Lösungen zu den Pflichtaufgaben GUI

1. Erweitern des Workshops um ein Model [PU]

Siehe Dokument *Lösungen zu den Übungen GUI*.

2. Die Übungsapplikation – ROI Calculator

Siehe Dokument *Lösungen zu den Übungen GUI*.

3. Erstellung des ROI-Calculators mittels Object-SceneGraph

Siehe Dokument *Lösungen zu den Übungen GUI*.

4. Erstellen des ROI-Calculators mit FXML [PA]

Nachdem Sie sich bis hierher einige Gedanken zur Verwendung der Container (Panes) gemacht haben und praktische Erfahrung mit dem Aufbau des User Interfaces gesammelt haben, sind Sie nun bereit für die Umsetzung der Benutzeroberfläche mit FXML unter Verwendung des Scene Builder.

4.1. Basis

Eine Musterlösung für diese Aufgabe finden Sie im Lösungsverzeichnis unter `FXML-Calculator`.

4.2. Umsetzung

a. Verifizieren Sie die Projektkonfiguration in `build.gradle` für FXML.

   Siehe Aufgabe 3.a). Hier muss das `javafx.fxml` Modul zwingend eingebunden werden.

b. Erstellen Sie den SceneGraph mit Hilfe des SceneBuilder als FXML-Spezifikation, laden diesen und binden ihn im Hauptfenster ein.

   Die `MainWindow.fxml` Datei finden Sie im Resourcen-Ordner

c. Erstellen Sie die Controllerklasse und verknüpfen Sie die Controls und Actions mit dem FXML-SceneGraph

Im Scene-Builder erstellen Sie die gleiche (oder eine bessere Version ihres Scene-Graphs. Die Verknüpfung der Controller-Felder über die FXML-Annotationen erlaubt es dem FXML-Loader, diese Felder direkt mit den Referenzen auf die Scene-Graph Nodes zu befüllen, sodass der Controller direkt auf die Nodes zugreifen kann.

In der anderen Richtung können in FXML automatisch EventHandler erstellt und, über die FXML-Annotation auf Methoden, mit diesen verknüpft werden. Das heißt, beim Auslösen des Events wird diese verknüpfte Methode aufgerufen.

d. Überlegen Sie sich, wie das vorhandene Datenfeld `resultBound` der Hilfsklasse `ValueHandler` verwenden werden kann, um die View (z.B. Textfeld im MainWindow) mittels Observer-Pattern zu aktualisieren, damit `ValueHandler` selbst keine Referenz auf die View oder den Controller benötigt.
   - Welche Hilfsmittel bietet JavaFX dazu an?
   - Passen Sie ihre Anwendung entsprechend an.

`ValueHandler` bietet die eigentliche Berechnungslogik und könnte im MVC-Kontext zum Model gezählt werden.
In `resultBound` wird das Resultat (oder die Fehlermeldung) abgelegt und kann vom Controller nach der Berechnung abgefragt und ausgegeben werden, was mit dem MVC-Pattern absolut zulässig ist. Bei nur einem Controller und wenn das Modell von Aussen nicht verändert wird, funktioniert das auch tadellos. Sobald jedoch mehrere Controller bzw. Views ins Spiel kommen, muss das Modell diese alle über eine Änderung informieren können.

Dies sollte mit Hilfe des Observer-Patterns erfolgen. Wie in der FXML-WordCloud Übung könnte das mittels eigenen Observer- / Observable-Klassen umgesetzt werden.

Hier sollen jedoch die JavaFX internen Mechanismen `javafx.beans.value.ChangeListener` in Kombination mit `javafx.beans.value.ObservableValue`-Elementen verwendet werden.
Die entsprechenden Datenfelder werden als `javafx.beans.value.ObservableValue` umgesetzt, welche bei einer Änderung des Wertes alle mittels `addListener()` registrierten `ChangeListener`-Objekte informieren (Methode `changed()` aufrufen).

Weil das in JavaFX sehr häufig Verwendung findet, werden für die meisten Basistypen sogenannte Property-Klassen bereitgestellt, welche `ObservableValue` implementieren.
Für nahezu alle Attribute von JavaFX-Komponenten werden Properties verwendet. Das heisst, es kann ein ChangeListener registriert werden, welcher auf die Änderung reagiert (z.B. Fensterhöhe, Textfeldinhalt, ...). Properties bieten zudem noch die Möglichkeit, diese mittels `bind` bzw. `bindBidirectional` miteinander zu verknüpfen, d.h. es wird ein automatisch ein ChangeListener eingerichtet, welche das verbundene Property aktualisiert.

In `ValueHandler` wurde deshalb `resultBound` als `StringProperty` erstellt. Das heißt, der Controller kann einen `ChangeListener` registrieren, welcher die View (Textfield) automatisch aktualisiert, sobald das Resultat ändert. Zum Beispiel:

```
public void setValueHandler(ValueHandler valueHandler) {
    this.valueHandler = valueHandler;
    valueHandler.resultBoundProperty().addListener(new ChangeListener
<String>() {
        @Override
        public void changed(ObservableValue<? extends String> observable,
String oldValue, String newValue) {
            results.setText(newValue);
        }
    });
}
```

Oder noch einfacher, sie können `resultBound` direkt mit dem Inhalt des Resultat-Textfelds verbinden, indem Sie beim Initialisieren der Hilfsklasse die zwei Properties miteinander verbinden:

```
public void setValueHandler(ValueHandler valueHandler) {
    this.valueHandler = valueHandler;
    results.textProperty().bind(valueHandler.resultBoundProperty());
}
```

Nachteil dieser Lösung ist, dass das Modell nicht mehr unabhängig vom User-Interface ist, da JavaFX-Klassen verwendet werden. Obwohl der Mechanismus unabhängig vom GUI funktionieren würde, müssten die Libraries eingebunden werden.

e. Fügen Sie im MainWindow einen weiteren Button und zugehörige Aktion (e.g. `openResultWindow`) ein. Diese soll ein zusätzliches einfaches Resultatfenster öffnen.
   - Als Vorlage finden Sie im Praktikumsverzeichnis bereits eine FXML-Datei (`ResultWindow.fxml`) und einen leeren Controller.
   - Erweitern Sie den Controller von ResultWindow so, dass auch hier das Resultat angezeigt wird, sobald es in `ValueHandler` neu gerechnet wird.
   - Verwenden Sie in beiden Views das gleiche `ValueHandler` Objekt.

Wie in der Vorlesung gezeigt, wird für die neue View der entsprechende Scene-Graph aus der `ResultWindow.fxml` Datei geladen und kann in einer neuen Scene in einer neuen Stage (Window) geöffnet werden. Mit dem Laden der FXML-Datei wird auch eine Instanz von `ResultWindowController` erzeugt. Damit dieser auch die Resultate anzeigen kann, muss das Model gesetzt werden. Auch hier muss natürlich dann das TextFeld aktualisiert werden, wenn das Resultat im `ValueHandler` sich ändert. Das heißt, es muss ein `ChangeListener` registriert oder das Property mit dem Textfeld verbunden werden (siehe oben).
Wenn auch der Status der Berechnung (Farbe des Rahmens) aktualisiert werden soll, müsste dieser auch im Modell verfügbar sein. Als Variante mit dem aktuellen `ValueHandler` kann bei einer Aktualisierung des Resultats vom `ResultWindowController` mit `valueHandler.areValuesOk()` der Status abgefragt und die Rahmenfarbe entsprechend gesetzt werden (siehe Methode `updateResult()`).

In der Musterlösung wurden zwei Varianten für die neue View umgesetzt. Mit dem einen Knopf wird der Inhalt des aktuellen Fensters ersetzt. Mit dem anderen `Button` wird ein separates Fenster geöffnet, welches parallel aktualisiert wird.