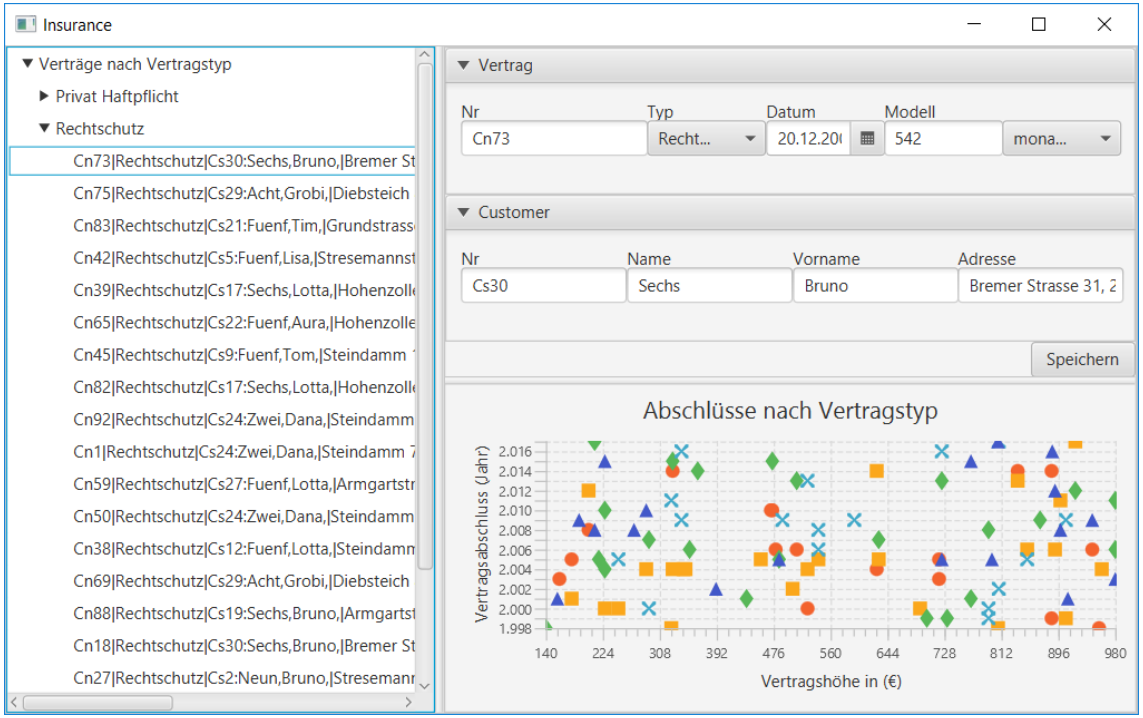
PM2 Java: GUI Entwicklung / MVC Design Pattern

EINFÜHRUNG

Graphische Benutzeroberflächen

Ein grafisches User Interfaces (GUIs) z.B. für Versicherungs-Verträge

TreeView



Label
Textfelder
Comboboxen
Textfelder für Zahleneingabe
DatePicker

ScatterChart

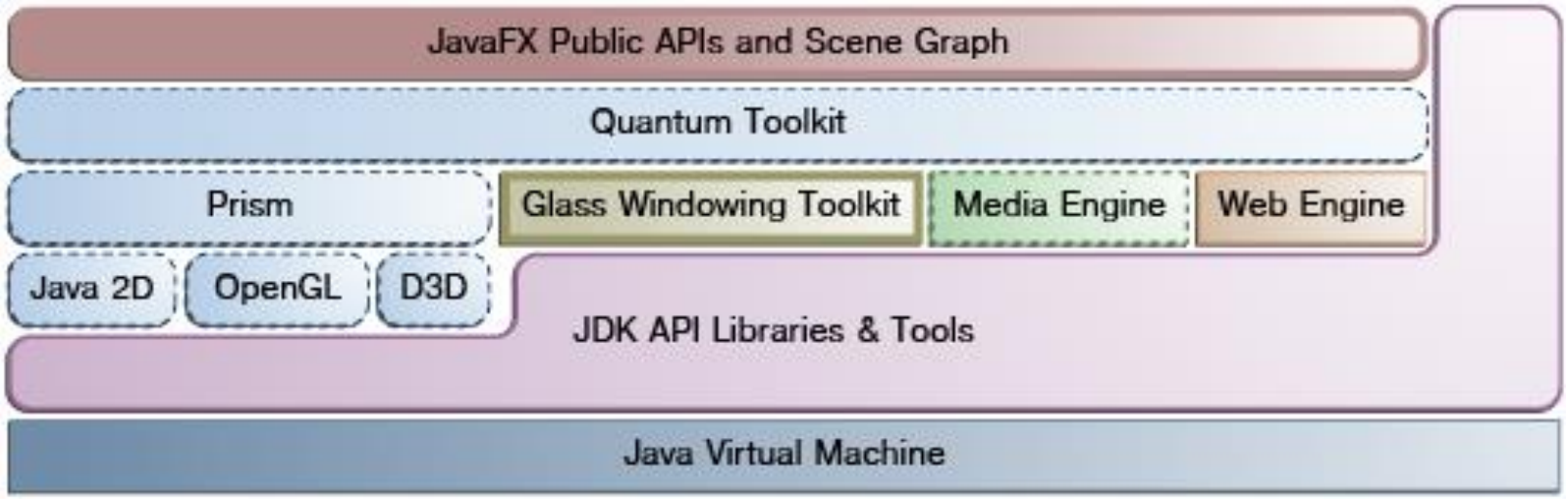
(Java) GUIs: Elemente und Funktionalitäten

- Grafische Komponenten (**UI-Controls**) für die Interaktion mit dem Benutzer: Fenster, Textfelder, Buttons, Menüs, Tabellen, Selektionslisten etc.
- Zeichnen grafischer 2D/3D Elemente, Zeichnen von UI-Controls (**Rendering**)
- Gruppierung von Elementen eines GUI's, z.B. für das Layout. (**Container**)
- Unterstützung für das Layout von Komponenten um z.B. die Größenverhältnisse bei der Skalierung zu erhalten. (**Layout-Container**)
- Unterstützung für Behandlung von Benutzerinteraktionen über verschiedene Eingabemedien wie Maus, Tastatur, Touch, Sprache, Geste etc. (**Event-Handling / Event-Dispatching**)
- Unterstützung für das Synchronisieren von Datenmodellen mit den Inhalten der GUI-Komponenten (**Model View Controller Pattern (MVC)**).

Java GUI Bibliotheken

- **AWT** (Abstract Window Toolkit)
  - seit Java 1.0, heute außer Teilen des Event-handlings nicht mehr benutzt
- **Swing**
  - Plattformunabhängige Darstellung
  - In Java geschrieben
  - nicht nur rechteckige Formen
- **JFC** (Java Foundation Classes)
  - Swing Komponentenbibliothek
  - Frei wählbares Look & Feel
  - Unterstützung für Drag&Drop.
  - Erweiterbare Klassen für Benutzerschnittstellen (sichtbare (J-Klassen) und unsichtbare Klassen (Modell und-Eventklassen))
- **JavaFX**
  - XML-basiertes Framework zur Entwicklung von GUI's ➔ FXML und SceneView (Grafisches Toolkit)
  - Darstellung der Komponenten in Baumstruktur
  - Integration mit CSS (Cascading Style Sheets)
  - Integration für Webseiten (WebView)
  - Schnittstellen zu Swing
  - Bibliotheken für UI-Komponenten wie Buttons, TreeViews etc.
  - (3D) Grafik mit Hardwarebeschleunigung
  - Eventhandling

Die JavaFX Architektur



Quelle: http://docs.oracle.com/javase/8/javafx/get-started-tutorial/img/jfxar_dt_001_arch-diag.png

Der Szenengraph - das JavaFX GUI-Modell

- ein Baum, dessen Knoten die visuellen Elemente des User Interfaces repräsentieren
- verarbeitet Benutzereingaben
- Basis für das Rendering (Aufbereiten für die Darstellung)
- Knoten haben
  - eine ID
  - eine style class (→ CSS)
  - optional:
    - Effekte (Schatten, Unschärfe,etc)
    - Tranparenz
    - Transformationen
    - Event Handler (für Maus, Tastatur und Eingabemethoden)
    - Applikations-spezifischen Zustand

Die JavaFX API‘s (Application Programming Interfaces)

- Integration mit den Java API‘s, wie z.B. Generics, Annotationen, Multithreading, Lambda Expressions
- Integration mit JavaScript und Groovy
- Properties, Bindings um Änderungen von Objekten zu propagieren
- Erweiterungen der Java Collection Klassen um observable Listen und Maps, um Datenmodelle mit GUI-Komponenten zu verbinden. (Änderungen im Modell werden automatisch in den GUI Komponenten sichtbar.)
- API’s für das Layout
- Multimedia API’s

Das Grafik-System

- Unterstützung für 2D und 3D Szenengraphen

- Subsysteme für Grafikbeschleunigung
  - **Prism:** Anzeige des Szenengraphen unter Verwendung von Hardware-oder Software-Renderern
    - DirectX 11 (Windows 7)
    - OpenGL (Mac, Linux, Embedded)
    - Software-Rendering, wenn Hardwarebeschleunigung nicht möglich

  - **Quantum Toolkit**:
    - Verbindung von Prism und Glass Windowing Toolkit für den darüber liegenden Layer
    - Threading Regeln für Rendering versus Event-Handling

Multi-Media

- Unterstützung für visuelle Medien und Audio.

- Formate:
  - Audio: MP3, AIFF, WAV
  - Video: FLV

- 3-Komponenten Modell:
  - Medien Objekt → Mediendatei
  - MediaPlayer zum Abspielen
  - MediaView zur Anzeige des Mediums

- **kein Stoff der Vorlesung**: Vertiefung im Eigenstudium → http://www.oracle.com/pls/topic/lookup?ctx=javase80&id=JFXMD

Cascading Style Sheets

- deklarative Vorgaben für das Aussehen des UI-Elemente
- dynamisch zur Laufzeit änderbar
- **kein Stoff der Vorlesung**: Vertiefung im Eigenstudium → http://docs.oracle.com/javase/8/javafx/user-interface-tutorial/css_tutorial.htm#JFXUI733

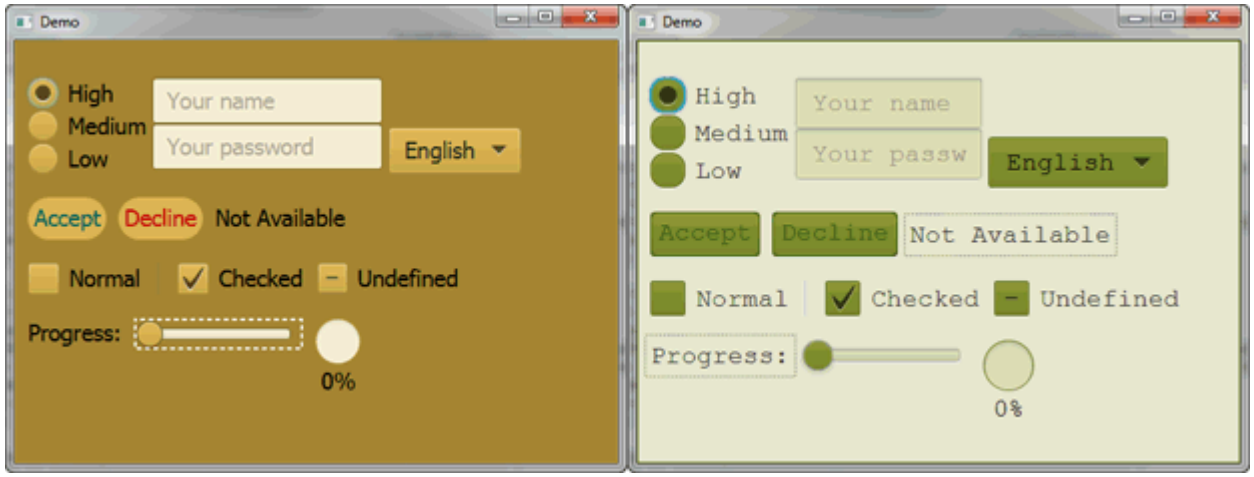


Abb.: 2 CSS Styles für die gleiche Menge an UI-Controls
Quelle: http://docs.oracle.com/javase/8/javafx/get-started-tutorial/img/css-style-sample.gif

UI-Komponenten (UI-Controls)

Abb.: Eine Auswahl an UI-Controls in JavaFX
Quelle: http://docs.oracle.com/javase/8/javafx/get-started-tutorial/img/uicontrols.png

Layout Container (1)

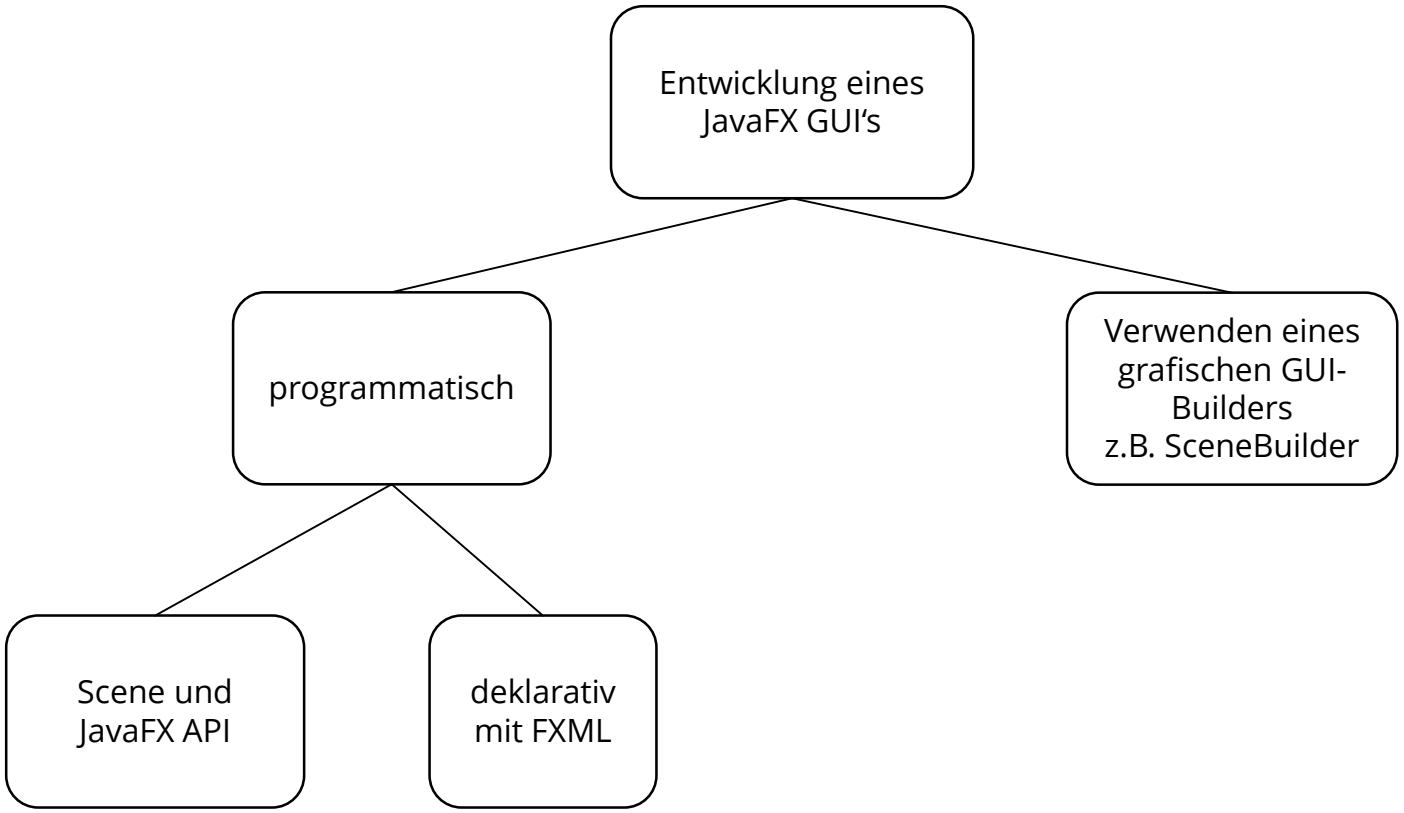
- absolute Koordinaten und feste Größen für UI Elemente führen beim Skalieren des enthaltenen Fensters zu unerwünschten visuellen Effekten.
- Layout Container übernehmen die Anordnung von UI Elementen eines Szenengraphen und erhalten bei Skalierungen die Proportionen und relativen Positionen der Elemente zueinander.
- Layout Container erlauben eine flexible und dynamische Anordnung von UI-Controls eines Szenengraphen
- Container können ineinander geschachtelt werden ➔ Kombination verschiedener Layouts für einen Szenegraphen möglich

Layout Container (2)

- Automatisiertes Layout für die gängigen Layout-Modelle:
  - **`BorderPane`**: Anordnung in den Regionen `top`, `bottom`, `right`, `left`, `center`
  - **`Hbox`**: horizontale Anordnung in einer Reihe
  - **`Vbox`**: vertikale Anordnung in einer Reihe
  - **`StackPane`**: Anordnung in einem „back-to-front“ Stapel.
  - **`GridPane`**: Anordnung in einem flexiblen Raster von Zeilen und Spalten
  - **`FlowPane`**: horizontale / vertikale Anordnung mit der Option Breite und Höhe festzulegen.
  - **`TilePane`**: Anordnung in einheitlich großen Kacheln
  - **`AnchorPane`**: Erzeugen von Anchor-Nodes für die Regionen `top, bottom, left, center`

VARIANTEN DER ENTWICKLUNG

Varianten des Entwicklungsprozesses



Vorbereitung

- Installation des JavaFX SceneBuilder Version 2. Der SceneBuilder ist das grafische Tool zum Erstellen von GUI's mit JavaFX. Download Link: http://gluonhq.com/labs/scene-builder/#download

Scene und JavaFX API‘s

- In IntelliJ:

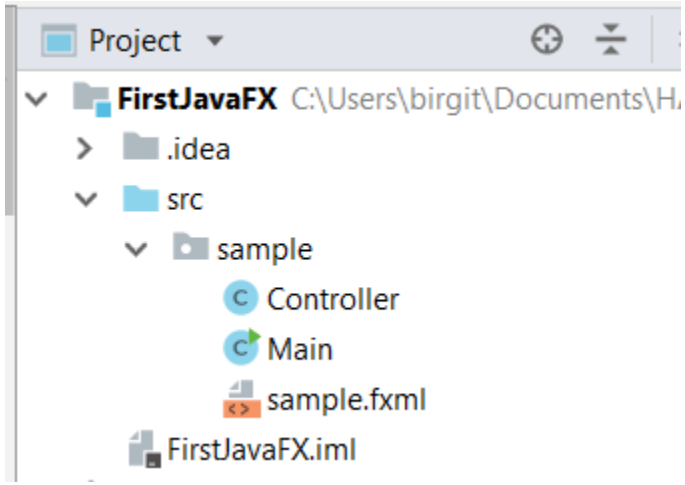
  `File→Project → JavaFX`

- erzeugt die neben stehenden Projektstruktur

- `Main.java` ist die Klasse, über die das JavaFX Programm gestartet wird.

- `Controller.java` ist die Klasse, die das Event-Handling für die UI Controls implementiert.

- `sample.fxml` enthält den UI-Aufbau



Aufbau von Main.java

Die Hauptklasse einer JavaFX Anwendung muss von Application ableiten

zentraler Einstiegspunkt

LayoutContainer

Scene ist der Container für den Inhalt des UIF mit Wurzelknoten root.

Darstellung des GUI's.

main wird benötigt, wenn die JavaFX Applikation direkt aus IntelliJ startbar sein soll.

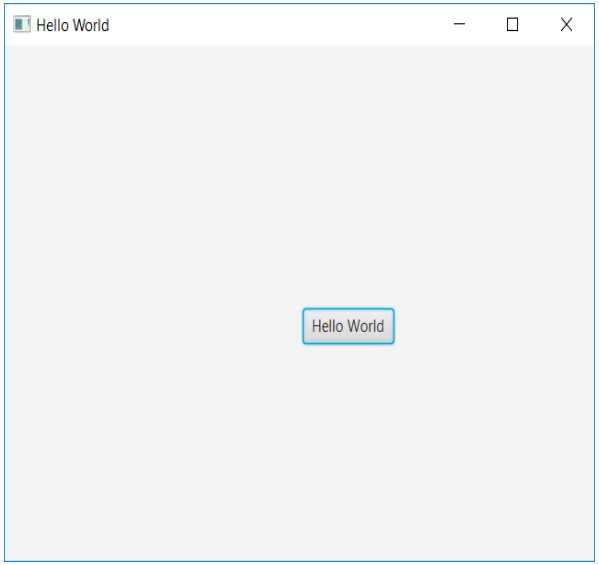
```
...

public class Main extends Application {

    @Override
    public void start(Stage primaryStage) throws Exception{
        Parent root = FXMLLoader.load(getClass().getResource("sample.fxml"));
        primaryStage.setTitle("Hello World");
        primaryStage.setScene(new Scene(root, 300, 275));
        primaryStage.show();
    }


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

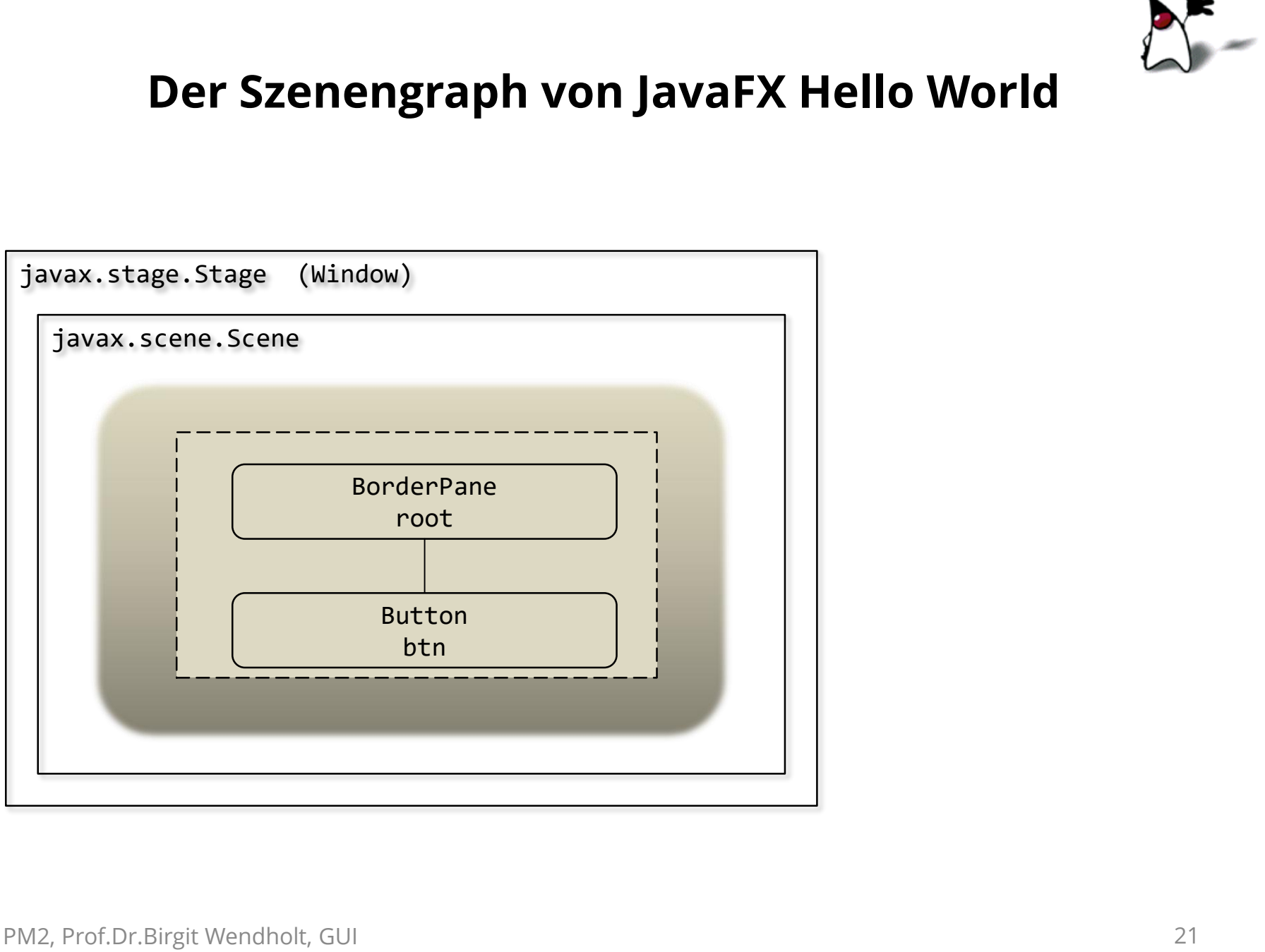
Hinzufügen eines Controls

```
10 public class HelloWorld extends Application {
11     @Override
12     public void start(Stage primaryStage) {
13         try {
14             Button btn = new Button("Hello World");
15             BorderPane root = new BorderPane();
16             root.setCenter(btn);
17             Scene scene = new Scene(root,400,400);
18             scene.getStylesheets().add(getClass().
19                     getResource("application.css").toExternalForm());
20             primaryStage.setScene(scene);
21             primaryStage.show();
22         } catch(Exception e) {
23             e.printStackTrace();
24         }
25     }
26
27     public static void main(String[] args) {
28         launch(args);
29     }
30 }
```

Das Control ist ein Button aus dem Package javafx.scene.control

btn wird als child in root an der Position „center" eingehängt.





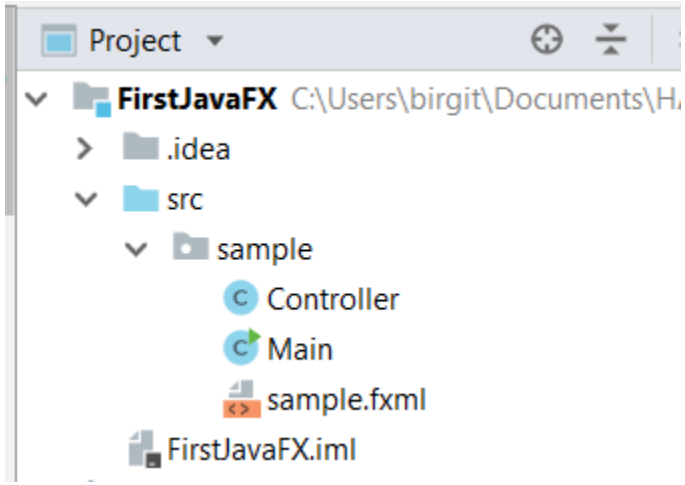
Der Szenengraph von JavaFX Hello World

Scene und JavaFX API's

- In IntelliJ:

  `File→Project → JavaFX`

- erzeugt die neben stehenden Projektstruktur



- `Main.java` ist die Klasse, über die das JavaFX Programm gestartet wird.
- `Controller.java` ist die Klasse, die das Event-Handling für die UI Controls implementiert.
- `sample.fxml` enthält den UI-Aufbau

Aufbau von Main.java

Die Hauptklasse einer JavaFX Anwendung muss von Application ableiten

Stage ist der Container für das UIF und muss die Scene enthalten

zentraler Einstiegspunkt

sample.xml enthält die GUI

```
...

public class Main extends Application {

    @Override
    public void start(Stage primaryStage) throws Exception{
        Parent root = FXMLLoader.load(getClass().getResource("sample.fxml"));
        primaryStage.setTitle("Hello World");
        primaryStage.setScene(new Scene(root, 300, 275));
        primaryStage.show();
    }


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

Scene ist der Container für den Inhalt des UIF mit Wurzelknoten root.

Darstellung des GUI's.

main wird benötigt, wenn die JavaFX Applikation direkt aus IntelliJ startbar sein soll.

Der Aufbau der `sample.fxml` der Ressource für den Szenengraph

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

        <?import javafx.scene.control.Button?>
        <?import javafx.scene.layout.BorderPane?>

<BorderPane maxHeight="-Infinity" maxWidth="-Infinity" minHeight="-Infinity"
            minWidth="-Infinity" prefHeight="400.0" prefWidth="600.0"
            xmlns="http://javafx.com/javafx/8.0.171"
            xmlns:fx="http://javafx.com/fxml/1"
            fx:controller="sample.Controller">

</BorderPane>
```

XML Header mit Version und Character Encoding.

spezielle Import Syntax für alle in der fxml verwendeten Javaklassen.

Namespace (xmlns) definiert die zulässigen XML-Elemente und Attribute, z.B. den fx:controller

Der für den Szenengraphen definierte Controller

Typ des Rootknotens des Szenengraphen

Der Aufbau der `sample.fxml` der Ressource für den Szenengraph

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

        <?import javafx.scene.control.Button?>
        <?import javafx.scene.layout.BorderPane?>


<BorderPane maxHeight="-Infinity" maxWidth="-Infinity" minHeight="-Infinity"
            minWidth="-Infinity" prefHeight="400.0" prefWidth="600.0"
            xmlns="http://javafx.com/javafx/8.0.171"
            xmlns:fx="http://javafx.com/fxml/1"
            fx:controller="sample.Controller">
<center>
   <Button mnemonicParsing="false" text="Hello World"
            BorderPane.alignment="CENTER" />
</center>
</BorderPane>
```

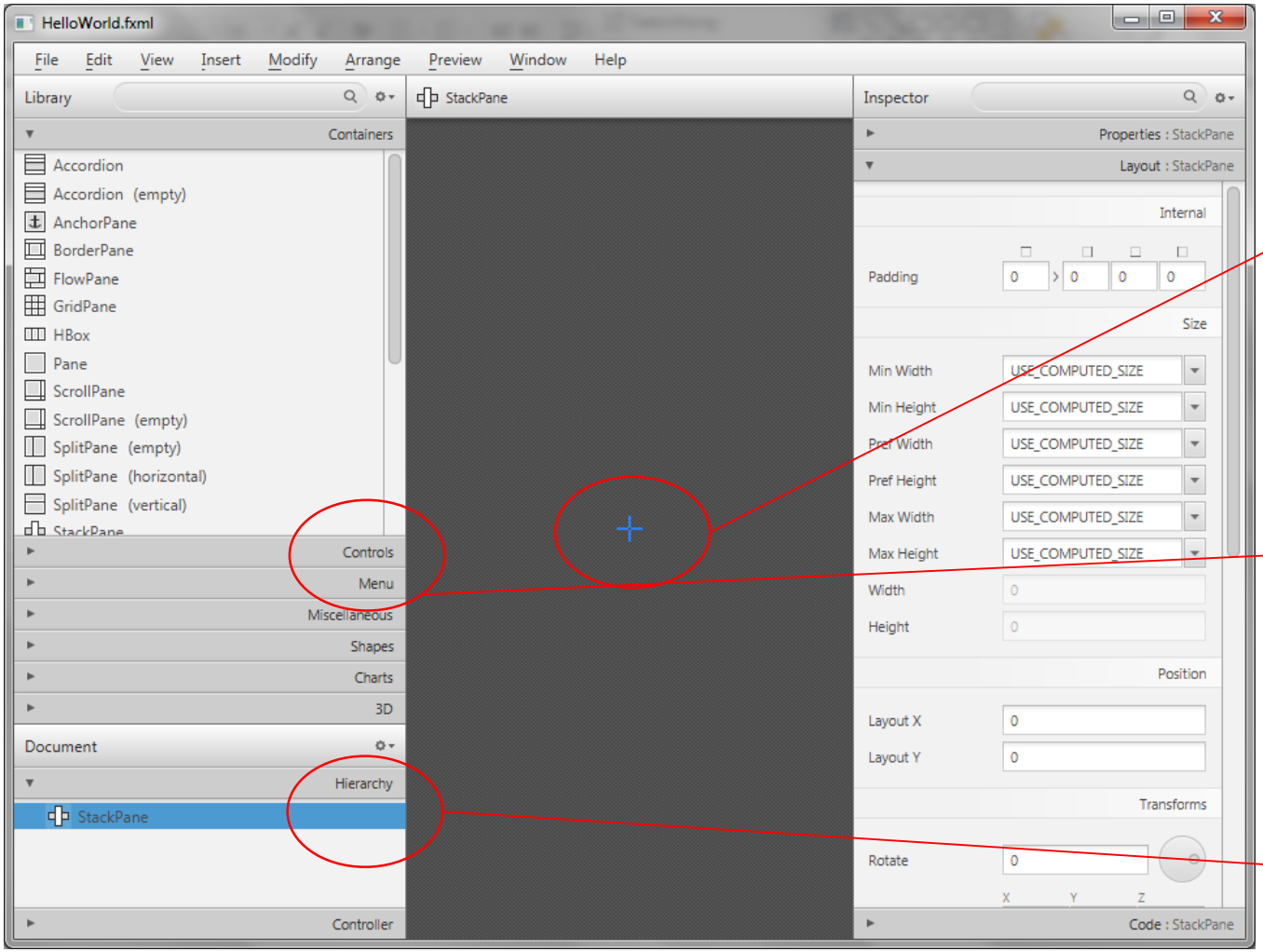
Einfügen eines Child-Nodes mit Typ Button und Text „Hello World“

Die Main-Klasse bleibt unverändert.
Die Spezifikation des Szenengraphen ist vollständig in der fxml Datei enthalten.

GUI‘s mit SceneBuilder

- In IntelliJ ein FXML Projekt erzeugen wie beim deklarativen Vorgehen.
- `sample.fxml` selektieren.
- Rechtsklick und dann im Kontextmenü
  `Open with SceneBuilder`
  auswählen.
- `SceneBuilder` startet nach ein paar Sekunden

GUI's mit SceneBuilder



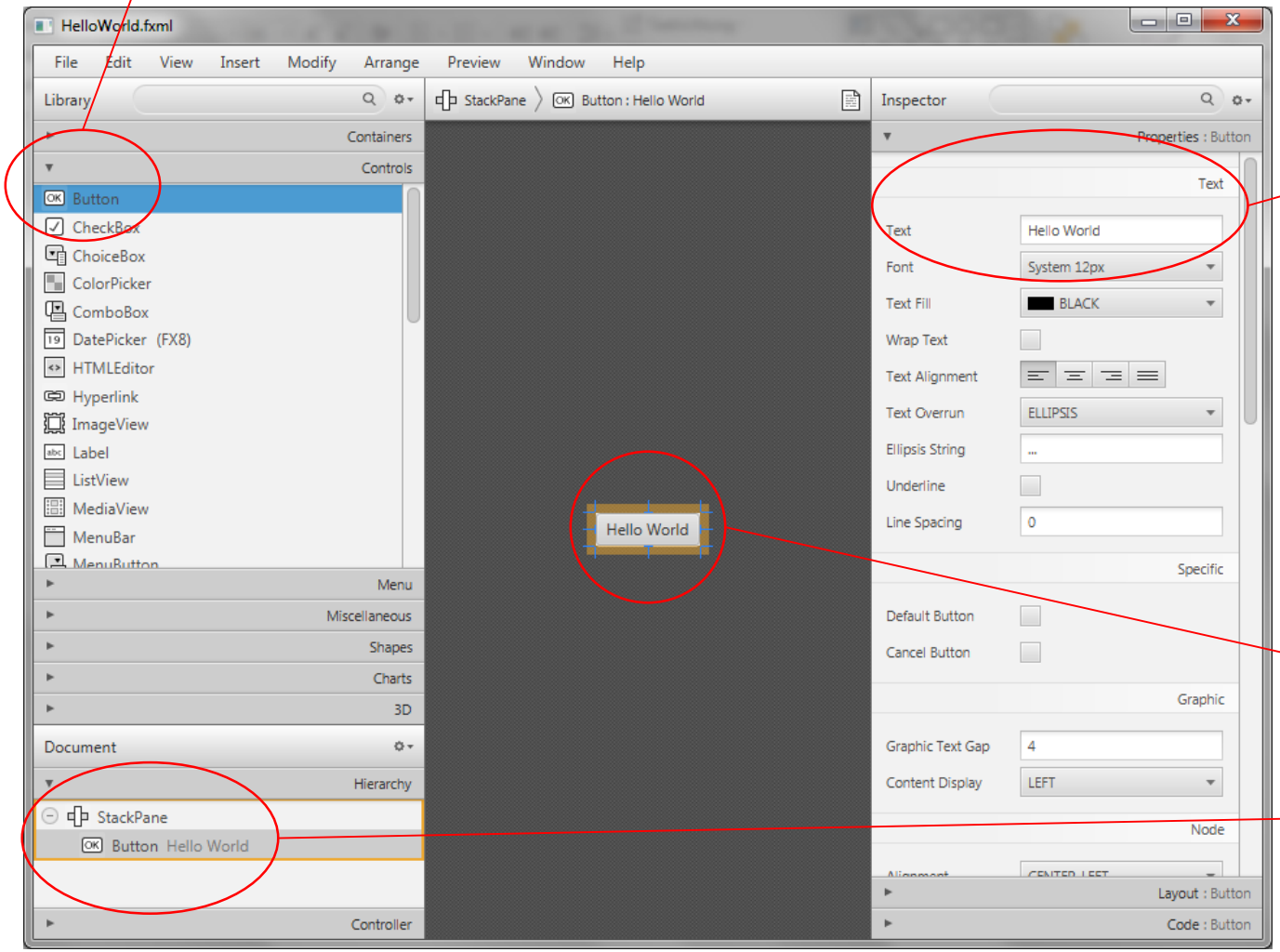
StackPane mit Höhe und Breite =0.
Pref Width und Pref Height auf z.B. 400 setzen,

Bereich/Menü mit Controls.

Initialer Szenengraph mit Rootnode StackPane

GUI's mit SceneBuilder

Im Menü Controls Button auswählen.



In Button Properties den Text für den Button ändern.

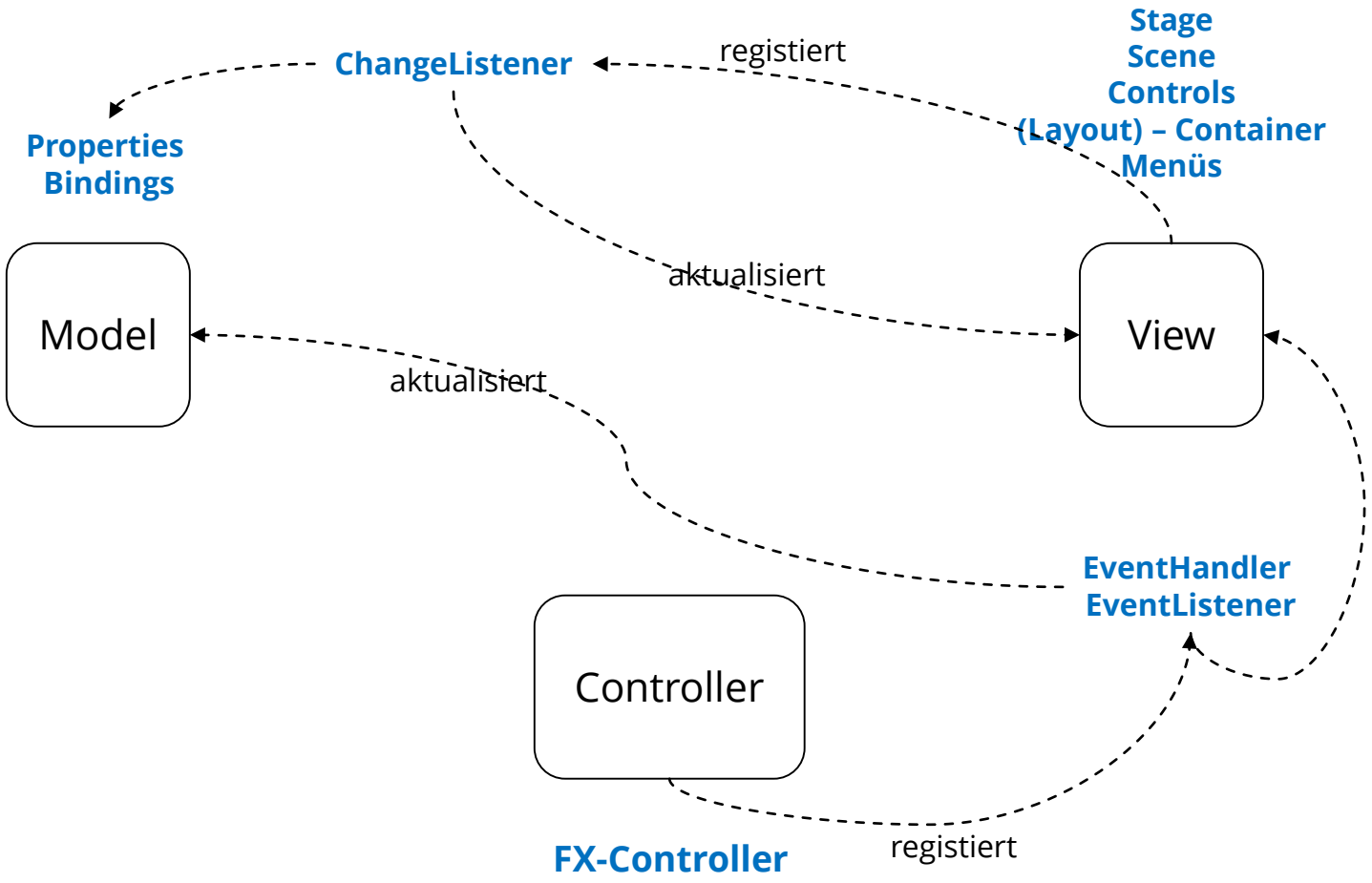
Mit Return bestätigen.

Button auf die StackPane ziehen.

Macht Button zum Child-Node von StackPane.

ELEMENT EINES JAVA FX GUIS

Model-View-Controller in JavaFX
- Elemente eines JavaFX GUI's -



View - Elemente eines JavaFX GUI‘s

- **Stage:**
  - umgebendes Fenster des GUI‘s für Anzeige, Resize, Ikonifizieren und Schließen
  - wird in der `start` Methode einer JavaFX `Application` als Parameter übergeben
- **Scene**:
  - ein Graph, der die View-Elemente des GUI‘s in einer Hierarchie von Parent und Child-Nodes verwaltet
- **Menüs:**
  - **Fenstermenüs** in der Menüleiste
  - **Kontextmenüs** für GUI-Elemente, die über die rechte Maustaste aktiviert werden.
- **Controls**: einzelne GUI-Elemente
  - Textfelder, Buttons, Comboboxen, Texteditoren, etc.
- **Container**:
  - Behälter für GUI-Elemente
  - **ohne Layouter** Group / Pane oder **Layout-Container**

Model- und Controller- Elemente eines JavaFX GUI's

- **Properties und Bindings (Schnittstelle zum Applikationsmodell)**:
  - **Properties**: beobachtbare Objektwerte, die „aktiv" Änderungen an registrierte Komponenten (z.B. View-Controls) mitteilen
  - **Bindings**: stellen Abhängigkeiten zwischen Properties und z.B. Inhalten von Controls her. ➔ Änderungen werden in allen abhängigen Controls sichtbar.

- **EventHandler / EventListener (Events, Event-Typen) die Controller des MVC:**
  - behandeln Ereignisse, die durch Benutzeraktionen oder Ändern von Objektwerten ausgelöst werden.
  - können auf Properties des Modell registriert werden und werden bei Änderungen getriggert.
  - werden aktiviert, wenn das Ereignis, für das sie registriert sind, auftritt

- **Controller in JavaFX die Schnittstelle zwischen View und Model**:
  - registriert EventHandler und EventListener mit deren Implementierungen
  - ändert dadurch indirekt die Inhalte der Modell-Elemente

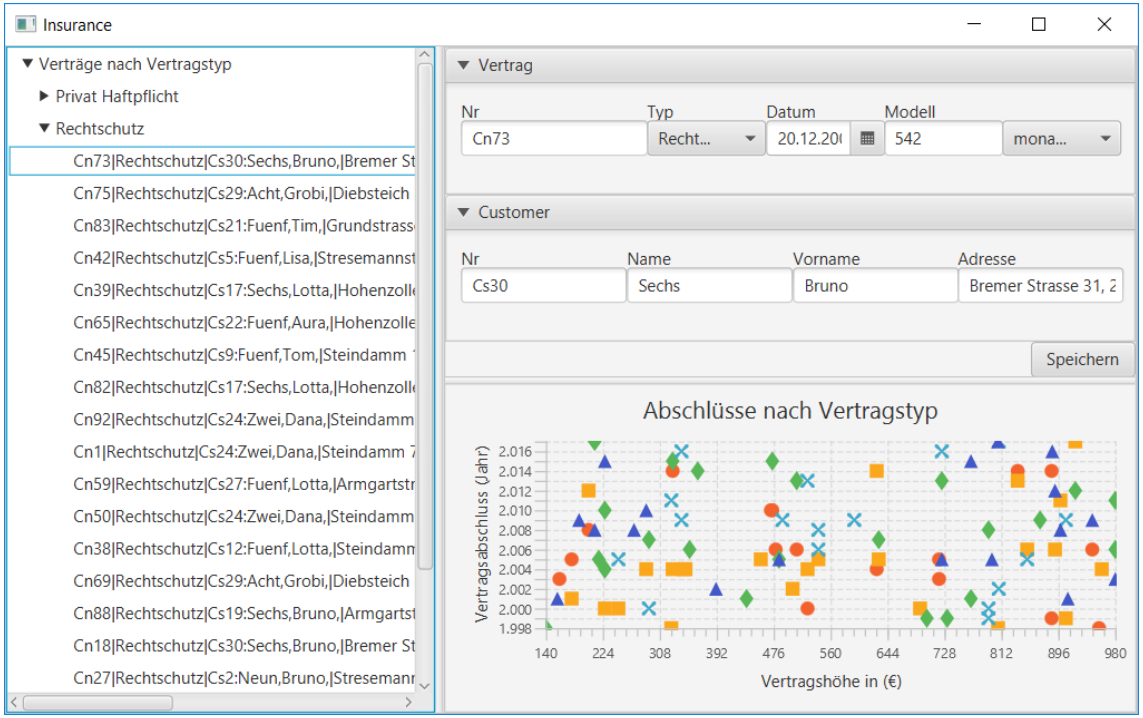
Umsetzung des Versicherungs-GUIs

BEISPIEL

Aufbau

ContractTreeView

ContractDetailView

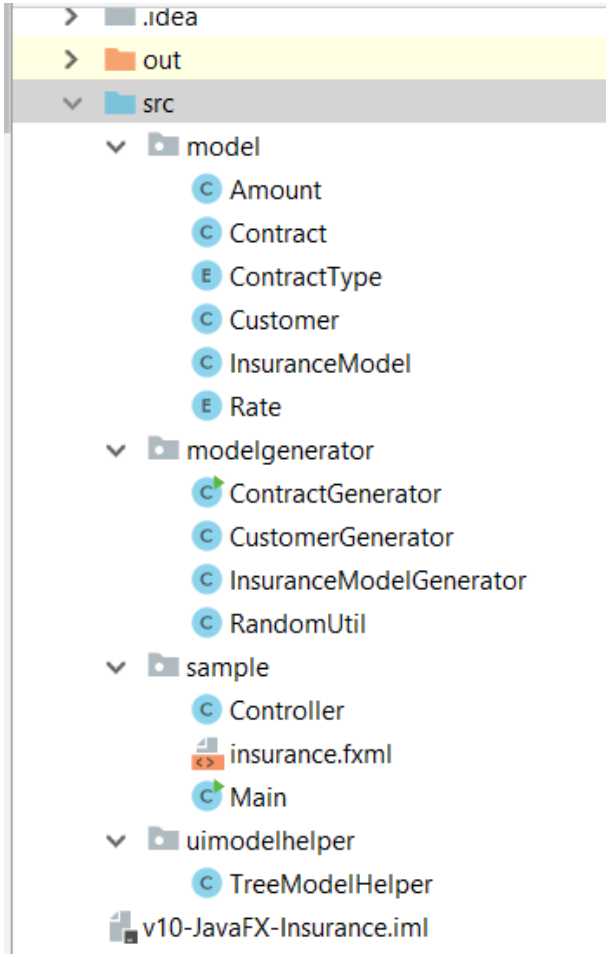


ContractScatterView

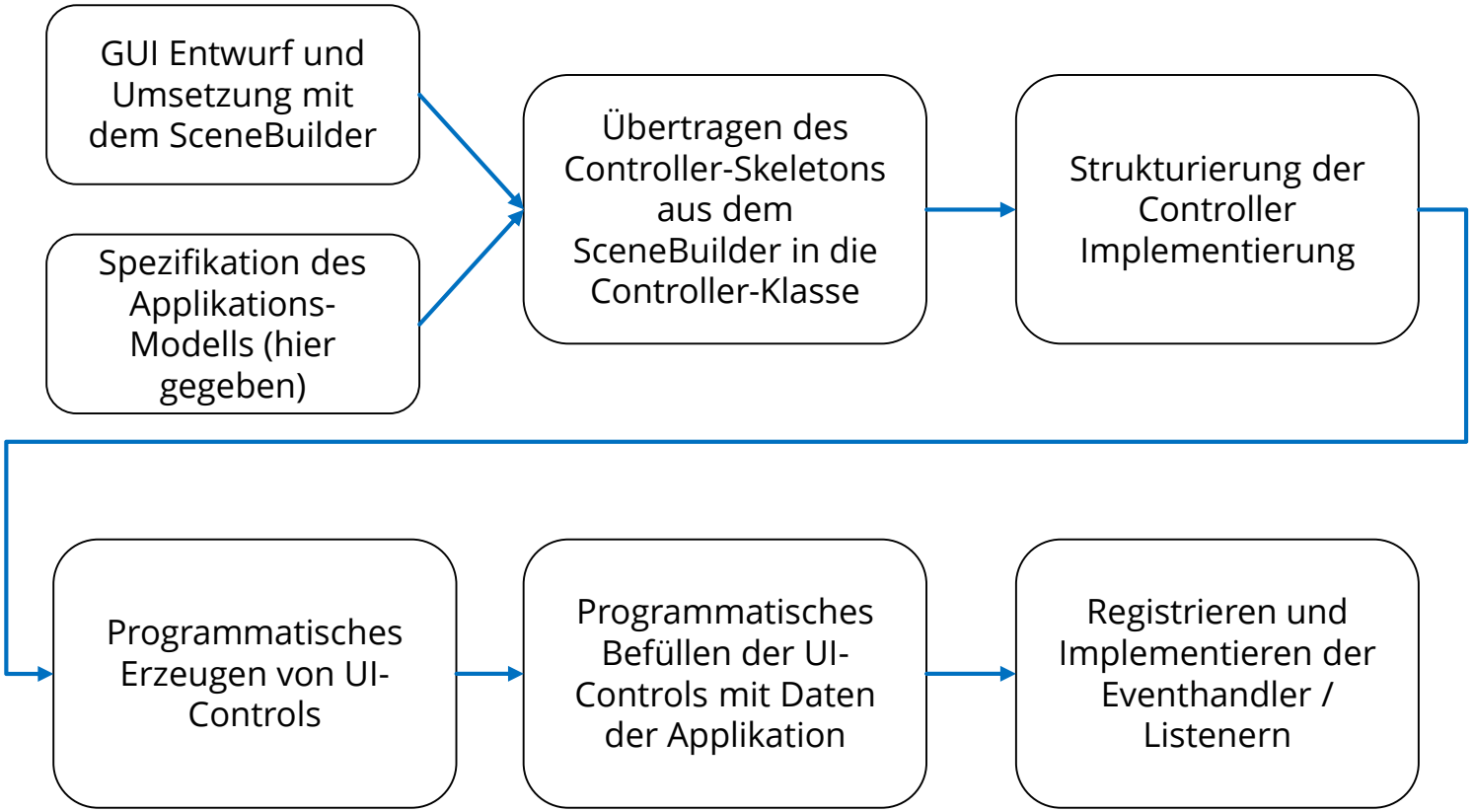
Die 3 Views sind innere Klassen der Klasse ***Controller*** des Beispiels in **V10-JavaFX-Insurance**

Struktur des JavaFX Projektes

- `model`: enthält die Klassen des Applikationsmodells.
- `modelgenerator`: Generiert Verträge mit Hilfe eines Zufallsgenerators
- `sample`: enthält die JavaFX Klassen `Main, Controller` und die `sample.fxml`, die den GUI Aufbau enthält
- `uimodelhelper`: enthält eine Hilfsklasse, die die Baumstruktur für die `TreeView` erzeugt. Kann eine hierarchische Struktur gruppiert nach Vertragstyp und Kunde aufbereiten.

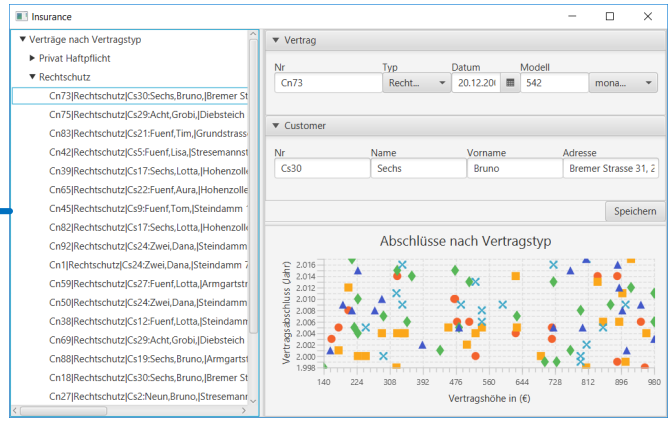


Schritte bei der Entwicklung einer JavaFX Applikation

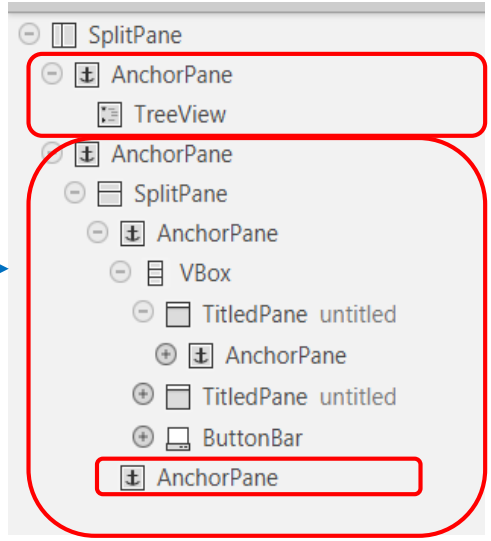


GUI ENTWURF UND UMSETZUNG

GUI-Struktur im SceneBuilder



Überführung



SceneBuilder GUI = ein Baum aus Containern und Controls

Root der Scene: vertikale Splitpane

linke Seite: AnchorPane mit der TreeView

rechte Seite: AnchorPane mit horizontalem Splitpane

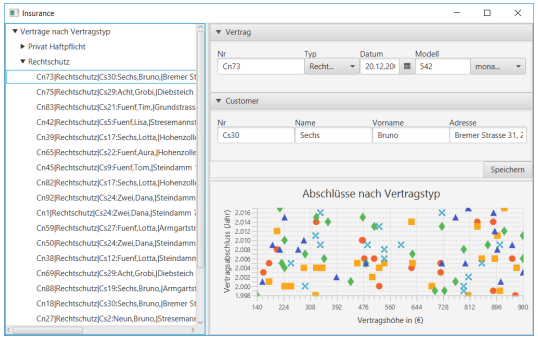
oberer Bereich: AnchorPane für VBox mir zwei TitledPanes für die Vertragsdetails und einer ButtonBox für das Speichern

unterer Bereich: AnchorPane in das programmatisch das ScatterChart eingefügt werden muss

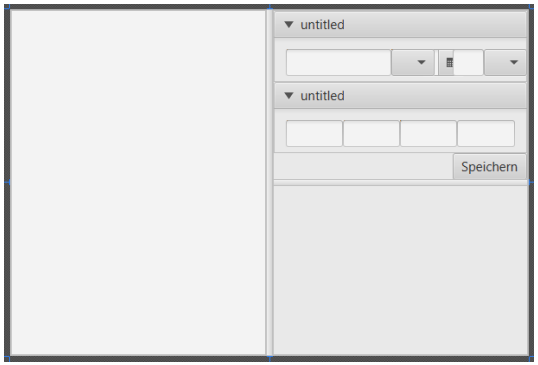
Kategorien von UI-Elementen im Beispiel

- **Container**:
  - → `SplitPane`: Zerlegen in 2 Bereiche mit einem Handle zum dynamischen Einstellen der Größe der 2 Bereiche
  - → `GridPane`: Anordnen der Komponenten in einem Raster
  - → `AnchorPane`: ermöglicht da einfache Verankern der enthaltenen UI-Komponenten an allen Seiten des Containers
  - → `VBox`: vertikale Anordnung von UI-Elemente
  - → `TitledPane`: Container mit Titel
  - → `ButtonBox`: Container für Buttons

- **Controls**:
  - – `Button, Label, TreeView, TextField, ComboBox, DatePicker, ScatterChart`



Fertiges GUI mit Daten des Modells
und programmatisch erzeugtem ScatterChart

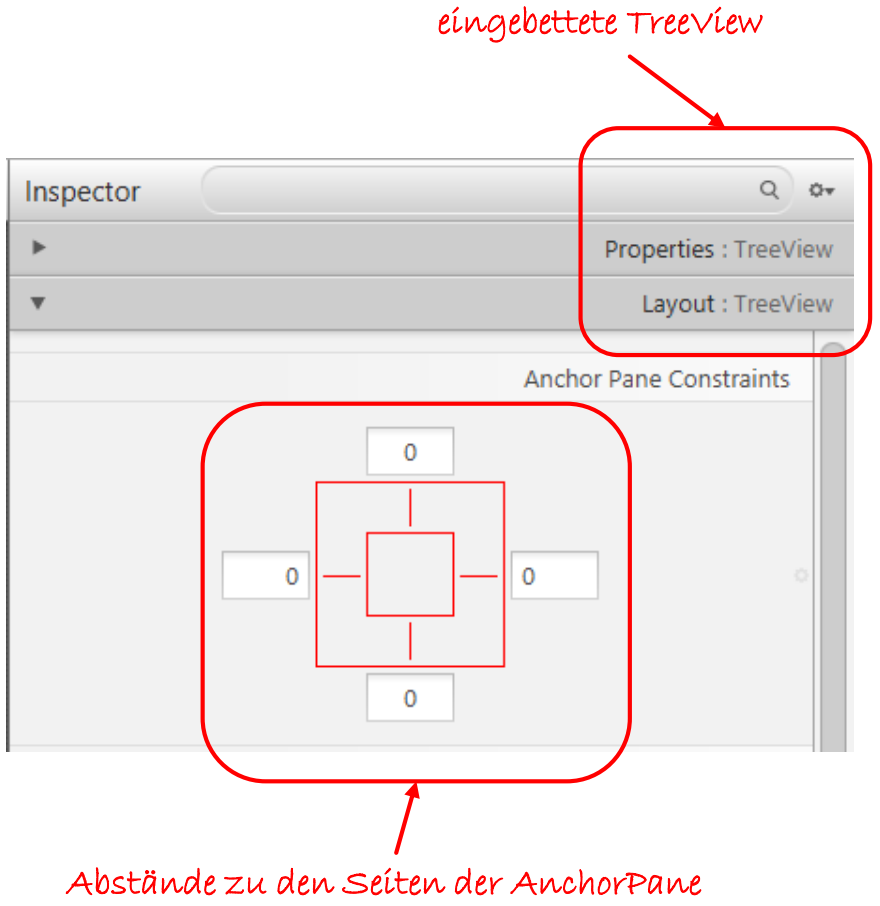


GUI in SceneBuilder vor Befüllen der UI-Controls
mit Daten und ohne ScatterChart

Verwendung von AnchorPane

- AnchorPane:
  - → Container, an dessen Seiten sich die enthaltenen Elemente anheften lassen.
  - → Dabei kann eine Abstand zu den jeweiligen Seiten im SceneBuilder eingestellt werden.
  - → Die Abstände werden immer in dem enthaltenen UI-Element eingestellt. Im Beispiel in der Layout-Sicht: die `TreeView`, die in das `AnchorPane` eingebettet ist.

- **Vorteil**:
  - → Bei Vergrößerung des `AnchorPane` bleiben die enthaltenen Komponenten an den Seiten **kleben** und es entsteht keine leere Fläche in der Hintergrundfarbe.



Verwendung von Containern

VBox

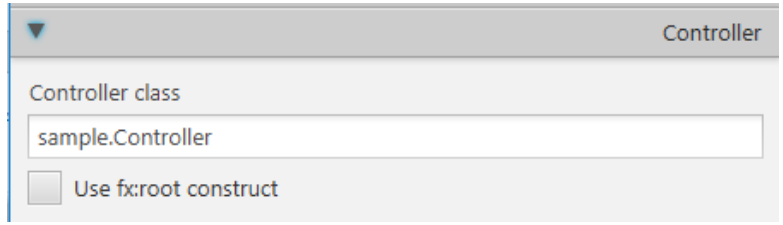
- wenn Elemente nebeneinander angeordnet werden sollen
- wenn Elemente immer gleich viel Platz einnehmen sollen
- wenn der gesamte Bereich des enthaltenen Containers ausgefüllt werden soll

GridPane

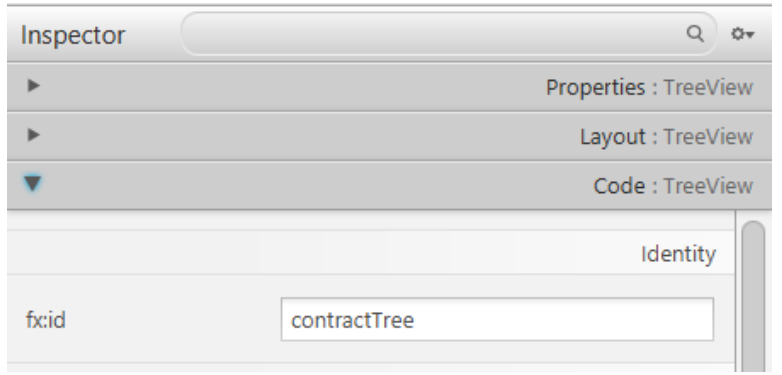
- wenn Elemente in einem flexiblen Raster angeordnet werden sollen
- wenn sich die Zellen des Rasters an die Größe der Elemente anpassen sollen
- wenn Elemente auch mehrere Spalten und Zeilen überdecken müssen (`rowspan` / `colspan` > 1)

Übertragen des Controller-Skeletons

- **Vorbereitung**:
  1. Eintragen der Controller-Klasse (voll qualifizierter Name) im SceneBuilder



  2. Vergabe von id's für Container und Controls, die später im Programm referenziert werden müssen, im **Code**-Bereich des **Inspektors** im SceneBuilder



- `View → Show Sample Controller Skeleton`:
  - → generiert zu jedem Container / Control eine mit @FXML annotierte Instanz-Variable der Controller-Klasse.
  - → muss in die Controller-Klasse in IntelliJ kopiert werden,

```
public class Controller {

    @FXML
    private TreeView<?> contractTree;

    @FXML
    private TitledPane contractTitledPane;
```

Aufbau des Controllers (Ausschnitt)

- Für jedes UI-Element mit `fx:id` wird eine Instanz-Variable, die mit `@FXML` annotiert ist, generiert.
- `@FXML` bewirkt, dass die UI-Elemente aus der fxml-Datei erzeugt werden und den Instanz-Variablen zugewiesen werden.
- `@FXML protected void initialize`: wird von JavaFX auf dem Controller-Objekt aufgerufen, nachdem alle `@FXML` Instanz-Variablen initialisiert sind.
- private innere Klassen
  - → `ContractTreeView`
  - → `ContractScatterView`
  - → `ContractDetailView`

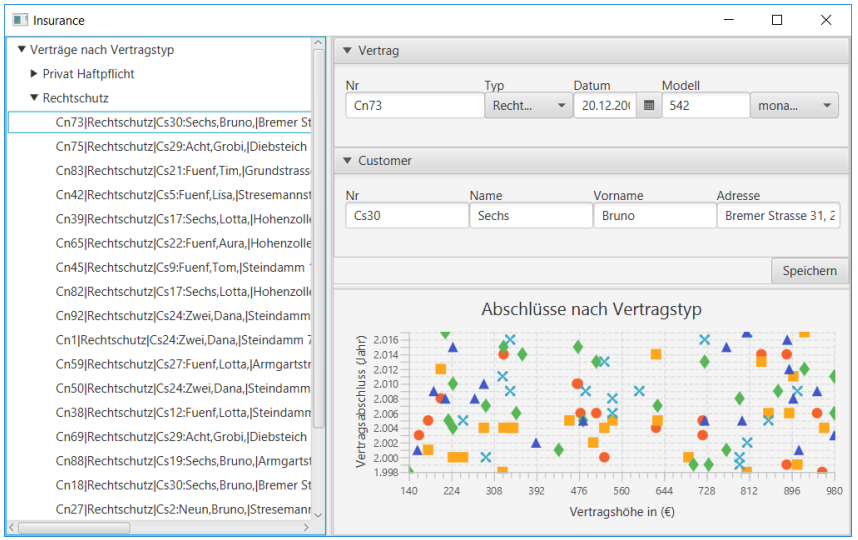
```
    @FXML
    private TextField customerSurName;

    @FXML
    private TextField customerFirstName;

    @FXML
    private TextField customerAddress;

    @FXML
    private Button saveButton;

    @FXML
    private AnchorPane scatterParent;

    @FXML
    protected void initialize() {...}


    private class ContractTreeView {...}

    private class ContractScatterView {...}


    private class ContractDetailView {...}

}
```

STRUKTURIERUNG DER CONTROLLER IMPLEMENTIERUNG

Strukturieren der Controller-Implementierung

- 3 große Bereiche der GUI:
  - → `ContractTreeView`:
    - Aufbau der Baumdarstellung für Verträge
    - Eventhandler für die Selektion von Elementen im `TreeView`
    - Registrierung eines Kontextmenüs für die Elemente einer TreeView
  - → `ContractDetailView`:
    - Darstellung der Details eines Vertrages mit unterschiedlichen Controls
    - Spezielle Textfelder für die kontrollierte Eingabe von Zahlen
    - Eventhandler für das Speichern von Änderungen
  - → `ContractScatterView`:
    - Darstellung eines Scatterplots für die Größen Abschlussdatum und Vertragshöhe nach Vertragskategorie

ContractTreeView **ContractDetailView**



ContractScatterView

Verwendung der inneren Klassen im Controller

Im initialize des Controllers werden nur die Daten der Applikation generiert und Objekte der inneren Klasse erzeugt. Die eigentliche Logik steckt in der Implementierung der inneren Klassen

```
@FXML
protected void initialize() {

    final List<Contract> generatedContracts =
          ContractGenerator.generate(MAX_CONTRACTS);

    contractTreeView = new
          ContractTreeView(generatedContracts);
    contractTreeView.initialize();

    contractDetail = new ContractDetailView();

    contractScatterView = new
           ContractScatterView(generatedContracts);
}
```

Erzeugen der Daten für die Applikation

Erzeugen und Initialisieren der ContractTreeView

Erzeugen der ContractDetailView

Erzeugen der ContractScatterView

ContractScatterView

PROGRAMMTISCHES ERZEUGEN VON UI-CONTROLS

Programmtisches Erzeugen des ScatterChart

- Ein ScatterChart ermöglicht die Darstellung 3‘er Größen in einem 2-dimensionalen Koordinatensystem.
- Im Beispiel:
  - → Größe 1 x-Achse: Höhe des Vertragsabschlusses
  - → Größe 2 y-Achse: Jahr des Vertragsabschlusses
  - → Größe 3 Kategorien unterhalb der x-Achse: Vertragstyp
- Da die Werte der drei Größen vom Datenbestand abhängen,
  - → muss der ScatterChart zur Laufzeit im Controller erzeugt werden und
  - → dann dem Parent-Container hinzugefügt werden.

```
int minX = minAmount().getEuro();
int maxX = maxAmount().getEuro();
int tickUnitX = (maxX - minX) / 10;


// Achsen des ScatterCharts
final Axis xAxis = new NumberAxis(minX, maxX, tickUnitX);
final Axis yAxis = new NumberAxis(minY, maxY, tickUnitY);

// Beschriftung des Achsen
xAxis.setLabel(VERTRAGS_HOEHE);
yAxis.setLabel(VERTRAGS_ABSCHLUSS);

// Erzeugen des ScatterCharts mit den Achsen
ScatterChart<Integer, Integer> contractScatter =
            new ScatterChart<>(xAxis, yAxis);
contractScatter.setTitle(ABSCHLUESSE_NACH_VERTRAGS_TYP);

// programmatisches Hinzufügen des ScatterCharts zu
dem ParentContainer scatterParent (Typ AnchorPane)
scatterParent.getChildren().add(contractScatter);
// Verankern des ScatterCharts in der AnchorPane
AnchorPane.setTopAnchor(contractScatter, 1.0);
AnchorPane.setLeftAnchor(contractScatter, 1.0);
AnchorPane.setRightAnchor(contractScatter, 1.0);
AnchorPane.setBottomAnchor(contractScatter, 1.0);
```

Erläuterung der `initialize` Methode der `ContractScatterView`

```
private void initialize() {

    ... Bestimmen von minX / maxX, minY / maxY ...

    // Achsen des ScatterCharts
    final Axis xAxis = new NumberAxis(minX, maxX, tickUnitX);
    final Axis yAxis = new NumberAxis(minY, maxY, tickUnitY);

    // Beschriftung des Achsen
    xAxis.setLabel(VERTRAGS_HOEHE);
    yAxis.setLabel(VERTRAGS_ABSCHLUSS);

    // Erzeugen des ScatterCharts mit den Achsen
    ScatterChart<Integer, Integer> contractScatter =
                        new ScatterChart<>(xAxis, yAxis);
    contractScatter.setTitle(ABSCHLUESSE_NACH_VERTRAGS_TYP);

    // programmatisches Hinzufügen des ScatterCharts zu dem ParentContainer scatterParent (Typ AnchorPane)
    scatterParent.getChildren().add(contractScatter);
    // Verankern des ScatterCharts in der AnchorPane
    AnchorPane.setTopAnchor(contractScatter, 1.0);
    AnchorPane.setLeftAnchor(contractScatter, 1.0);
    AnchorPane.setRightAnchor(contractScatter, 1.0);
    AnchorPane.setBottomAnchor(contractScatter, 1.0);
... Hier geht es noch weiter mit dem Befüllen des ScatterChart mit Werten
```

minimale und maximale Werte für die x-/y-Achse (NumberAxis)

Erzeugen der x-/y- Achsen für das ScatterChart

Beschriftung der x-/y-Achsen

Erzeugen des ScatterChart

Hinzufügen zum ParentContainer. Dazu muss dieser Container im SceneBuilder eine id erhalten.

Programmatisches Anheften an die Parent-AnchorPane

Berechnen der Minima und Maxima für die Achsen

```
// Min Max Werte für die Achse bestimmen
int minX = minAmount().getEuro();
int maxX = maxAmount().getEuro();
int tickUnitX = (maxX - minX) / 10;
```

minAmount / maxAmount der Klasse ContractScatterView bestimmen die minimale und maximale Höhe aller Verträge.
tickUnitX bestimmt den Abstand der X-Achseneinheiten

```
// Min Max Werte für die Y-Achse bestimmen
int minY = minDate().getYear();
int maxY = maxDate().getYear();
int tickUnitY = 1;
```

minDate / maxDate berechnen das minimale / maximale Abschlußdatum aller Verträge. Der Typ ist ein LocalDate. LocalDate ist eine Klasse des Java 8 Time-API und ist Locale-sensitive. D.h. berücksichtigt Sprach- und Landestypische Eigenschaften von Datums-Objekten, insbesondere auch Zeitzonen.
Für diesen Typ ist getYear() eine gültige Methode.

Programmatisches Hinzufügen zum Parent

- Der JavaFX Szenen-Graph ist eine Hierarchie aus Containern und Controls
- Ein Container kann >1 Kindknoten enthalten.
- Kindknoten sind entweder Container oder Controls.
- Zugriff auf die Kindknoten mit `getChildren().`
- Hinzufügen von Elementen über `getChildren().add(...)` u.ä. Methoden.
- Dazu wird im Programm eine Referenz auf den Parent benötigt. Im Beispiel: `scatterParent`

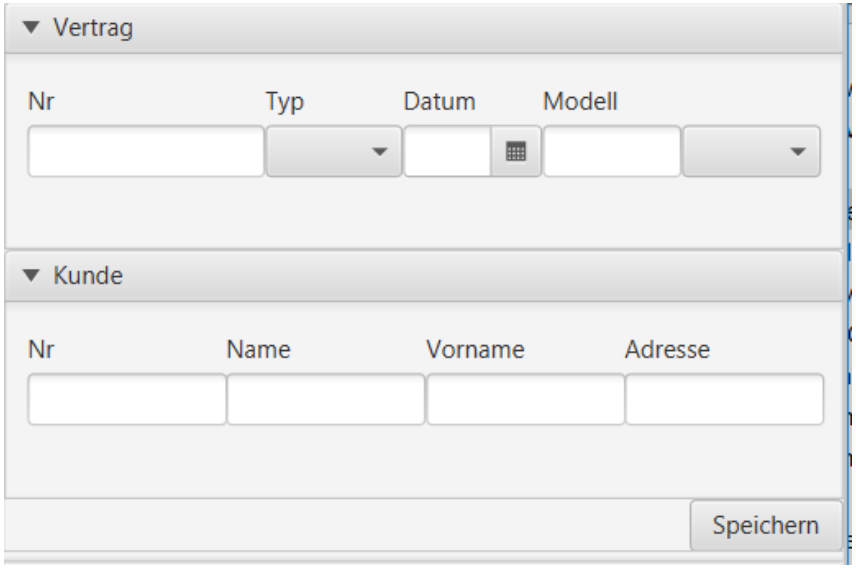
```
scatterParent.getChildren().add(contractScatter);
// Verankern des ScatterCharts in der AnchorPane
AnchorPane.setTopAnchor(contractScatter, 1.0);
AnchorPane.setLeftAnchor(contractScatter, 1.0);
AnchorPane.setRightAnchor(contractScatter, 1.0);
AnchorPane.setBottomAnchor(contractScatter, 1.0);s
```

TextField - ComboBox – DatePicker- TreeView – ScatterChart –

PROGRAMMTISCHES BEFÜLLEN DER UI-CONTROLS

Controls der `ContractDetailView`

- Textfelder (`TextField`):
  - **Vertrag Nr** / **Kunde Nr:** nicht editierbar (Einstellung im SceneBuilder)
  - **Name / Vorname / Adresse**: editierbare Textfelder
  - **Modell**: erstes Element für Zahleneingabe
- Datumseingabe (`DatePicker`) für das Datum
- Dropdown-Boxen (`ComboBox`):
  - **Typ** des Vertrags, `ContractType` modelliert als `Enum`
  - **Modell**: zweites Elemente Bezahlrate, `Rate` modelliert als `Enum`



Befüllen der Controls der `ContractDetailView`

- Wir unterscheiden
  - → Controls mit **statischem** Inhalt, der unabhängig von den Vertragsobjekten ist und im `initialize` der View gesetzt wird: im Beispiel `ComboBox`'en für `ContractType` und `Rate`
  - → Controls mit **dynamischem** Inhalt, der abhängig von selektierten Vertragsobjekt ist und im `update` der View gesetzt wird: im Beispiel alle Controls

- Der Inhalt der `ComboBox` wird in JavaFX in einem zugeordneten `SingleSelectionModel` verwaltet.

- Dieses Modell muss zu Beginn mit einer `ObservableList` initialisiert werden.

- Die Klasse `FXCollections` hat eine Reihe von statischen Methoden, die Java-Collections in JavaFX-Observable-Collections überführt: z.B. `FXCollection.observableArrayList(T…);`

- Mit `setItems(ObservableList<T> …)` wird das Modell der `ComboBox` initialisiert.

Befüllen der ComboBox'en

```
public void initialize() {

    contractTitledPane.setText(contractTitel);
    customerTitledPane.setText(customerTitel);


    contractTypeBox.setItems(
         FXCollections.observableArrayList(
                 ContractType.values()));

    contractRateBox.setItems(
         FXCollections.observableArrayList(
                 Rate.values()));

...
}
```

Initialize für die Controls mit statischem Inhalt

Setzen der Titel für die TitledPanes für Vertrag und Kunde

Initialisieren des Modells der ComboBox für die Vertragstypen mit den Werten des Enums ContractType

Initialisieren des Modells der ComboBox für die Vertragstypen mit den Werten des Enums Rate

Nach dem Initialisieren ist in den ComboBox'en noch kein Eintrag selektiert, da kein Vertrag selektiert ist.

Befüllen der Controls mit dynamischem Inhalt

update mit Vertragsdaten

```
public void update(Contract contract) {

    this.contract = contract;
    contractNrField.setText(contract.getContractId());
    Customer customer = contract.getCustomer();
    customerNrField.setText(customer.getCustomerId());
    customerFirstNameField.setText(customer.getFirstName());
    customerSurNameField.setText(customer.getSurName());
    customerAddressField.setText(customer.getAddress());

    contractDateField.setValue(contract.getContractDate());

    contractAmountField.setText(
          contract.getPaymentModel().getAmount().getEuro() + " ");


    contractTypeBox.getSelectionModel().
          select(contract.getContractType().ordinal());
    contractRateBox.getSelectionModel().
          select(contract.getPaymentModel().getRate().ordinal());

}
```

Befüllen der Textfelder

Setzen des Wertes des DatePicker

Setzen des Wertes eines Textfeldes für Zahlen

Selektieren von ContractType und Rate eines Vertrages in den ComboBox'en

Textfelder mit eingeschränkten Formaten für die Eingabe

- Das Textfeld für die Vertragshöhe soll nur die Eingabe von ganzen Zahlen erlauben.

- Dazu muss ein spezieller Formatierer für das Textfeld gesetzt werden:

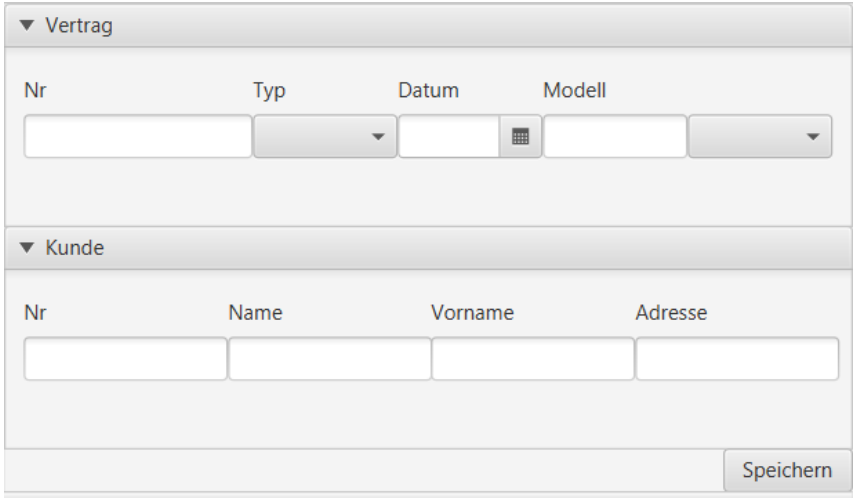
```
contractAmountField.setTextFormatter(
     new TextFormatter<>(
             new IntegerStringConverter()));
```

- Wird nun in das Textfeld keine Zahl eingegeben, dann wird die Eingabe ignoriert und auf den alten Wert zurückgesetzt.

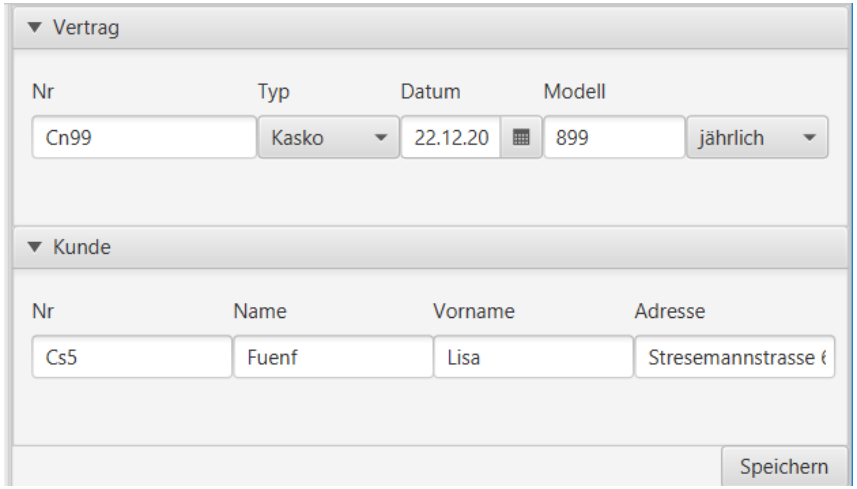
- **Allgemeiner**: Um Eingaben für Textfelder einzuschränken muss ein `TextFormatter` mit einem `StringConverter` gesetzt werden.
  - Der `StringConverter` ist eine abstrakte Klasse, in der die 2 Methoden `toString` und `fromString` überschrieben werden müssen.
  - `toString` wandelt ein Objekt (z.B. einen Gelbbetrag 24,56) in eine Zeichenkette um
  - `fromString` wandelt eine Zeichenkette in den zugehörigen Objekttyp zurück.

`ContractDetailView` vor und nach dem Update

View nach `initialize/ clear`



View nach `update(Contract)`



Control der `ContactScatterView`

- Das `ScatterChart` der View wird mit Datenreihen (`XYChar.Series<T,V>`) befüllt.
  → Aufruf: `contractScatter.getData().add(series)`

- Für jeden Vertragstyp (`ContractType`) wird eine Datenreihe von Daten, die Abschlusshöhe und Abschlussjahr kombinieren, erzeugt.
  – dazu muss die Liste der Verträge zunächst nach Vertragstyp gruppiert werden. (Streaming-Api)

- Jedes Datum einer Reihe ist von Typ `XYChart.Data<T,V>`.
  → Hinzufügen zu einer Reihe: `series.getData().add(… aData …)`

Befüllen des ScatterChart

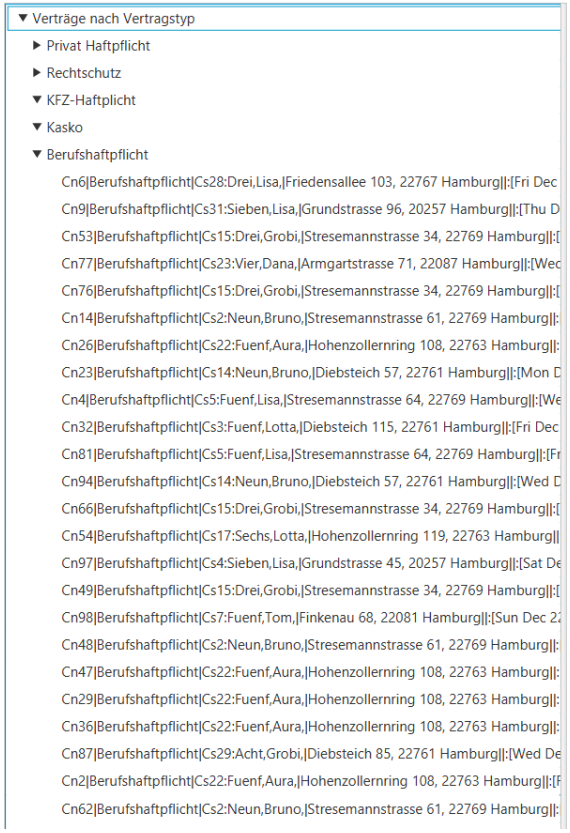
```
// Gruppieren nach Vertragstyp
Map<ContractType, List<Contract>> contractsByType =
          contracts.stream().collect(Collectors.groupingBy(Contract::getContractType));

// Für jeden ContractType wird hier eine Serie von Daten berechnet  Kombination aus Kosten und Jahr)
for (Map.Entry<ContractType, List<Contract>> typeContracts : contractsByType.entrySet()) {

    // Serie für einen Vertragstyp
    XYChart.Series<Integer, Integer> series = new XYChart.Series();
    // Name der Serie = Typ des Vertrags
    series.setName(typeContracts.getKey().toString());
    // Calendar für das Ermitteln des Jahres
    Calendar cal = Calendar.getInstance();
    // Für Verträge des selben Vertragstyps werden die Daten Koordinaten eingetragen. An dieser Position
    // erscheint dann das Symbol des entsprechenden ContractType
    typeContracts.getValue().forEach(con -> {
        cal.setTime(con.getContractDate());
        series.getData().
                add(new XYChart.Data(con.getPaymentModel().getAmount().getEuro(),
                                     cal.get(Calendar.YEAR)));
    });
    // zum Schluss wird die Serie dem ScatterChart hinzugefügt
    contractScatter.getData().add(series);
}
```

Befüllen der `TreeView`

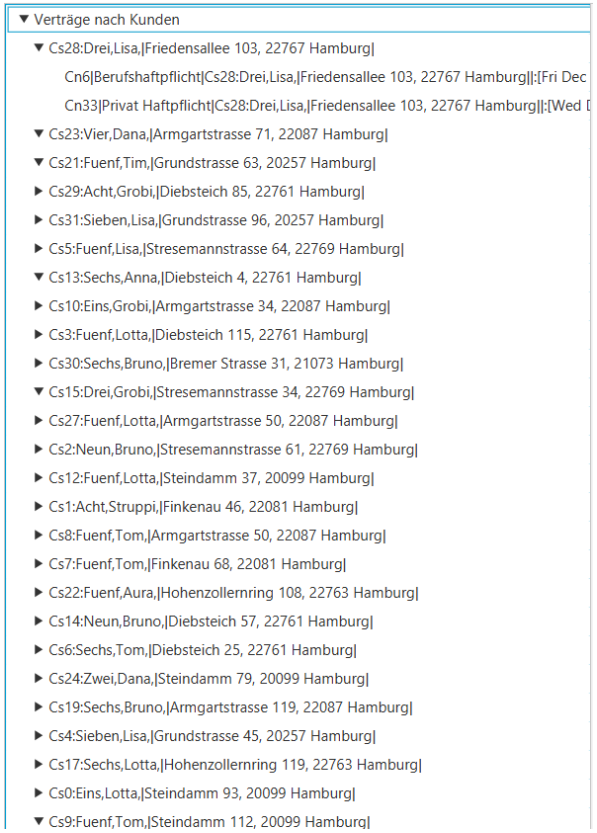
- In der Beispiel-Applikation werden die Verträge nach Vertragstyp und Kunde gruppiert.
- Die `TreeView<T>` besteht in beiden Fällen aus
  - → einer Wurzel, die über die Gruppierung des Baumes Auskunft gibt
  - → mehreren Zwischenknoten für die Gruppen
  - → mehreren Blättern, die die Vertragsobjekte enthalten und den Gruppen zugeordnet sind.
- Der Inhalt einer `TreeView<T>` wird als Baum von `TreeItem<T>` Elementen verwaltet. Der View wird der Wurzelknoten des Baumes über `setRoot( … )` zugeordnet.
- Daher muss die Liste von Verträgen zunächst in einem Baum aus `TreeItem<T>` Elementen für die entsprechenden Gruppen transformiert werden.
- Diese Transformation wird von den statischen Methoden der Hilfsklasse `TreeModelHelper` erledigt.
- Da eine `TreeView` nur `TreeItem` Elemente mit Inhalten gleichen Typs zulässt, wir aber insgesamt 4 Inhaltstypen haben, `String` für die Wurzel, `ContractType` oder `Customer` für die Zwischenknoten und `Contract` für die Blätter, muss mit `TreeItem<Object>` gearbeitet werden.

`TreeView` mit den 2 Gruppierungen

Nach Vertragstyp



Nach Kunde



Erzeugen des `TreeItem` -Baums nach Vertragskategorie

```
public static TreeItem<Object> createContractTreeByContractType(
          TreeItem<Object> root, List<Contract> contracts){

    if (byContractType==null){

      Stream.of(ContractType.values()).map(ct -> {
            TreeItem<Object> typeNode = new TreeItem<>(ct);


            typeNode.getChildren().addAll(

                    contracts.stream().

                       filter(cont -> cont.getContractType()==ct).
                       map(cont -> new TreeItem<Object>(cont)).
                       collect(Collectors.toList()));


            return typeNode;
        }).forEach(ct -> root.getChildren().add(ct));
        byContractType = root;
    }
    return byContractType;
}
```

Wurzel TreeItem<Object> root wird beim Aufruf übergeben

Zwischenknoten: ContractType auf ein TreeItem abbilden

Kindknoten hinzufügen.

Kinder der Zwischenknotens: Alle Verträge, die den gleichen ContractType (ct) wie der Zwischenknoten haben, auf ein TreeItem abbilden.

Zwischenknoten für ContractType der Wurzel des Baumes (root) hinzufügen.

REGISTRIEREN DER EVENT-HANDLER

`ContractDetailView`: Speichern der Änderungen

- Wird in der `ContractDetailView` der `Button` speichern gedrückt, dann
  - → sollen alle Änderungen aus den GUI-Controls in das `Contract` -Objekt übertragen werden
  - → soll die Änderung in der `ContractTreeView` sichtbar werden
  - → soll die Änderung in der `ContractScatterView` sichtbar werden. Eigentlich haben nur die Änderungen am Jahr des Vertragsabschlussdatum und der Vertragshöhe Auswirkung auf die Darstellung der View, aber aus Gründen der Vereinfachung wird bei jeder Änderung die Ansicht aktualisiert

- Um dieses Verhalten umzusetzen wird
  - → ein `EventHandler` für ein `ActionEvent` mit `setOnAction` beim Button regristiert, der
    - den Inhalt der GUI-Controls in das `Contract` Objekt überträgt und
    - `update` auf der `ContractTreeView` sowie der `ContractScatterView` aufruft

ContractDetailView: Speichern der Änderungen

```
saveButton.setOnAction((ActionEvent e) -> {
    if (contract != null) {

        contract.setContractType(contractTypeBox.getValue());

        contract.setContractDate(Date.from(
          contractDateField.getValue().atStartOfDay().atZone(
                                        ZoneId.systemDefault()).toInstant()));

        contract.setPaymentModel(new PaymentModel(
          contractRateBox.getValue(),
          new Amount(Integer.parseInt(contractAmountField.getText()))));

        contract.getCustomer().setFirstName(customerFirstNameField.getText());
        contract.getCustomer().setSurName(customerSurNameField.getText());
        contract.getCustomer().setAddress(customerAddressField.getText());

        contractScatterView.updateView();
        contractTreeView.updateView();
    }

});
```

Registrieren eines EventHandler für ein ActionEvent

Übertragen der Daten aus den Controls in das Contract Objekt

Vertragstyp aus der ComboBox lesen

LocalDate aus dem contractDateField lesen und in einen Date-Objekt umwandeln

Rate und Höhe des Vertrags als PaymentModel setzen

Kundendaten aus den Textfeldern lesen

Das Update der beiden anderen Views anstossen

ContractTreeView: updateView

```
public void updateView() {
    int currentSelection =
        contractTree.getSelectionModel().getSelectedIndex();

    contractTree.refresh();

   contractTree.getSelectionModel().select(currentSelection);
}
```

currentSelection merken, damit diese nach dem refresh wieder gesetzt werden kann.

refresh auf einer TreeView bewirkt, dass der Inhalt der TreeItem's erneut gelesen werden (Inhalte erneut gerendert werden)

currentSelection erneut setzen

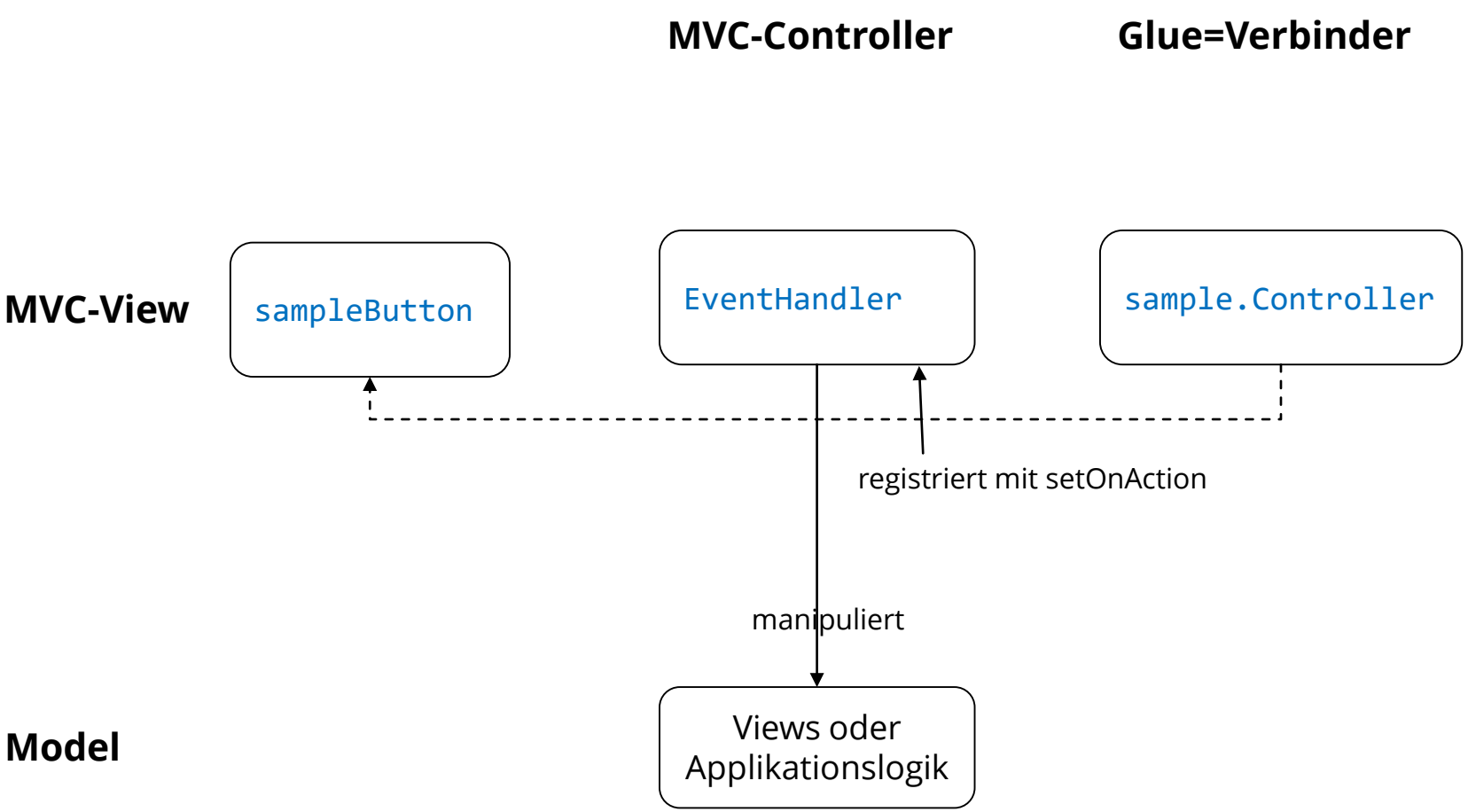
ContractScatterView: updateView

```
public void updateView() {

    if (scatterParent.getChildren().size() > 0) {

        scatterParent.getChildren().clear();
    }

    initialize();
}
```

Wenn die View bereits eine ScatterChart enthält, dann muss dieser gelöscht werden.

Initialize berechnet das neue ScatterChart auf Basis der geänderten Daten.

Veranschaulichung



`ContractTreeView`: Selektion von Verträgen

- Wird in der `TreeView` ein Vertrag selektiert, dann soll der Vertragsinhalt in der `ContractDetailView` angezeigt werden.
- Um über das Selektionsereignis informiert zu werden, muss ein **Eventhandler** für dieses Ereignis registriert sein.
- Der Eventhandler ist ein `ChangeListener` für die `Properties` `selectedItem` / `selectedIndex` des `SelectionModel` der `TreeView`.
- Der Eventhandler wird auf dem Model der View (`contractTree.getSelectionModel()`) auf die `selectedItemProperty()` mit `addListener(aChangeListener)` registriert.
- Wenn ein Selektionsereignis auftritt, werden alle registrierten Handler informiert und deren Quelltext für die Behandlung des Events ausgeführt.
- Die Handler werden als anonyme innere Klassen / Lambda-Ausdrücke implementiert.

Exkurs: One-Method-Interfaces und Lambda-Ausdrücke

- Jede Implementierung eines One-Method-Interfaces über eine anonyme innere Klasse lässt sich seit Java 8 in einen Lambda Ausdruck transformieren. Dabei wird die Konstruktion einer anonymen inneren Klasse und die Methodendefinition ersetzt durch die Aufzählung der Lambda-Variablen gefolgt von der Lambda-Körper.

```
public class OneMethodInterface {

    public static void main(String[] args) {
        IOneMethod<String,Integer> anonymousIOne = new IOneMethod<>() {
            @Override
            public void onlyMethod(String arg1, Integer arg2) {

                System.out.printf("Anonymous Impl %s %s\n", arg1,arg2);
            }
        };
        anonymousIOne.onlyMethod("Holy", 5);


        IOneMethod<String,Integer> lambda = (arg1,arg2) -> {

            System.out.printf("Lambda Impl %s %s\n", arg1,arg2);

        };
        lambda.onlyMethod("Holy", 5);
    }
}
interface IOneMethod<T,V>{
    public void onlyMethod(T arg1,V arg2);
}
```

Anonyme innere Klasse

Methodenname und Parameter

Methodenimplementierung

Methodenaufruf

Lambda-Variablen

Lambda-Körper

Methodenaufruf

Einzige Methode des Interfaces

Registrieren eines Eventhandlers über eine anonyme innere Klasse

Registrieren eines Eventhandlers auf Änderungen des selektierten Elements im Model der TreeView

```
contractTree.getSelectionModel().selectedItemProperty().addListener(

new ChangeListener<TreeItem<Object>>() {

    @Override
    public void changed(
                    ObservableValue<? extends TreeItem<Object>> observable,
                    TreeItem<Object> oldValue,
                    TreeItem<Object> newValue) {

        if (newValue != null &&
            newValue.getValue() instanceof Contract) {
            contractDetail.update((Contract) newValue.getValue());
        }

    }

});
```

Der Eventhandler für Properties ist ein ChangeListener hier realisiert als anonyme innere Klasse.

Einzige Methode des ChangeListener Interfaces

Implementierung der Methode: Nur wenn der neue Wert != null und vom Typ Contract ist, kann update auf der ContractDetailView aufgerufen werden. (siehe Folie 60)

Registrieren eines Eventhandlers über einen Lambda-Ausdruck

Registrieren eines Eventhandlers auf Änderungen des selektierten Elements im Model der TreeView

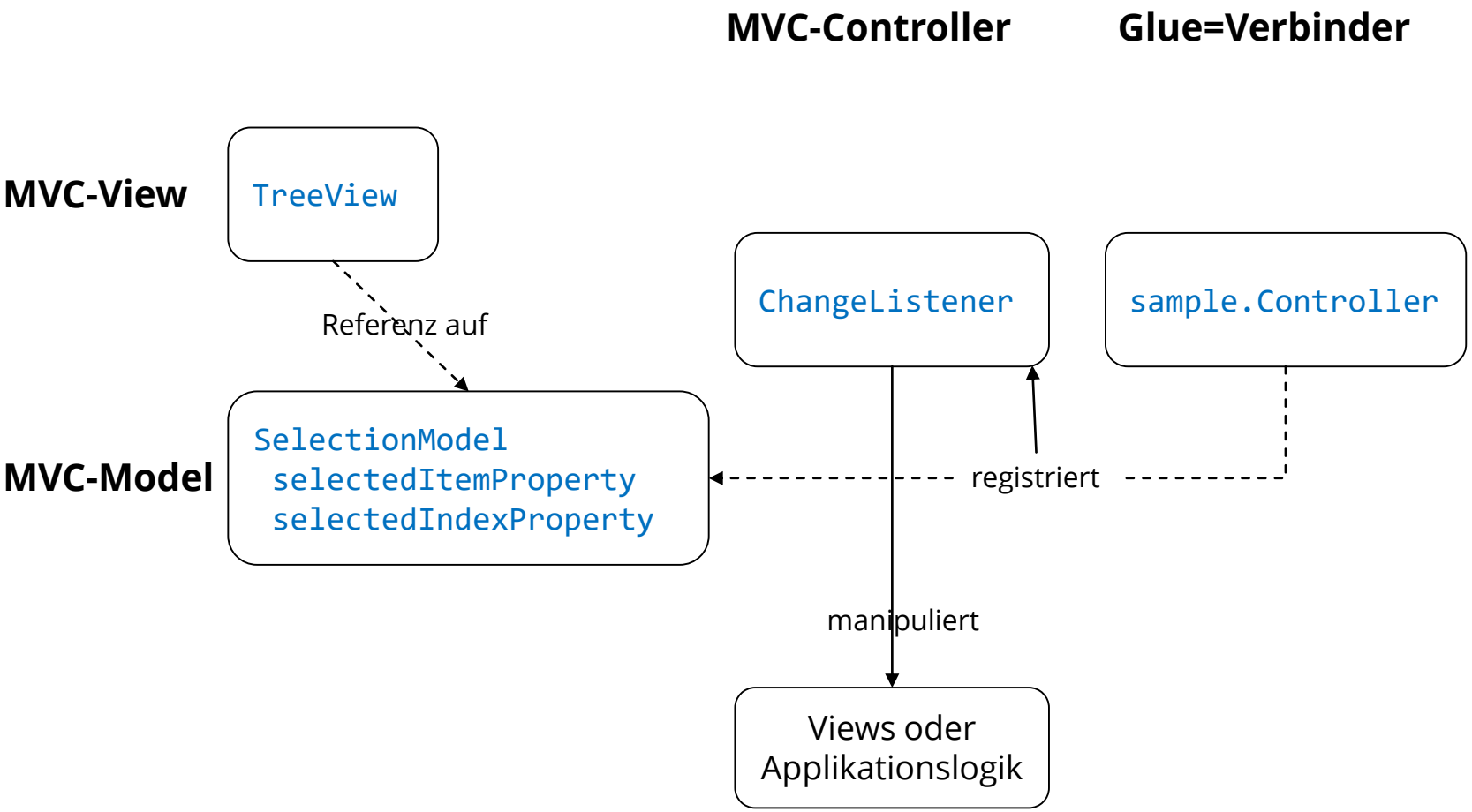
```
contractTree.getSelectionModel().selectedItemProperty().addListener(

    (observable, oldValue, newValue) -> {

     if (newValue != null &&
         newValue.getValue() instanceof Contract) {
         contractDetail.update((Contract) newValue.getValue());
     }
});
```

Lambda-Variablen

Lambda-Körper:
Nur wenn der neue Wert != null und vom Typ Contract ist, kann die ContractDetailView aktualisiert werden.

- Wird in der `TreeView` ein neuer Vertrag selektiert, dann wird auf der View `contractDetail` die Methode `update` aufgerufen, die die Controls der View mit den Werten des neuen Vertrags befüllt. (vgl. Folie 60)

Veranschaulichung



Abschließende Bemerkung

- nicht nur `TreeView` hat ein `SelectionModel`.
- `ListView`, `ComboBox`, `ChoiceBox` und `TableView` verfügen ebenfalls über ein `SelectionModel`, über das auf gleiche Weise wie für die `TreeView` `ChangeListener` an die Properties gebunden werden können.
- Das Registrieren von `ChangeListenern` für diese Controls verbleibt zum Selbststudium.

`ContractTreeView`: Hinzufügen eines Kontextmenüs

- Über einen Rechtsklick soll bei allen Knoten der `TreeView` ein Kontextmenü erscheinen, das den Wechsel zwischen den Gruppierungen nach Vertragstyp und nach Kunde ermöglicht.
- Der Inhalt des `MenuItem` 's ist abhängig von der aktuell angezeigten Gruppierung in der `TreeView`.
- Zum Zeitpunkt eines Rechts-Clicks auf ein `TreeItem` 's ist die Gruppierung bekannt (boolesche Variable `switchToCustomer`).
- Durch Registrierung einer `CellFactory` für die „Zellen" des Baumes kann dynamisch über den Inhalt der Menuitems des Kontextmenüs entschieden werden.
- Eine `CellFactory` manipuliert typicherweise die Inhalte der `TreeItem` Elemente. Kontext-Menüs werden sehr häufig zur Manipulation der Objekte, an die sie gebunden sind, verwendet.
- Die Manipulation erfolgt in der `updateSelected` Methode der `TreeCell`-Elemente der `CellFactory`.
- Diese Methode wird für die Inhalte der `TreeItem` 's aufgerufen, wenn ein `TreeItem` selektiert wurde.
- In dieser Methode kann dann entschieden werden, welcher Inhalt im Kontextmenu angezeigt werden soll.
- Da der Inhalt der `TreeItem` als Text gerendert wird, verwenden wir eine `TextFieldTreeCell`.

ContractTreeView: Hinzufügen eines Kontextmenüs Überblick

(1) Registrieren einer CellFactory mit TextFieldTreeCells für die TreeView

```
contractTree.setCellFactory((TreeView<Object> tv) -> new TextFieldTreeCell<>() {
        private final MenuItem switchItem;

        private final ContextMenu switchViewMenu = new ContextMenu();
        {
```

(2) Erzeugen des Menuitems

(3) Registrierung eines Eventhandlers

```
        }

        @Override
        public void updateSelected(boolean selected) {
```

(4) Beschriftung des Menuitems abhängig vom Zustand der TreeView

```
        }
    });
}
```

ContractTreeView: Erzeugen des Menuitems und Registrierung eines Eventhandlers

```
private final ContextMenu switchViewMenu = new ContextMenu();
{
    switchItem = new MenuItem();
    switchViewMenu.getItems().add(switchItem);

    switchItem.setOnAction((ActionEvent e) -> {
        if (switchToCustomer) {

            contractTree.setRoot(
                TreeModelHelper.createContractTreeByCustomer(
                    new TreeItem<>("Verträge " + switchToCustomerText),
                    contracts));
            switchToCustomer = false;

        } else {

            contractTree.setRoot(
              TreeModelHelper.createContractTreeByContractType(...);
            switchToCustomer = true;
        }
        contractDetail.clear();
        contractTree.setShowRoot(true);
        contractTree.getRoot().setExpanded(true);
        return;
    });
    setEditable(false);
    setContextMenu(switchViewMenu);
}
```

(2) Erzeugen des Menuitems

(3) Registrierung eines Eventhandlers bei Selektion

Hier wird in die Sicht „nach Kunden“ gewechselt

Hier wird in die Sicht „nach Vertragstyp“ gewechselt

Bei jedem Wechsel werden die Inhalte der ContractDetailView gelöscht.

TextFieldTreeCell ist editierbar. Das wird hier unterbunden.

Setzen des Kontextmenus.

ContractTreeView: Kontextabhängige Anzeige

(1) Registrieren einer CellFactory mit TextFieldTreeCells für die TreeView

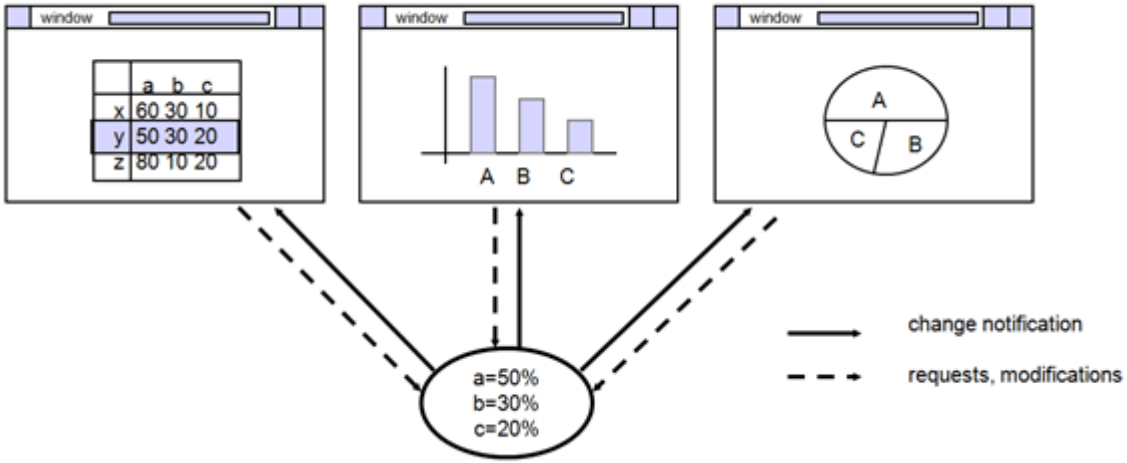
```
contractTree.setCellFactory((TreeView<Object> tv) -> new TextFieldTreeCell<>() {
        private final MenuItem switchItem;

        (2) Erzeugen des Menus / Menuitems

        (3) Registrierung eines Eventhandlers

        @Override
        public void updateSelected(boolean selected) {

          if (switchItem != null) {
                    switchItem.setText(switchToCustomer ?
                              switchToCustomerText :
                              switchToContractTypeText);
          }
        }
    });
}
```

(4) Beschriftung des Menuitems abhängig vom Zustand der TreeView

MVC- MODEL VIEW CONTROLLER

MVC

- MVC ist eines der bekanntesten Designpatterns bei der Entwicklung von graphischen Oberflächen. (sehr alt bereits in Smalltalk realisiert).
- MVC trennt die Datenhaltung von der Darstellung und der Manipulation der Daten
  - **Model**: hält die Daten und informiert die View über Änderungen über „Change Notifications“
  - **View**: übernimmt die Darstellung
  - **Controller**: übernimmt die Manipulation der Daten durch Interpretation von Benutzeraktionen



MVC

Model

- enthält den funktionalen Anwendungskern und kapselt diesen in Daten und Methoden.
- Methoden werden in den Controllern (EventHandler) aufgerufen.
- Mit den Daten werden die Views initialisiert.
- Verschiedene Views lassen sich beim Model registrieren, so dass Änderungen im Model in allen abhängigen Views sichtbar werden.

View

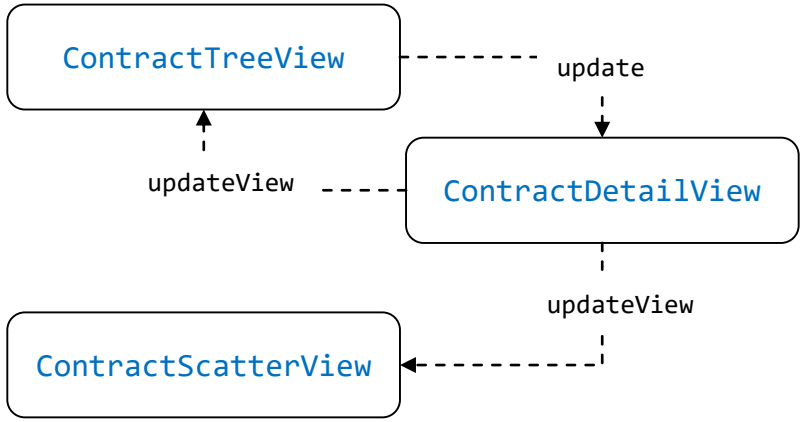
- Verantwortlich für die Darstellung der grafischen Benutzeroberfläche (Layout, Controls, Rendering)

Controller

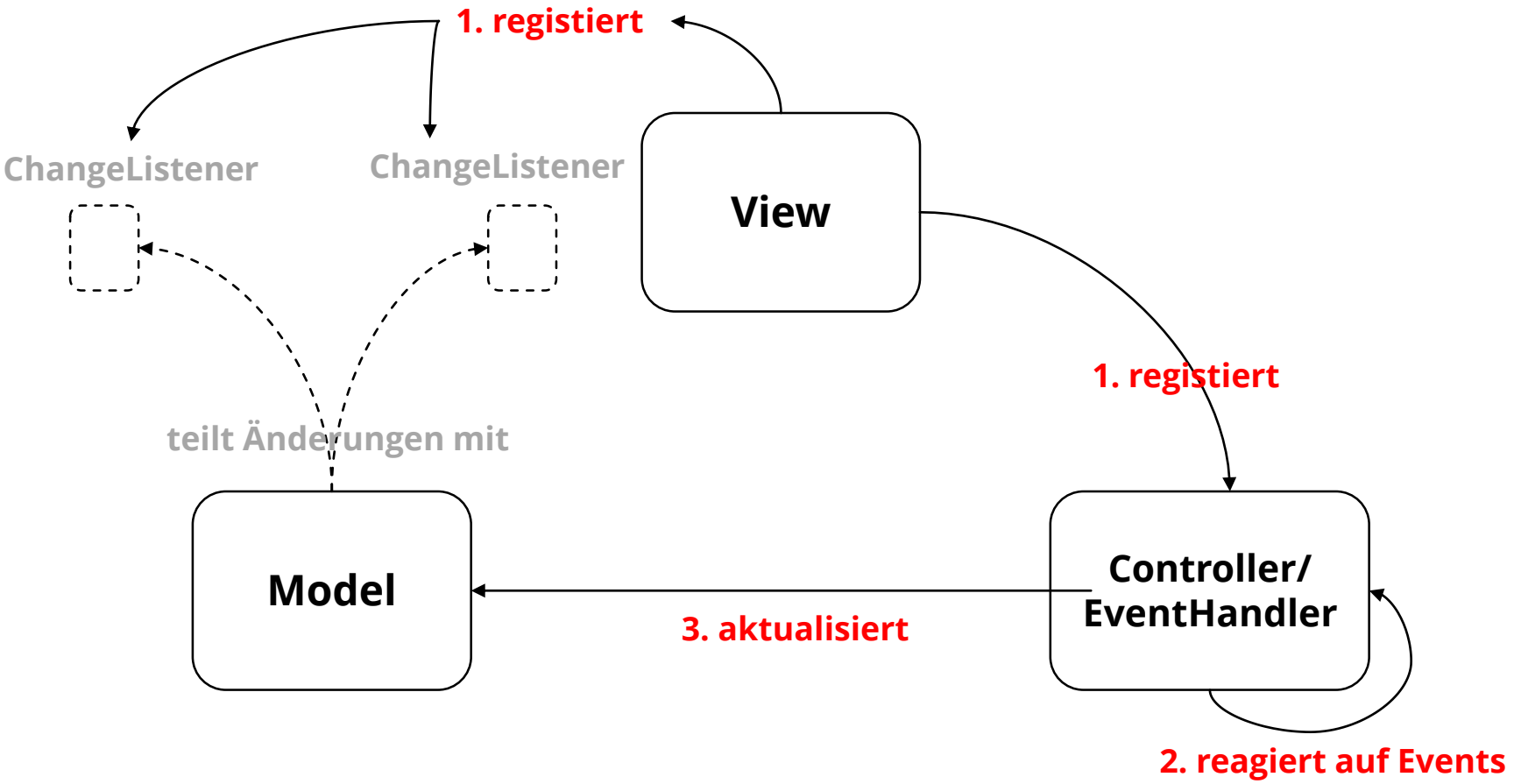
- nimmt Benutzereingaben in Form von Ereignissen (Events) entgegen.
- behandelt Ereignisse, indem er EventHandler bei den Views registriert, die Änderungen im Model vornehmen.

Bewertung der 1‘ten Lösung

- In der ersten Lösung bestehen **Abhängigkeiten zwischen den Views** (siehe rechte Grafik). Änderungen im Model werden zwischen den Views ausgetauscht. Die Lösung folgt nicht dem MVC-Muster.
- Werden dem GUI neue Views hinzugefügt, z.B. ein `PieChart`, dann müssen sich die Implementierungen der bestehenden Views ändern.
- Wird eine neue Funktionalität z.B. das Hinzufügen neuer Verträge, gefordert, entstehen neue Abhängigkeiten zwischen den Views.
- Ziel ist es die 1‘te Lösung in eine MVC konforme Lösung zu refaktorisieren.



Das allgemeine MVC -Muster



MVC in JavaFX

Model

- JavaBeans mit JavaFX-Properties (`ObservableValue` / `ObservableList`)
  - Binden verschiedener Views an Properties macht Model-Änderungen automatisch in den Views sichtbar (**Bindings**).
  - Registrieren von Event-Handlern auf Properties ermöglicht Updates abhängiger Views

- **ODER**: ein GUI-nahes separates Modell, dass veränderliche Eigenschaften als Properties modelliert

View

- Szenengraph: Controls für das Rendering und Layout-Container für die Anordnung der View-Elemente
- spezielle Controls mit `ObservableList` als zugrundeliegendes Model

Controller

- `EventHandler`, die auf Interaktion des Benutzers mit der View reagieren (z.B. das Drücken eines Buttons / die Selektion eines Elementes)
- ändern die Model-Eigenschaften / die Java-FX Properties

ACHTUNG

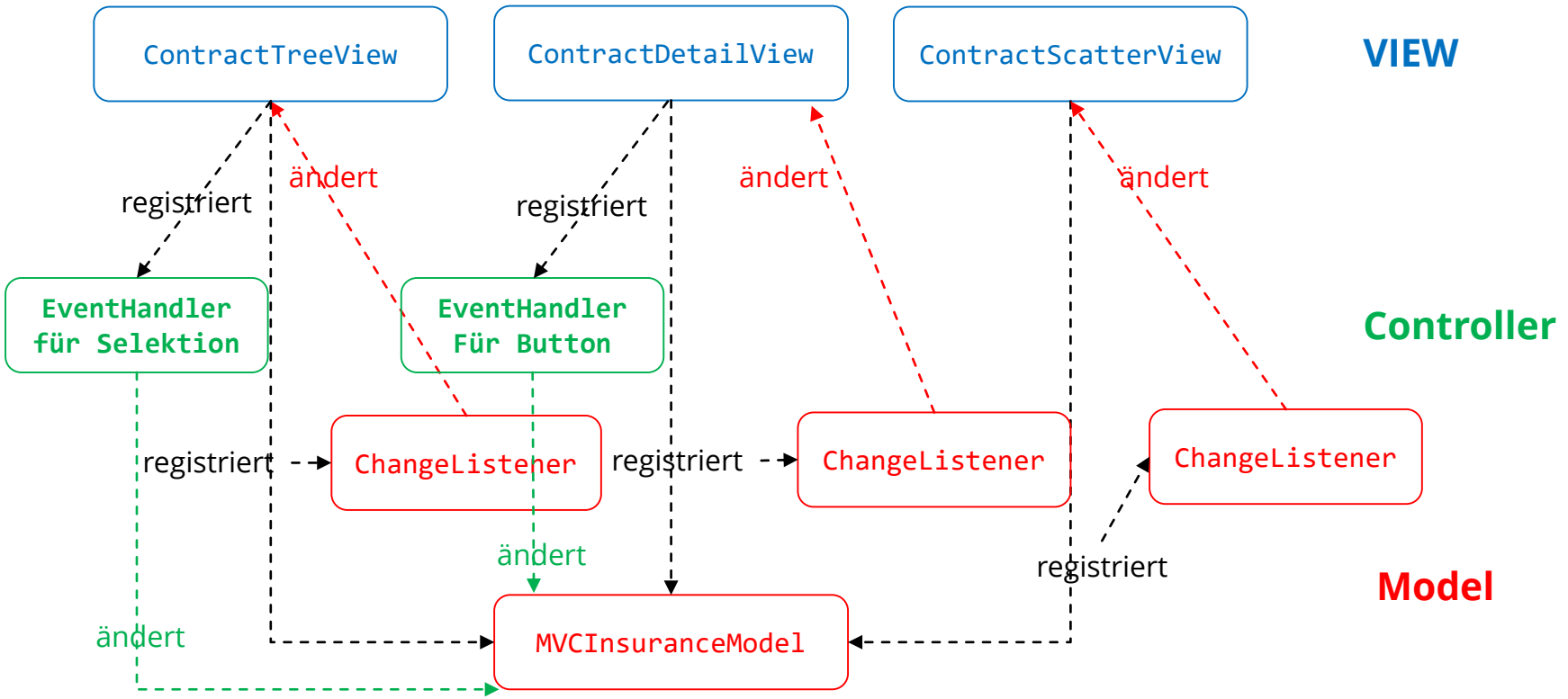
- Die Java-FX Controller-Klasse ist kein Controller im Sinne des MVC Patterns.

- Die Java-FX Controller-Klasse bündelt die View-Komponenten eines Szenengraphen und registriert die EventHandler bei den View-Komponenten.

- Damit ist die Java-FX Controller-Klasse eine Art Glue (Kleber) zwischen View MVC-Controller und Model.

Aufbau der 2‘ten Lösung

MVC-Pattern



JavaFx-Beans

- **JavaFX-Beans** sind Daten-Klassen, mit deren Hilfe sich Model-Daten an GUI-Controls binden lassen.

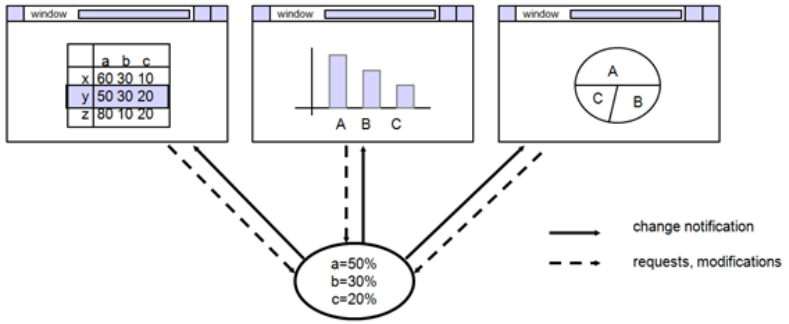
- JavaFX-Beans verfügen über einen Eventmechanismus, der Änderungen an registrierte Listener (`ChangeListener` / `InvalidationListener`) propapiert.

- Dazu müssen die Attribute eines JavaFX-Beans als **Properties** modelliert werden.
  - → An Properties lassen sich `Change-/InvalidationListener` binden.
  - → Properties werden über **Bindings** an GUI-Controls gekoppelt.

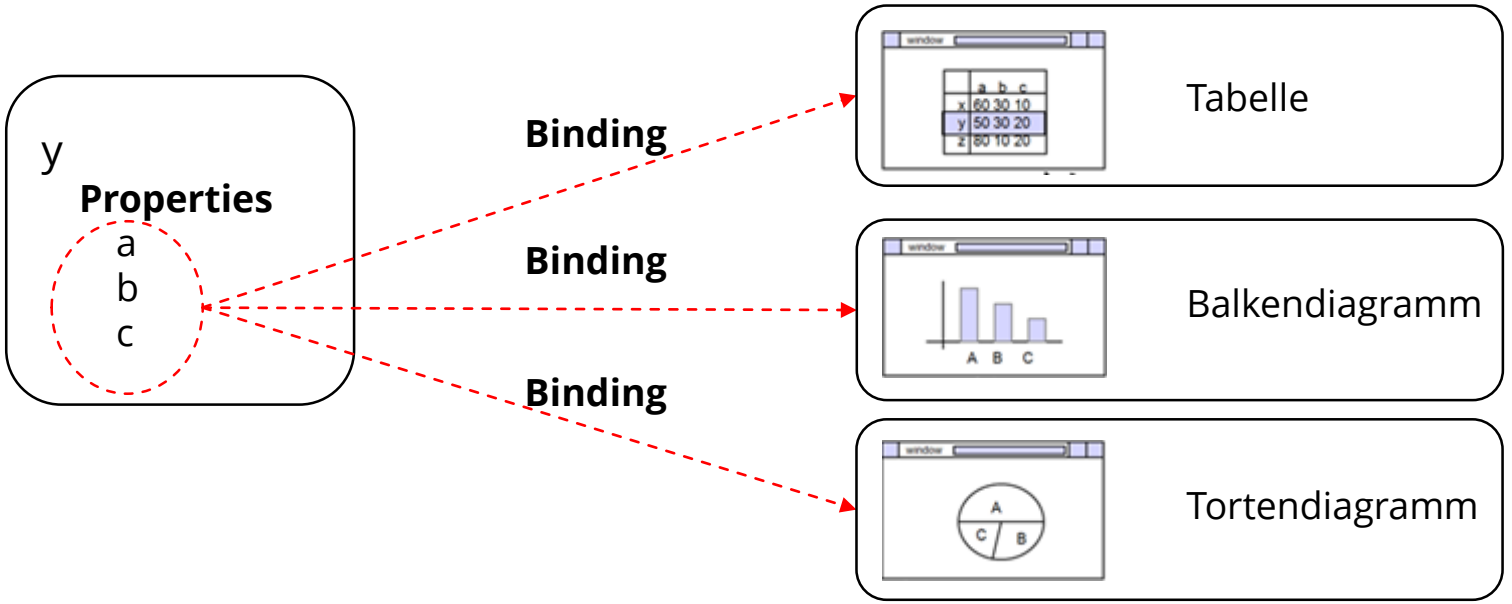
Properties und Bindings

- **Properties**:
  - spezielle Eigenschaften von Objekten, die Auskunft über Änderungen ihres Wertes geben

- **Bindings**:
  - definieren Abhängigkeiten zwischen Objekten auf Basis von Properties, in dem sie Änderungen in den Properties aktiv propagieren
  - **Beispiel**: Wenn sich in der Grafik für das Objekt y einer der Werte a-c ändert, dann wird diese Änderung sofort in allen Views sichtbar werden.
  - Das funktioniert in unserem Versicherungsbeispiel aber nicht auf diese Art, da die Modellobjekte in verschiedenen Datenstrukturen gewrapped sind.



Bindings auf Properties machen Änderungen in abhängigen Objekten / Controls sichtbar



Konventionen für Properties in JavaFX

abstrakte Basisklasse

Property amountDue

Builtin-Property Klasse für Basisdatentyp

```
public class Bill {

    private DoubleProperty amountDue = new SimpleDoubleProperty();

    public double getAmount() {
        return amountDue.get();
    }

    public void setAmount(double amount) {
        amountDue.set(amount);
    }

    public DoubleProperty amountDueProperty() {
        return amountDue;
    }

}
```

Reader und Writer / Getter und Setter für Inhalt der Property amountDue

Reader / Getter für Property amountDue

Vordefinierte Property-Klassen in JavaFX

Abstrakte Basisklasse	Implementierungs-Klasse

Basisdatentypen (nicht char und byte):
`<Btype>Property`
(`btype` ist der Wrappertyp des Basisdatentyps)
 	
`DoubleProperty`
`IntegerProperty`
	Basisdatentypen:
	`Simple<Btype>Property`
	(`btype` ist der Wrappertyp des Basisdatentyp)
	
	`SimpleDoubleProperty`
	`SimpleIntegerProperty`

`StringProperty`	`SimpleStringProperty`
`ListProperty`, `SetProperty`, `MapProperty`	`SimpleListProperty`, `SimpleSetProperty`, `SimpleMapProperty`
`ObjectProperty`	`SimpleObjectProperty`

Event-Handler für Wertänderungen einer Property

- Properties signalisieren Wert-Änderungen, auf die Event-Handler / `ChangeListener` registriert werden können.

```
public class Bill {

    private DoubleProperty amountDue = new SimpleDoubleProperty();

    public double getAmount() {

    public void setAmount(double amount) {

    public DoubleProperty amountDueProperty() {

    public static void main(String[] args) {
        Bill bill = new Bill();
        bill.amountDueProperty().addListener(
                (ObservableValue<? extends Number> ov, Number oldVal,
                        Number newVal) -> {
                    System.out.println("Bill has changed to " + newVal);
                });

        bill.setAmount(24.89);
    }
}
```

registrieren eines ChangeListener für Änderungen in amountDue als Lambda Ausdruck

jede Änderung von amountDue triggert den Listener und damit die Konsolenausgabe

VERTIEFUNG

BINDINGS

Bindings

- bestehen aus ein oder mehreren Quellen, den **Dependencies**.
- beobachten die **Dependencies** und nehmen Aktualisierungen automatisch vor.
  - **high-level Bindings-API**:
    - definiert mit dem „Fluent API“ oder alternativ den statischen Methoden der `Bindings`-Klasse typische Methoden für das Binden von Objekt-Properties.
    - Methoden entsprechen den Operatoren für Basisdatentypen und erzeugen ein Binding, dessen Wert kompatibel mit den Ergebnis der Operatoren ist.
  - (low-level API: komplexere Nutzung aber effizienter und weniger Speicherbedarf)
- Uni-oder bidirektionales Binden von Properties wird unterstützt.

Fluent-API

```
public class BindingsDemo {

    public static void main(String[] args) {
        IntegerProperty ip1 = new SimpleIntegerProperty(1);
        IntegerProperty ip2 = new SimpleIntegerProperty(2);
        IntegerProperty ip3 = new SimpleIntegerProperty(3);
        IntegerProperty ip4 = new SimpleIntegerProperty(4);

        NumberBinding nbs = ip1.add(ip2);
        NumberBinding nbc = ip1.add(ip2.multiply(ip3)).divide(ip4);
        System.out.println(nbs.getValue());
        System.out.println(nbc.getValue());
        ip1.setValue(3);
        System.out.println(nbs.getValue());
        System.out.println(nbc.getValue());
    }
```

Methoden des Fluent-API Binden Operationen an Properties

Ändert sich eines der Properties, dann auch der Wert des Bindings

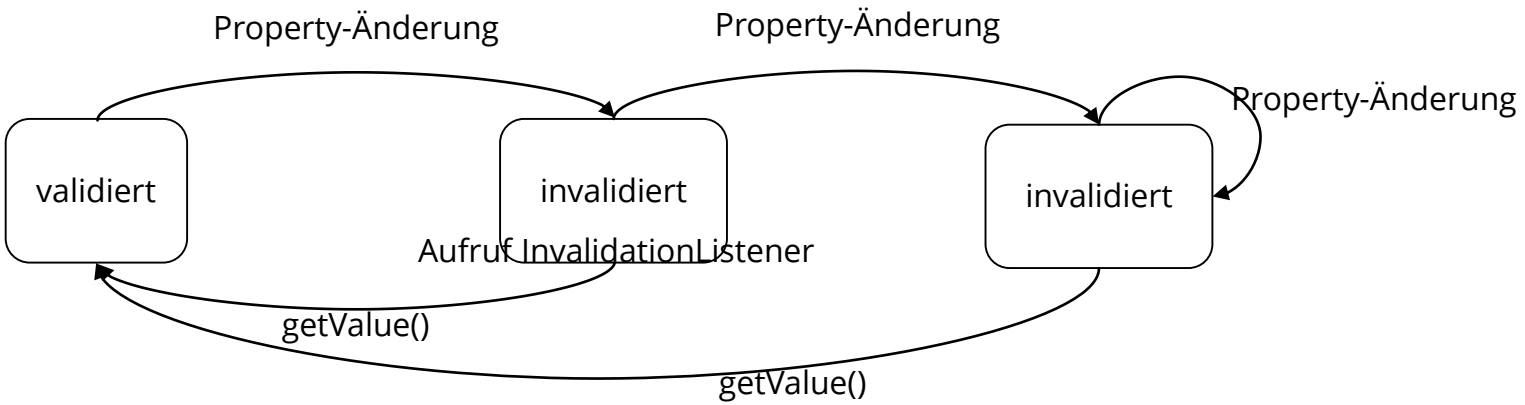
Fluent API und statische Methoden von `Bindings`

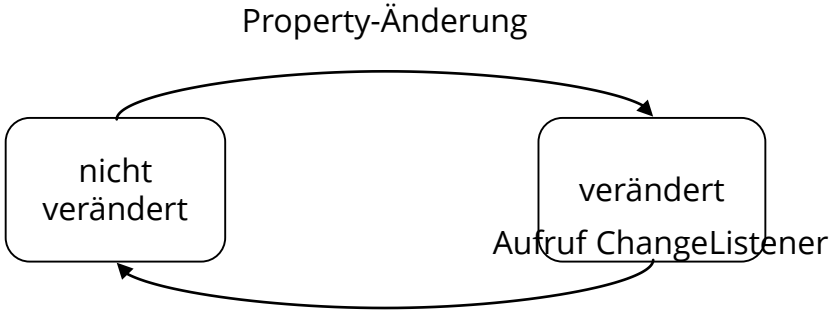
```
public class BindingsDemoMixed {

    public static void main(String[] args) {
        IntegerProperty ip1 = new SimpleIntegerProperty(1);
        IntegerProperty ip2 = new SimpleIntegerProperty(2);
        IntegerProperty ip3 = new SimpleIntegerProperty(3);
        IntegerProperty ip4 = new SimpleIntegerProperty(4);

        NumberBinding nbs = Bindings.add(ip1,ip2);
        NumberBinding nbc = ip1.add(Bindings.multiply(ip2,ip3)).divide(ip4);
        System.out.println(nbs.getValue());
        System.out.println(nbc.getValue());
        ip1.setValue(3);
        System.out.println(nbs.getValue());
        System.out.println(nbc.getValue());
    }

}
```

statische Methoden der Klasse Bindings sind äquivalent zu den Methoden des Fluent API

Mischformen sind auch möglich

Invalidierung versus Änderung





Verhalten eines Invalidierungs-Listeners

```
public class BindingsObservableInvalidation {

    public static void main(String[] args) {
        IntegerProperty ip1 = new SimpleIntegerProperty(1);
        IntegerProperty ip2 = new SimpleIntegerProperty(2);

        NumberBinding nbs = ip1.add(ip2);

        nbs.addListener((Observable o) -> {
            System.out.println("invalid ");
        });

        ip1.set(20);
        ip1.set(3);
        ip1.set(7);
        System.out.println(nbs.getValue());
        ip1.set(4);
        ip1.set(7);
    }
}
```

```
invalid
9
invalid
```

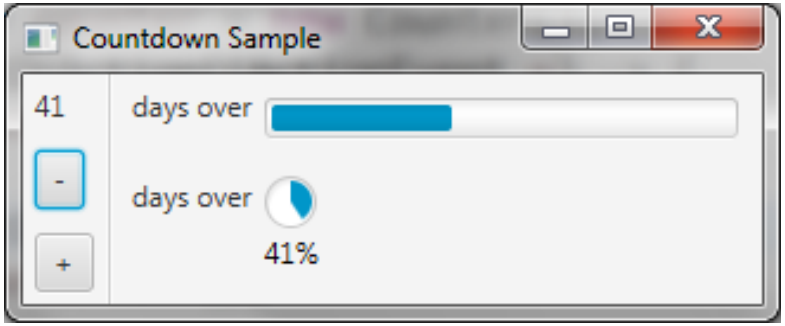
Verhalten eines Change-Listeners

```
public class BindingsObservableChange {

    public static void main(String[] args) {
        IntegerProperty ip1 = new SimpleIntegerProperty(1);
        IntegerProperty ip2 = new SimpleIntegerProperty(2);

        NumberBinding nbs = ip1.add(ip2);

        nbs.addListener((ObservableValue<? extends Number> ov, Number oldVal,
                Number newVal) -> {
            System.out.println("changed");
        });

        ip1.setValue(20);
        ip1.setValue(3);
        ip1.setValue(7);
        System.out.println(nbs.getValue());
        ip1.setValue(4);
        ip1.setValue(7);
    }
}
```

```
changed
changed
changed
9
changed
changed
```

Unidirektionales Binden

- **Ziel**: Änderungen einer Properties direkt in abhängigen Properties sichtbar machen
- **Anwendung**: Binden von Control-Inhalten an Properties eines Model-Objektes
- **Bsp.**: Countdown Sample
  - `ProgressBar`, `ProgressIndicator` und `Label` Controls werden an den Stand eines Zählerobjektes gebunden
  - mit den „-“ und „+“ Buttons kann der Zähler herunter und herauf gezählt werden
  - die geänderten Werte werden sofort in allen gebundenen Controls angezeigt



Undirektionales Binden (1)

```
public class CountdownController {
    @FXML
    protected void initialize() {
        Counter counter = new Counter();
        dec.setOnAction((ActionEvent e) -> {
            counter.dec();
        });
        inc.setOnAction((ActionEvent e) -> {
            counter.inc();
        });

        bar.progressProperty().bind(counter.countProperty().divide(100.0));
        indi.progressProperty().bind(counter.countProperty().divide(100.0));

        counter.setCount(Integer.parseInt(initial.getText()));

        initial.textProperty().bind(counter.countProperty().asString());
    }
```

Event-Handler für die Buttons „-“ und „+“ ändern das Modell-Objekt Counter

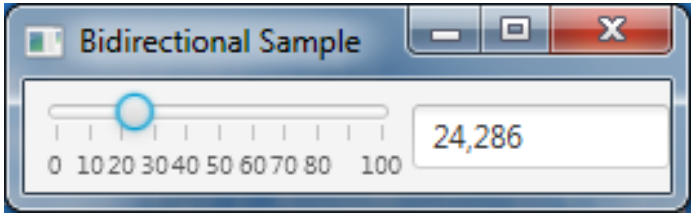
bindet Properties der Controls an die countProperty des Objektes counter

Änderungen von countProperty werden unmittelbar in den Controls sichtbar

Undirektionales Binden (2)
das Modell `Counter`

```
class Counter {
    private IntegerProperty count = new SimpleIntegerProperty(0);
    public void dec() {
        setCount(getCount() - 1);
    }
    public void inc() {
        setCount(getCount() + 1);
    }
    public void setCount(int val) {
        count.set(val);
    }
    public int getCount() {
        return count.get();
    }
    public IntegerProperty countProperty() {
        return count;
    }
```

Bidirektionales Binden (2)




```
public class BidirectionalController {

    @FXML
    private Slider slider;
    @FXML
    private TextField text;
    @FXML
    protected void initialize(){
    slider.setValue(10);

    text.textProperty().bindBidirectional(
          slider.valueProperty(),
          NumberFormat.getNumberInstance());
    }
}
```

Bidirektionales Binden zwischen der textProperty von text und der valueProperty von slider

Das zweite Argument konvertiert den Text in eine Zahl für den Slider

UMSETZUNG DER VARIANTE 2

Das `MVCInsuranceModel`

- Das Modellieren des Inhaltes der `ContractDetailView` eines `Contract` – Objektes als JavaFX-Bean würde den Quelltext sehr unübersichtlich werden lassen.
- Daher beschränken wir uns im Modell auf 2 Properties und eine `ObservableArrayList`.
- Die `ObservableArrayList` ließe sich später dazu benutzen, um das Einfügen neuer Verträge zu entdecken und danach die Aktualisierung der `TreeView` anzustoßen.
- Die Property `initialContractProperty` wird gesetzt, wenn ein Vertrag z.B. in der `TreeView` selektiert wurde.
- Die Property `changedContractProperty` wird gesetzt, wenn ein Vertrag geändert wurde.

```
public class MVCInsuranceModel {

    private ObservableList<Contract>
                        contracts;
    private ObjectProperty<Contract>
                        initialContractProperty;
    private ObjectProperty<Contract>
                        changedContractProperty;

}
```

Das `MVCInsuranceModel`

- Das Modell erzeugt im Konstruktor die Liste der Verträge, indem es den `ContractGenerator` aufruft.

- Das Modell hat darüber hinaus Methoden,
  - → um die Liste der Verträge zu lesen
  - → um die Property Inhalte zu lesen und zu setzen
  - → um die Properties zu lesen.

- Wird der Wert der Property `changedContractProperty` gesetzt, dann wird zugleich auch die Liste der Verträge aktualisiert.

Änderungen im `sample.Controller`

- Die `sample.Controller` Klasse erzeugt im `initialize` eine Instanz des `MVCInsuranceModel` und übergibt den Views eine Referenz im Konstruktor.
- Das Erzeugen der Liste der Verträge wurde in die Model-Klasse verschoben.

```
@FXML
protected void initialize() {

    final MVCInsuranceModel mvcInsuranceModel = new MVCInsuranceModel();

    contractTreeView = new ContractTreeView(mvcInsuranceModel);
    contractTreeView.initialize();
    contractDetail = new ContractDetailView(mvcInsuranceModel);
    contractScatterView = new ContractScatterView(mvcInsuranceModel);
}
```

Änderungen in der `ContractTreeView`

- Im `EventHandler` für die Selektion von Elementen, wird jetzt nicht mehr ein `update` auf der `ContractDetailView` aufgerufen.
- Stattdessen werden die Properties des Models `mvcInsuranceModel` gesetzt.
- Das Setzen der Properties triggert die registrierten `ChangeListener`.

```
contractTree.getSelectionModel().selectedItemProperty().addListener(
        (observable, oldValue, newValue) -> {
            if (newValue != null && newValue.getValue() instanceof Contract) {
                mvcInsuranceModel.setInitialContract((Contract) newValue.getValue());
                mvcInsuranceModel.setChangedContract(null);
            }
        });
```

Änderungen in der ContractTreeView

- Gleichzeitig registriert die `ContractTreeView` im Konstruktor einen `ChangeListener` für die Property `contractChangedProperty`, da in diesem Fall auch der Inhalt der View aktualisiert (`updateView`) werden muss.

Registieren des ChangeListener
hier als anonyme innere Klasse gelöst

```
public ContractTreeView(MVCInsuranceModel mvcInsuranceModel) {
    this.mvcInsuranceModel = mvcInsuranceModel;
    this.mvcInsuranceModel.
          changedContractPropertyProperty().addListener(new ChangeListener<Contract>() {
        @Override
        public void changed(ObservableValue<? extends Contract> observable,
                            Contract oldValue,
                            Contract newValue) {
            if (oldValue == null || (newValue != null && !oldValue.equals(newValue))) {
                updateView();
            }
        }
    });
}
```

oldValue: der Wert, der vor der Änderungen in der Property stand
newValue: der Wert, der neu gesetzt werden soll

Änderung in der `ContractDetailView`

- Die View registriert im Konstruktor einen `ChangeListener` für die `initialContractProperty` des Modells.
- Ändert sich der Wert (getriggert durch das Setzen bei Selektion in der `TreeView`) werden die View-Inhalte aktualisiert (`update`).

```
public ContractDetailView(MVCInsuranceModel mvcInsuranceModel) {
    this.mvcInsuranceModel = mvcInsuranceModel;
    this.initialize();
    if (mvcInsuranceModel.getInitialContract() != null) {
        update(mvcInsuranceModel.getInitialContract());
    }
    this.mvcInsuranceModel.initialContractProperty().addListener(
          (observable, oldValue, newValue) -> {
        mvcInsuranceModel.setChangedContract(null);
        if (newValue == null) {
            clear();
        }
        if (oldValue == null || !oldValue.equals(newValue)) {
            update(newValue);
        }
    });

}
```

Registieren des ChangeListener hier als Lambda-Ausdruck gelöst

Änderung in der ContractDetailView

- Das Speichern der Änderungen bewirkt das Setzen der contractChangedProperty und triggert die Aktualisierung in der ContractTreeView und der ContractScatterView

Kopien von Objekten, damit Änderungen in den Properties erkannt werden.

```
saveButton.setOnAction((ActionEvent e) -> {
    try {
        Contract contract;
        if (mvcInsuranceModel.getChangedContract() != null) {
            contract = mvcInsuranceModel.getChangedContract().clone();
        } else {
            contract = mvcInsuranceModel.getInitialContract().clone();
        }

          ... UNVERÄNDERT ...

          mvcInsuranceModel.setChangedContract(contract);
        }
    } catch (CloneNotSupportedException e1) {
        e1.printStackTrace();
    }
});
```

Setzen des Wertes der Property triggert die ChangeListener in der ContractTreeView / ContractScatterView.

Änderungen in der `ContractScatterView`

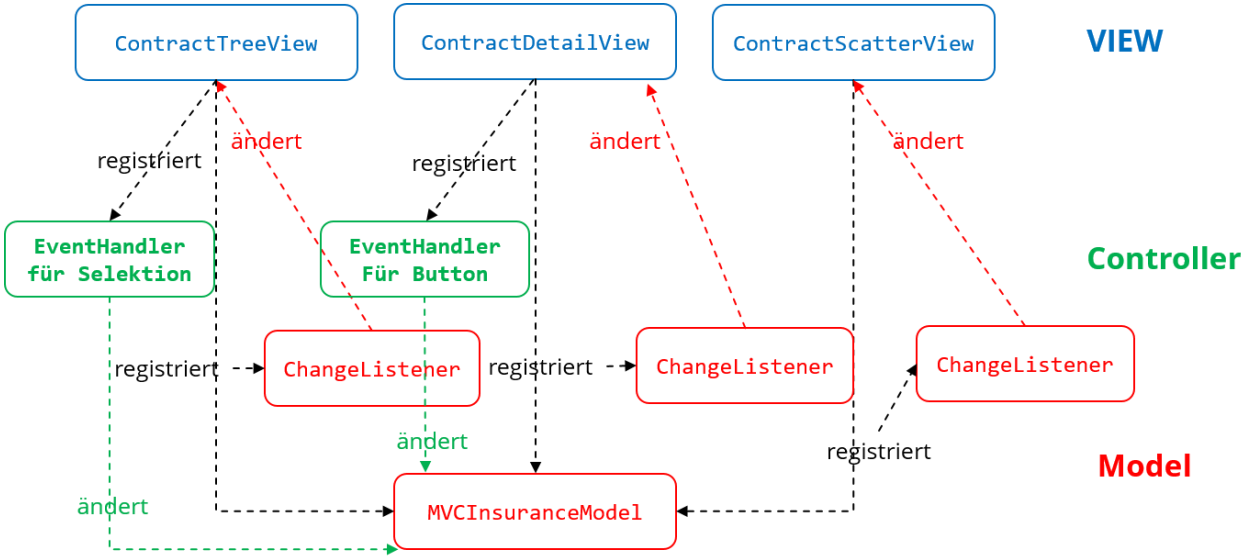
- Die View registriert im Konstruktor einen `ChangeListener` auf die `contractChangedProperty`, prüft auf legale Werte und ob eine Aktualisierung des Scatterplots notwendig ist, um dann die View zu aktualisieren (`updateView`).

```
public ContractScatterView(MVCInsuranceModel mvcInsuranceModel) {
    this.mvcInsuranceModel = mvcInsuranceModel;
    this.mvcInsuranceModel.changedContractPropertyProperty().
          addListener((observable, oldValue,  newValue) -> {
            if (oldValue == null || !oldValue.equals(newValue)) {
                if (newValue != null) {
                    int newAmount = newValue.getPaymentModel().getAmount().getEuro();
                    int newYear = newValue.getContractDate().getYear();
                    if (newAmount < minX || newAmount > maxX || newYear < minY
                        || newYear > maxY ||
                        !newValue.getContractType().equals(contractType)) {
                        contractType = newValue.getContractType();
                        updateView();
                    }
                }
            }
    });
    initialize();
}
```

Registieren des ChangeListener hier als Lambda-Ausdruck

Aktualisieren der View

Bewertung der 2‘ten Lösung



- Es existieren keine Abhängigkeiten mehr zwischen den Views.
- Die Views, ihre `EventHandler` und ihre `ChangeListener` bilden eine geschlossene Komponente.
- Aktualisierungen erfolgen immer über Änderungen im Modell.
- Mit diesem Ansatz lassen sich jetzt weitere Views an das Modell „andocken“, ohne die Implementierung der bestehenden Views ändern zu müssen.

Vertiefung

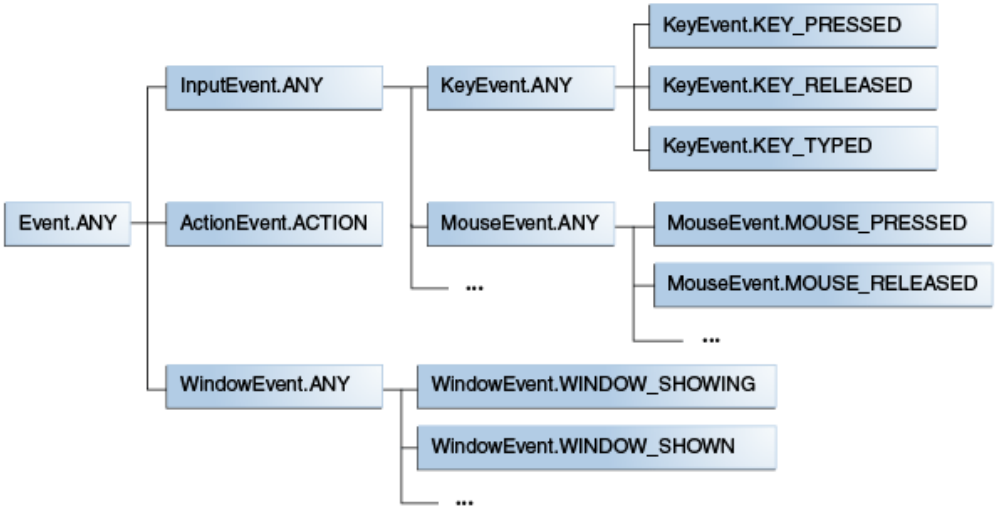
EVENTS, EVENTHANDLER UND -FILTER

Quelle: http://docs.oracle.com/javase/8/javafx/events-tutorial/events.htm#JFXED117

JavaFX Events

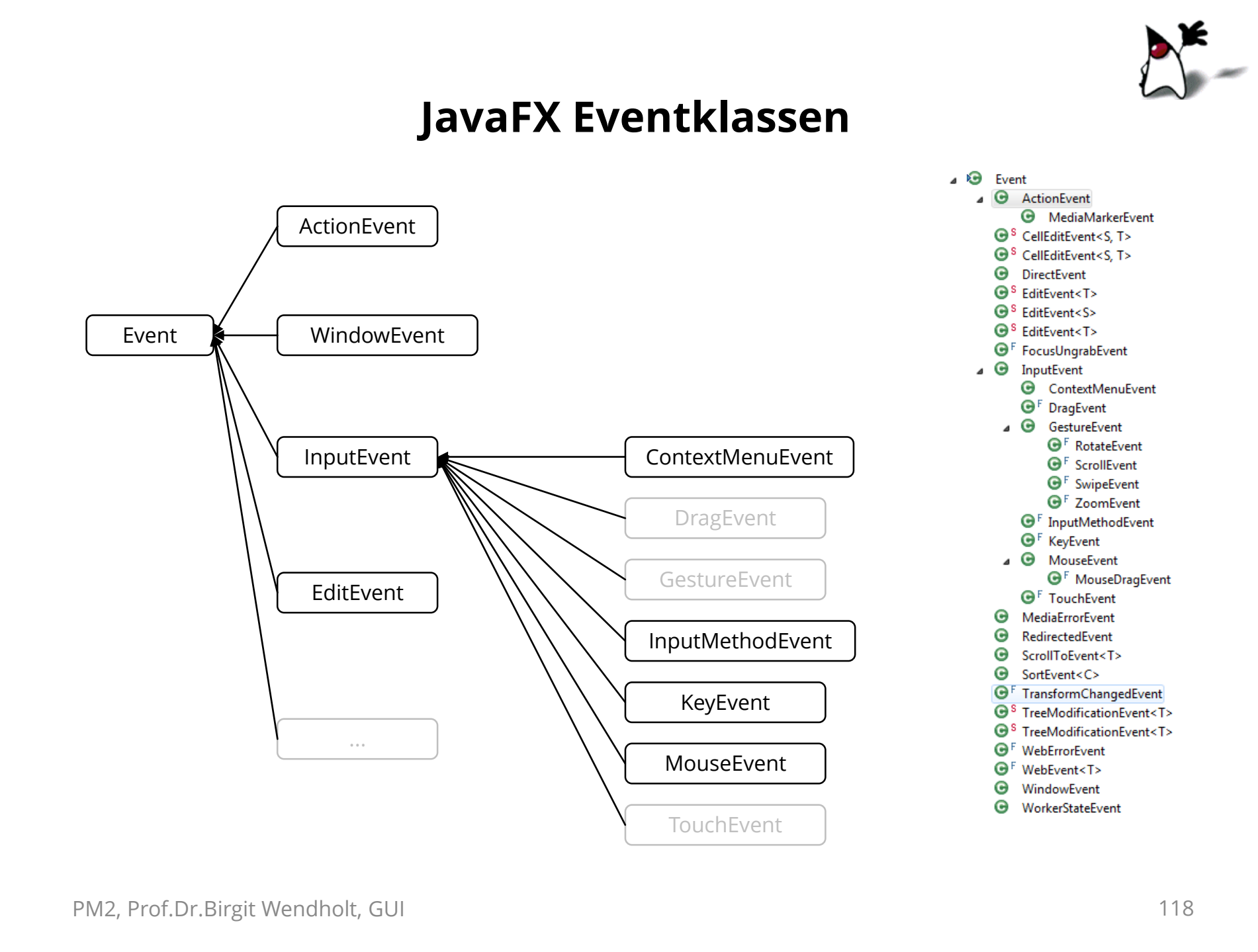
- Events sind Objekte der Subklassen von `javafx.event.Event`
- Events werden entlang einer *event dispatch chain* über die Knoten des Szenengraphen von der Quelle bis zum Ziel weitergeleitet.
- Jeder Knoten auf dem Weg zum Ziel kann das Event in einem Eventfilter konsumieren. In diesen Fällen erreicht das Event den Zielknoten nicht.
- Jedes Event hat
  - einen **Eventtyp**, der das Ereignis charakterisiert.
  - eine Quelle (**Source**), der Ursprung des Events. Die Quelle ändert sich im Verlaufe des eve*nt dispatching*.
  - ein Ziel (**Target**), der Knoten der das Event behandeln soll

Eventtypen



Quelle:
http://docs.oracle.com/javase/8/javafx/events-tutorial/img/event_type_hierarchy.gif

- Eventtypen sind hierarchisch organisiert. Jeder Eventtyp hat einen „Supertyp“, der mit der Methode `getSuperType` ermittelt wird.
- Ein Eventtyp eines Knotens ist dann immer auch vom Eventtyp seiner Elternknoten.
- Wird z.B. der Eventtyp `Event.ANY` erwartet, werden alle Subtypen behandelt. (**Anmerkung**: das funktioniert nicht zuverlässig)
- Die Sub-Supertyp Beziehung der Eventtypen ist keine Vererbungsbeziehung zwischen Klassen!
- Eventtypen sind als Klassenkonstanten der Eventklassen implementiert.



JavaFX Eventklassen

Event Targets

- Objekte der Klassen, die das Interface `EventTarget` implementieren, sind zulässige Ziele für Events.
- Die Methode `buildEventDispatchChain` des Interfaces `EventTarget` ist die Grundlage für die Auslieferung eines Events an das `Target`.
- Die JavaFX Klassen `Node`, `Scene`, `Window` implementieren das Interface `EventTarget`.

➔ Alle Elemente eines JavaFX GUI's können Events empfangen.

➔ GUI Implementierungen behandeln nur Events und müssen sich um die Implementierung der *event dispatch chain* **nicht** kümmern.

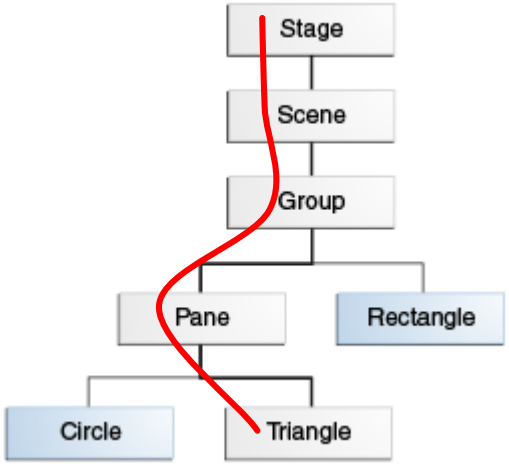
Prozess der Eventauslieferung

1. **Auswahl des Targets auf Basis von Regeln**: (z.B.)
   - Tasten-Events → Element, das den Fokus hat
   - Maus-Events → Element, an der Position des Cursors
2. **Berechnung der Route**:
   - Die Route ist der Pfad von der `Stage` über die `Scene` entlang des Szenengraphen bis zum Target.
3. **Fangen und Behandeln von Events (capturing)**:
   - Auslieferung von der Wurzel entlang der Route in Richtung Target
   - Knoten mit Event-Filter: Nach Anwenden des Filters weiterleiten
   - Knoten ohne Event-Filter: Event weiterleiten
   - Event erreicht das Target, wenn kein Filter es frühzeitig konsumiert.
4. **Zurückverteilen von Events (bubbling)**:
   - Event erreicht das Target → Event wird entlang der Route zurück zur Wurzel geschickt
   - Knoten mit Event-Handler: Nach Anwenden des Handlers weiterleiten
   - Knoten ohne Event-Handler: Event weiterleiten
   - Ende des Prozesses: Event wurde konsumiert oder erreicht die Wurzel

Veranschaulichung

Beispiel Applikation, in der das Dreieck mit einem Mausklick selektiert wird.

Quelle: http://docs.oracle.com/javase/8/javafx/events-tutorial/img/node_image.png



Route für die Beispielapplikation.

Quelle: http://docs.oracle.com/javase/8/javafx/events-tutorial/img/dispatch_chain.png

Event-Handling

- Interface: `EventHandler`; Methode: `handle`
- *Event-Filter* und *Event-Handler* implementieren gleiches Interface
- **Event-Filter**:
  - Aufruf in der Capturing-Phase
  - registriert mit `addEventFilter`
- **Event-Handler**:
  - Aufruf in der Bubbling-Phase
  - registriert mit `addEventHandler`
- Konsumieren von Events (Methode `consume`) beendet das Eventhandling.

Registrieren von Event-Handlern oder -Filtern

- Das Codebeispiel zeigt das Registrieren eines Event-Filters und eines Event-Handlers (Source in `v16-GUI-JavaFXStyle-HelloWorldWithEventHandler`)

```
/* add event filter */
btn.addEventFilter(MouseEvent.MOUSE_CLICKED,new EventHandler<MouseEvent>() {
    @Override
    public void handle(MouseEvent event) {
        System.out.println("Mouse clicked during capturing phase");
        hello.setText(btn.getText());
    }
});
/* add event handler */
btn.addEventHandler(MouseEvent.MOUSE_CLICKED,new EventHandler<MouseEvent>() {
    @Override
    public void handle(MouseEvent arg0) {
        System.out.println("Mouse clicked during bubbling phase");
        hello.setText(btn.getText());
    }
});
```

Event-Filter:
Event-Typ
EventHandler Objekt

Event-Handler:
Event-Typ
EventHandler Objekt

Registrieren von Event-Handlern mit Komfortmethode

- Das Codebeispiel zeigt das Registrieren eines Event-Handlers mit einer Komfortmethode (Source in `v11-GUI-JavaFXStyle-HelloWorldWithEventHandler`)

```
Button btn = new Button("Hello World");
Label hello = new Label();
VBox root = new VBox();
root.getChildren().addAll(btn,hello);
/* add listener */
btn.setOnMouseClicked(new EventHandler<MouseEvent>() {
    @Override
    public void handle(MouseEvent arg0) {
        hello.setText(btn.getText());
    }
});
```

In hello soll bei Mausklick auf btn der Text von btn angezeigt werden.

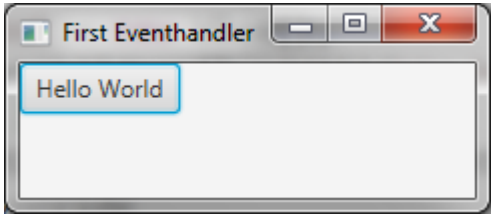
Komfort Methode zum Registrieren eines Event-Handlers für einen Eventtyp

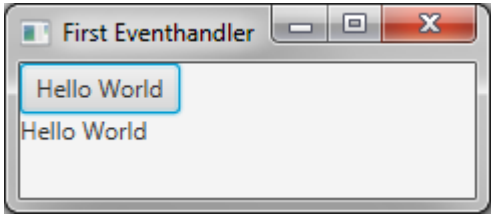
Typ EventHandler parametrisiert mit der Eventklasse

Behandlung des Events / Füllen des Labeltextes

Registrieren von Event-Handlern mit Komfortmethode

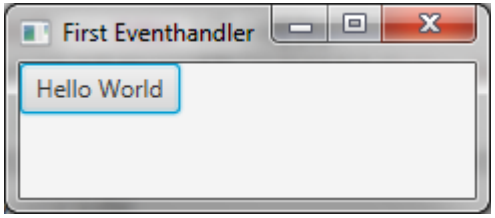
```
Button btn = new Button("Hello World");
Label hello = new Label();
VBox root = new VBox();
root.getChildren().addAll(btn,hello);
/* add listener */
btn.setOnMouseClicked(new EventHandler<MouseEvent>() {
    @Override
    public void handle(MouseEvent arg0) {
        hello.setText(btn.getText());
    }
});
```

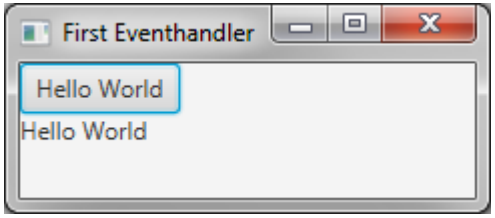




Registrieren von Event-Handlern mit Komfortmethode und lambda-Ausdruck

```
Button btn = new Button("Hello World");
Label hello = new Label();
VBox root = new VBox();
root.getChildren().addAll(btn,hello);
/* add listener */
btn.setOnMouseClicked((MouseEvent me) -> {
    hello.setText(btn.getText());
});
```





Registrieren mit Komfortmethoden

- Komfortmethoden = Kurzformen zum Registrieren von Event-Handlern in JavaFX
- Syntax der Komfortmethoden:

```
setOn<event-type>(EventHandler<? super event-class> value )
setOn<event-type>((event-class event) -> {…} );
setOn<event-type>(event -> {…} );
```

- Am Beispiel des Event-Typs `MouseEvent.MOUSE_CLICKED`:

```
setOnMouseClicked(new EventHandler<MouseEvent>() {…});
```

  realisiert mit einer anonymen inneren Klasse

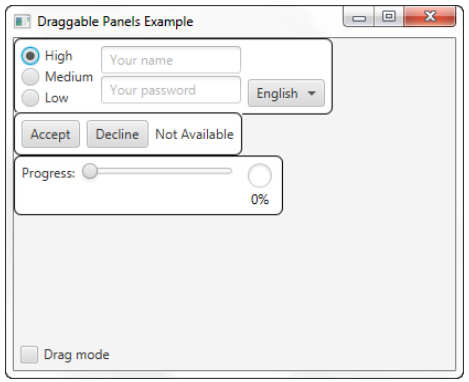
- Am Beispiel des Event-Typs `MouseEvent.MOUSE_CLICKED`:

```
setOnMouseClicked(e -> {…});
```

  realisiert mit einem lambda- Ausdruck

Einsatz von Event-Filtern während der Capturing Phase

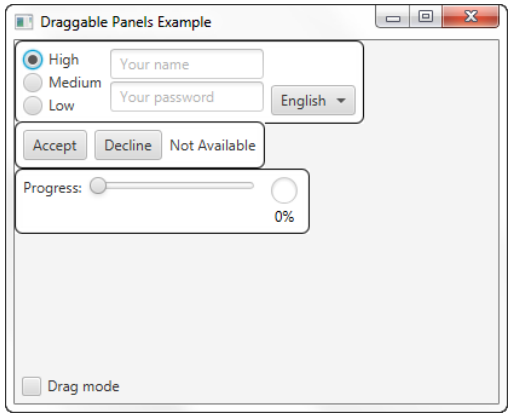
- Event-Filter können an allen Knoten der Event-Route registriert sein.
- Es können Filter für mehrere Event-Typen registriert sein.
- Es können Sub- und Supertypen von Events gleichzeitig verwendet werden. Die Filter der Subtypen werden immer vor den Filtern der Supertypen abgearbeitet.
- Parent Nodes können mit Event-Filtern
  - ein gemeinsames Verhalten für alle Kindknoten erzwingen
  - das Behandeln von Events durch Kindknoten unterbinden (interception)
- Beispiel: Draggable Panel
  - **Drag-Mode aktiv**:
    - alle Input-Events werden für die Kindknoten blockiert
    - Drag-Events werden vom Parent behandelt
  - **Sonst**:
    - Input-Events werden an die Kindknoten weitergegeben



Einsatz von Event-Filtern: Draggable Panel Beispiel (1)

- Parent Nodes können mit Event-Filtern
  - ein gemeinsames Verhalten für alle Kindknoten erzwingen
  - das Behandeln von Events durch Kindknoten unterbinden (interception)

- Beispiel: Draggable Panel
  - **Drag-Mode aktiv**:
    - alle Input-Events werden für die Kindknoten blockiert
    - Drag-Events werden vom Parent behandelt
  - **Sonst**:
    - Input-Events werden an die Kindknoten weitergegeben

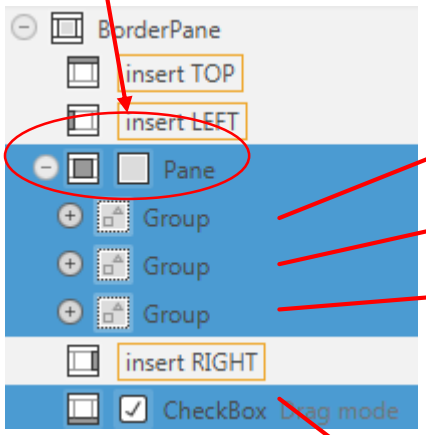


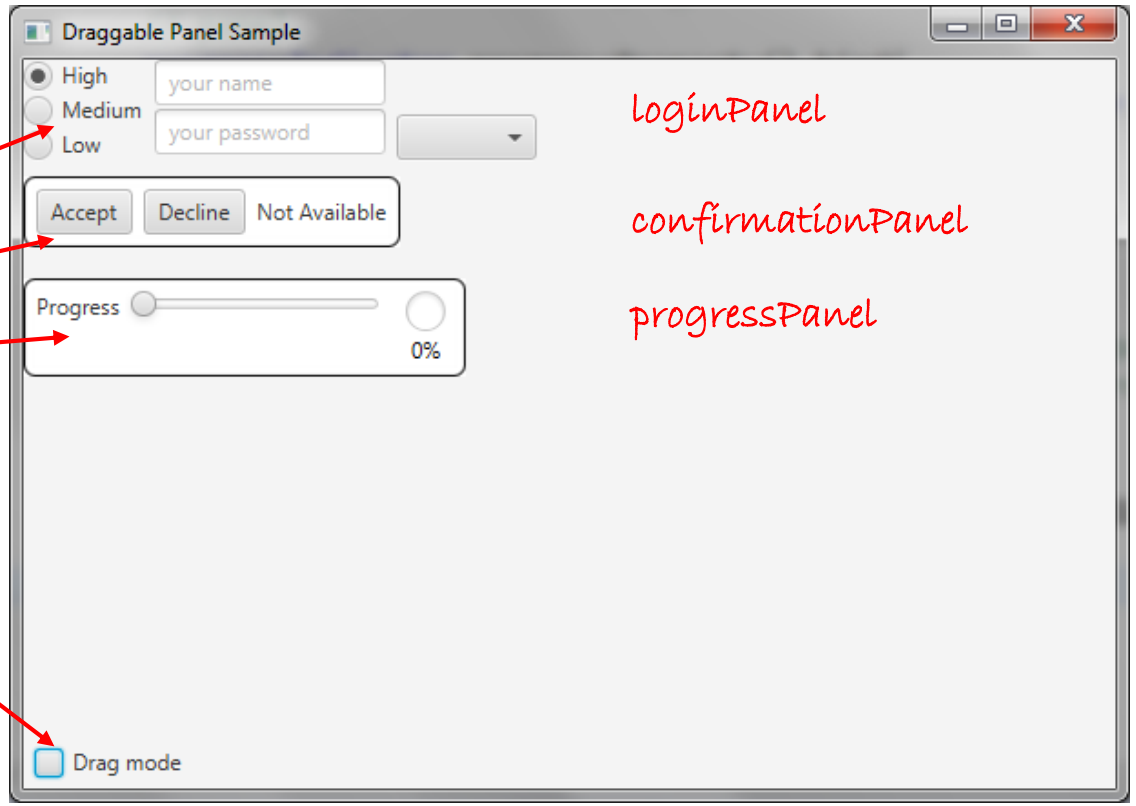
- (Quelle: http://docs.oracle.com/javase/8/javafx/events-tutorial/filters.htm).
- Ursprünglicher Sourcecode (http://docs.oracle.com/javase/8/javafx/events-tutorial/draggablepanelsexamplejava.htm#CHDHIGHI) für das Erstellen mit SceneBuilder und unter Verwendung von Java 8 Sprachfeatures modifiziert (-> v11-GUI-JavaFX-EventFilter-DraggablePanel)

Einsatz von Event-Filtern: Draggable Panel Beispiel (2)

3 Groups als Wrapper für die Panel loginPanel, confirmationPanel, progressPanel.
Jede Group wird zum Interceptor von Input-Events und implementiert Filter für das Draggen der Panel

parent der Groups





loginPanel

confirmationPanel

progressPanel

Checkbox zum Ein-/Aus-schalten des Dragmodus

Einsatz von Event-Filtern: Draggable Panel Beispiel (3)

- DragMode aktiv (`dragModeActiveProperty.get()==true`):
  - Blockieren aller Input-Events für die Kindknoten durch Konsumieren der Events in einem Event-Filter für den allgemeinen Event-Typ `InputEvent.ANY`
  - Bei `MouseEvent.MOUSE_PRESSED`: Aufzeichnen der Position eines Panels und der Position der Maus in einem `DragContext` Objekt
  - Bei `MouseEvent.MOUSE_DRAGGING`: Aktualisieren der Position der Panels mit den Daten der `DragContext` Objekts und der aktuellen Mausposition.
- DragMode inaktiv:
  - Weitergabe aller Events an die Kindknoten

Einsatz von Event-Filtern: Draggable Panel Beispiel (4)
Konsumieren von `InputEvent.ANY`

```
/* if dragging is active block all input events for child nodes */
wrapperGroup.addEventFilter(InputEvent.ANY, (InputEvent e) -> {
    if (dragModeActiveProperty.get()) {
        e.consume();
    }
});
```

Einsatz von Event-Filtern: Draggable Panel Beispiel (5)
Filter für `MouseEvent.MOUSE_PRESSED`

```
DragContext dragContext = new DragContext();

/*
 * if dragging is active and mouse is pressed, remember initial mouse
 * cursor coordinates and node position
 */
wrapperGroup.addEventFilter(
        MouseEvent.MOUSE_PRESSED,
        (MouseEvent e) -> {
            if (dragModeActiveProperty.get()) {
                dragContext.initialTranslateX = wrapperGroup
                        .getTranslateX();
                dragContext.initialTranslateY = wrapperGroup
                        .getTranslateY();
                dragContext.mouseAnchorX = e.getX();
                dragContext.mouseAnchorY = e.getY();
            }
        });
```

Einsatz von Event-Filtern: Draggable Panel Beispiel (6)
Filter für `MouseEvent.MOUSE_DRAGGING`

```
/* if dragging is active and mouse is dragging update node position */
wrapperGroup.addEventFilter(MouseEvent.MOUSE_DRAGGED,
        (MouseEvent e) -> {
            if (dragModeActiveProperty.get()) {
                wrapperGroup
                        .setTranslateX(dragContext.initialTranslateX
                                + e.getX() - dragContext.mouseAnchorX);
                wrapperGroup
                        .setTranslateY(dragContext.initialTranslateY
                                + e.getY() - dragContext.mouseAnchorY);
            }
        });
```

Hinweise zu den Beispielen für Controls in JavaFX

- Die Beispiele stammen aus dem Oracle JavaFX Tutorial Kapitel: Using JavaFX UI Controls (http://docs.oracle.com/javase/8/javafx/user-interface-tutorial/ui_controls.htm#JFXUI336)
- Für jedes GUI findet sich dort ein Beispielprogramm, dass nur aus Java-Code besteht und keine `fxml` Dateien enthält und damit auch nicht den `SceneBuilder` verwendet.
- Die Beispielprogramme wurden von mir für den `SceneBuilder` und für das FXML Programmiermodell adaptiert.
- Der adaptierte Quelltext wird auf den jeweiligen Folien referenziert und wird mit diesen Folien ausgeliefert.
- In der Vorlesung werden die GUI Beispiele Life demonstriert.
- Nur die **wesentlichen** Aspekte finden sich auf den Folien!

Zusammenfassung 1

- GUI‘s bestehen im Wesentlichen aus
  - grafischen Komponenten den **UI-Controls**
  - einer **Render-Komponente** für Zeichnen von UI-Controls
  - **Containern** (Gruppen) für das Gruppieren von Elementen
  - **Layout-Container** für das Layout von Komponenten über relative Größen
  - Event-Handling Mechanismen um u.a. auf Benutzerinteraktionen reagieren zu können

- GUI‘s separieren Daten, Anzeige und Manipulation auf Basis des **Model View Controller Pattern (MVC)**.

Zusammenfassung 2

- **JavaFX** verwendet einen **Szenen-Graphen** für die Repräsentation eines GUI's. (Container können Container und UI-Controls enthalten.)
- Mit die Knoten des Szenen-Graphen lassen sich alle UI-relevanten Daten verknüpfen, die für alle Kind-Knoten gelten, wenn diese nicht überschrieben werden.
- Die Konstruktion des Szenen-Graphen erfolgt
  - **programmatisch** und/oder
  - deklarativ auf Basis von **FXML**
- Für die Konstruktion des Szenen-Graphen für JavaFX hat sich der **SceneBuilder** als Tool etabliert. In dem SceneBuilder lassen sich alle statischen Eigenschaften der Knoten des Szenen-Graphen modellieren.
- Die dynamischen Eigenschaften, wie z.B. das Reagieren auf Events, wird in JavaFX mittels **EventHandler / EventListener-Klassen** realisiert.
- JavaFX stellt ein große Menge an **UI-Controls** zur Verfügung. Die meisten sind in den SceneBuilder integriert.
- Das Verhalten einiger JavaFX-Controls kann programmatische über sogenannte **Cell-Factories**, die spezielle Implementierungen von Cell-Editoren liefern, adaptiert werden.

Zusammenfassung 3

- **Eventhandling** erfolgt in JavaFX mittels der EventHandler, die auf Ereignisse registriert werden.
- Zu den wesentlichen Ereignissen gehören
  - **Aktionen** (ActionEvent) wie z.B. das Drücken eines Knopfes oder die Selektion eines Listeneintrages
  - **Eingaben** (InputEvent) über z.B. Tastatur oder Maus
  - **Window-Events** wie z.B. das Schließen oder Ikonifizieren eines Fensters
  - **Change-Events** wie z.B. das Ändern eines Wertes oder das Hinzufügen von Elementen zu einer Sammlung von Objekten
- Eventhandling erfolgt entlang der **Event-Dispatch-Chain** von der Wurzel des Szenen-Graphen zu dem Ziel-Objekt in 2 Richtungen:
  - von der Wurzel zum Ziel-Objekt: **Capturing**
  - vom Ziel-Objekt zur Wurzel: **Bubbleing**
- Eventhandler für
  - Handler für das Capturing werden mit addListener registriert
  - Handler für Bubbleing werden mit addHandler registiert
- Eventhandler für ChangeEvents setzen entweder **JavaFX-Bean, ObservableList oder ObservableMap** Datentypen voraus
- Mittels **Bindings** lassen sich Änderungen in der GUI an Änderungen von Objekten knüpfen sowohl uni- als auch bidirektional.