere EventHandler-Funktionen (Parameter werden nicht benötigt)

```
fun onDeleteClick() {  
}  
  
fun onCancelClick() {  
}  
  
fun onSaveClick() {  
}
```


3. CRUD: Event Handler fürs Speichern implementieren

- ▲ Speichern ist maximal einfach – es funktioniert bei korrekter Datenbindung automatisch! (in dieser einfachen Variante; Persistierung in eine DB braucht Zusatzcode → folgt später)
- ▲ Es müssen einzig die Ansichten wieder zur Tabellen/Listen-Ansicht umgeschaltet werden

```
fun onSaveClick() {  
    listPane.isVisible = true  
    editPane.isVisible = false  
}
```

4. CRUD: Event Handler fürs Abbrechen implementieren

- ▲ Der Nachteil der automatischen Datenbindung zeigt sich beim Abbrechen:
Datenmutationen sind sofort produktiv und lassen sich nicht (einfach) rückgängig machen
- ▲ Lösung: Daten vor der Bearbeitung in eine temporäre **Kopie** zwischenspeichern
- ▲ Controller: Kopie anlegen und mit aktuellen Daten befüllen

```
lateinit var oldModel: Fach
```

```
fun setFach(fach : Fach) {  
    model = fach  
    oldModel = Fach(fach.getName(), fach.getLektionen(), fach.getDozent(), fach.getObligatorisch())  
    ...  
}
```

4. CRUD: Event Handler fürs Abbrechen implementieren

- ▲ Controller: Beim Klick auf „Abbrechen“ Daten zurückkopieren
- ▲ Tabelle aktualisieren (ist in diesem Fall nötig, da geänderte Daten bereits übernommen)
- ▲ Ansichten umschalten (analog Speichern)

```
fun onCancelClick() {  
  
    model.setName(oldModel.getName())  
    model.setLektionen(oldModel.getLektionen())  
    model.setDozent(oldModel.getDozent())  
    model.setObligatorisch(oldModel.getObligatorisch())  
  
    table.refresh()  
  
    listPane.isVisible = true  
    editPane.isVisible = false  
  
}
```

5. CRUD: Event Handler fürs Löschen implementieren

- ▲ Beim Löschen von Daten zeigt sich ein weiterer Vorteil des JavaFX-Datenbindings:
Entfernen des betreffenden Objekts aus der internen Liste → Entfernen aus der Tabelle
- ▲ Die Tabelle muss in diesem Fall nicht einmal aktualisiert werden (geschieht automatisch)
- ▲ Da wir keine Fremdschlüssel modelliert haben, löschen wir einfach aus allen drei Listen

```
fun onDeleteClick() {  
    deleteFachFromList(faecher1)  
    deleteFachFromList(faecher2)  
    deleteFachFromList(faecher3)  
    listPane.isVisible = true  
    editPane.isVisible = false  
}
```

```
fun deleteFachFromList(liste : ObservableList<Fach>) {  
    var fachToDelete = liste.find { f -> f.getName() == model.getName() }  
    if (liste.contains(fachToDelete)) liste.remove(fachToDelete!!)  
}
```

6. Refactoring

- Der tableMouseClicked- sowie alle drei Button Event Handler schalten die Ansichten um

→ Auslagerung in separate Funktion („Refactoring“)

```
fun tableMouseClicked(event: MouseEvent) {  
    ...  
    showEditPane(true)  
    ...  
}  
  
fun onDeleteClick() {  
    ...  
    showEditPane(false)  
}  
  
fun onCancelClick() {  
    ...  
    showEditPane(false)  
}  
  
fun onSaveClick() {  
    showEditPane(false)  
}  
  
fun showEditPane(visible : Boolean)  
{  
    listPane.isVisible = !visible  
    editPane.isVisible = visible  
}
```

7. CRUD: Button fürs Erstellen neuer Fächer hinzufügen

- ▲ Auf der Edit-Maske haben wir Operation für Speichern, Abbrechen und Löschen eingebaut
- ▲ Von den CRUD-Operationen fehlt noch Speichern – dies bauen wir auf der **Listenansicht** ein

```
<VBox fx:id="listPane">
  <Label fx:id="welcomeText" />
  <HBox prefHeight="200.0" prefWidth="600.0">
    <children>
      <TreeView fx:id="tree" prefHeight="200.0" prefWidth="150.0" onMouseClicked="#treeMouseClicked" />
      <TableView fx:id="table" prefHeight="200.0" prefWidth="440.0" onMouseClicked="#tableMouseClicked">
        <columns>
          <TableColumn prefWidth="81.0" text="Name" />
          <TableColumn prefWidth="81.0" text="Lektionen" />
          <TableColumn prefWidth="120.0" text="Dozent" />
          <TableColumn prefWidth="100.0" text="Obligatorisch" />
        </columns>
      </TableView>
    </children>
  </HBox>
  <Button onAction="#onHelloButtonClick" text="Fächer anzeigen" />
  <Button onAction="#onCreateClick" text="Hinzufügen" />
</VBox>
```

8. CRUD: Event Handler fürs Erstellen neuer Fächer implementieren

- ▲ Nun fügt der entsprechende Event Handler **der gewählten Fächerliste einen leeren Eintrag (Fach) hinzu** und ruft danach damit die **Edit-Maske** zur Bearbeitung des Eintrags auf
- ▲ Mittels **Ausnahmebehandlung** (Exception Handling) reagieren wir auf Laufzeitfehler, z.B. wenn der TreeView noch nicht mit Daten befüllt oder keine Fächerliste ausgewählt wurde. Eine entsprechende Fehlermeldung geben wir vorerst nur über die Konsole aus.

```
try {  
    // Auszuführender Code, welcher potentiell zu einem Laufzeitfehler führt  
}  
catch (exception : Exception) {  
    // Code, der ausgeführt wird, nachdem ein Laufzeitfehler aufgetreten ist  
}
```

- ▲ Mithilfe des **return** Statements bewirken wir den vorzeitigen Abbruch der Funktion

```
showEditPane(true)  
return
```

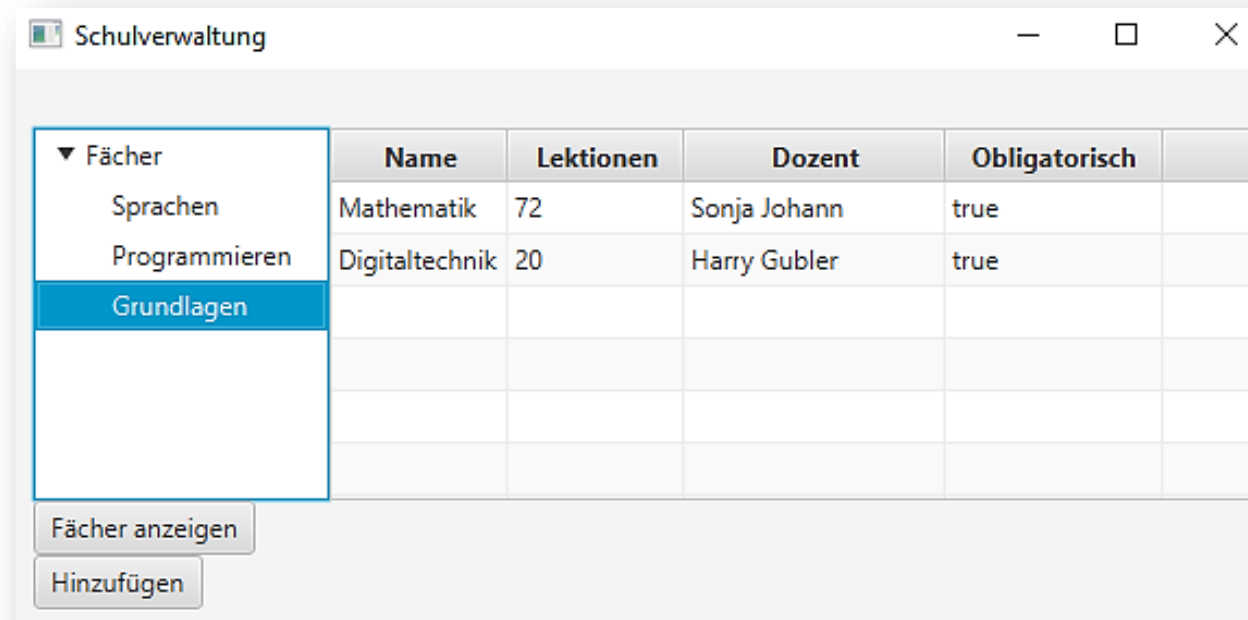
8. CRUD: Event Handler fürs Erstellen neuer Fächer implementieren

```
fun onCreateClick() {
    try {
        val fachbereich = (tree.getSelectionModel().getSelectedItem() as TreeItem<*>).value as String
        if (fachbereich != null && fachbereich != "") {
            var newFach = Fach("", 0, "", false)
            var proceed = true
            when (fachbereich) {
                "Sprachen" -> faecher1.add(newFach)
                "Programmieren" -> faecher2.add(newFach)
                "Grundlagen" -> faecher3.add(newFach)
                else -> proceed = false
            }
            if (proceed) {
                setFach(newFach)
                showEditPane(true)
                return
            }
        }
    }
    catch (exception : Exception) {}
    println("Bitte Fachbereich wählen, um Fach zu erstellen")
}
```


Aufgabe 5.4

« CRUD-Operationen »

- ▲ Erweitern Sie Ihr Projekt aus Aufgabe 5.3 um die CRUD-Operationen
- ▲ Unterstützen Sie Ihre Klassenkamerad:innen bei Schwierigkeiten
- ▲ Die Umsetzung ist korrekt, wenn das Erstellen, Speichern, Abbrechen und Löschen eines Fachs fehlerfrei funktioniert.



	Name	Lektionen	Dozent	Obligatorisch
Sprachen	Mathematik	72	Sonja Johann	true
Programmieren	Digitaltechnik	20	Harry Gubler	true
Grundlagen				



Anwendung optimieren – Usability steigern

Bis hierher funktioniert unsere Anwendung mit allen grundlegenden CRUD-Operationen, jedoch lässt sich das Benutzererlebnis noch verbessern:

▲ Dialoge

- Speichern, Abbrechen, Löschen werden ohne Bestätigung vorgenommen, Fehlermeldungen fehlen
- Zur Erhöhung der Usability können die bereits bekannten Info- und Confirm-Dialoge eingebaut werden

▲ UI-Elemente

- Es wurden bisher einfache, aber nicht immer die benutzerfreundlichsten Controls eingesetzt
- z.B. Spinner kann zur Lektionen-Wahl, ToggleButton für die Obligatorisch-Option verwendet werden

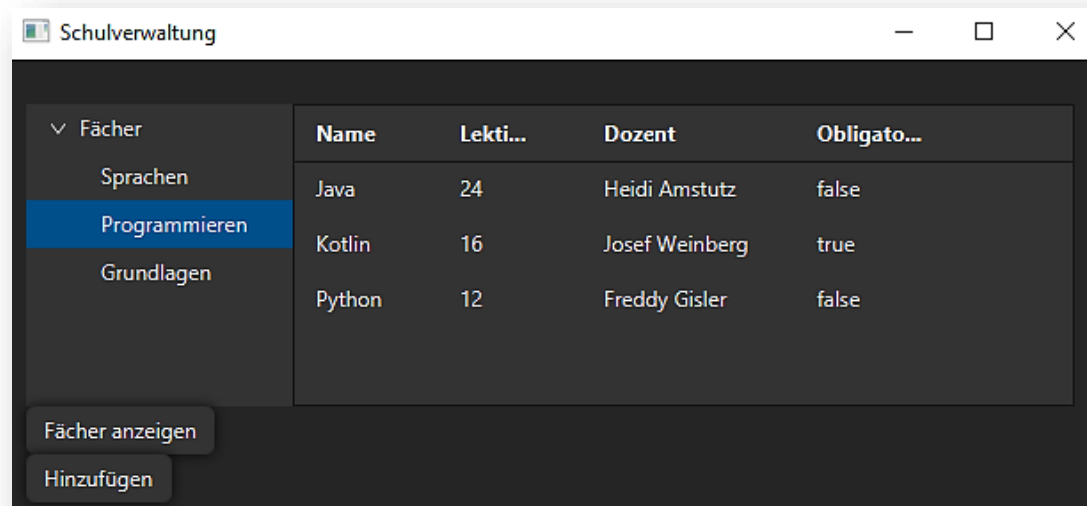
▲ Layout / Design

- Die Anwendung funktioniert, ist aber noch nicht übermässig „schön“
- So können Buttons umpositioniert, Abmessungen und Abstände optimiert, Styles eingebaut werden

Aufgabe 5.5

« Optimierung »

- Führen Sie am Projekt aus Aufgabe 5.4 einige Verbesserungen durch
 - Confirm-Dialoge für Speichern, Abbrechen und Löschen
 - Info-Dialog für fehlende Fächerauswahl beim Erstellen
 - Hinzufügen-Button rechts neben Fächer anzeigen Button stellen
 - Spinner (falls nicht bereits verwendet) für Lektionen-Auswahl
 - ToggleButton für Obligatorisch-Option



... Datenbank-Anbindung!

→ folgt nächstes Mal



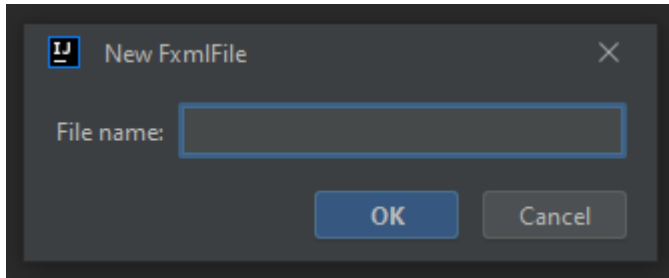
Anhang: Separate View & Controller erstellen

“Advanced”

Neue View erstellen – Variante separates FXML

1. Neue FXML-Datei erzeugen

Im Projektextplorer Rechtsklick auf Ordner, der die bestehende FXML-Datei enthält
→ New... → FXML File → Name (z.B. „fach-view“) eingeben und mit OK bestätigen



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

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

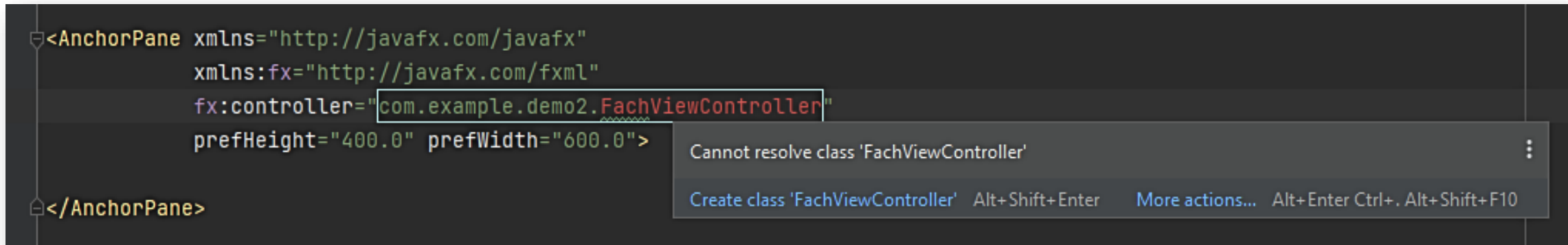
<AnchorPane xmlns="http://javafx.com/javafx"
             xmlns:fx="http://javafx.com/fxml"
             fx:controller="com.example.demo2.FachView"
             prefHeight="400.0" prefWidth="600.0">

</AnchorPane>
```

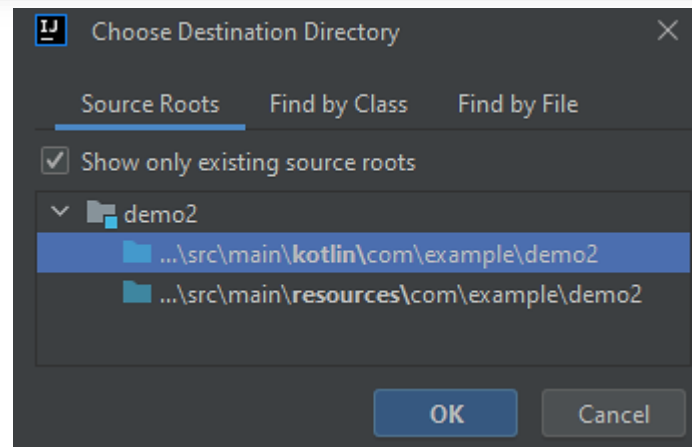

Neue View erstellen – Variante separates FXML

2. Controller zuweisen & generieren

Im Attribut `fx:controller` den Namen der zu erzeugenden (d.h. noch nicht existierenden) Controller-Klasse festlegen und diesen von der IDE automatisch generieren lassen



Code-Verzeichnis der bestehenden Klassen / Controller wählen



```
package com.example.demo2

class FachViewController {
}
```