Selbststudiums-Aufgabe 11 – Lösung

Layers, MVC

1. Selbststudium zu Layers

Lösungen s. Unterrichtsunterlagen

2. Micro MVC

```
private void initialize() {
   op1.textProperty().addListener(
      (observable, oldValue, newValue) -> value1.set(intFromString(newValue, 10))
   );
   value1.addListener(
      (observable, oldValue, newValue) -> op1.setText(stringFromInt(newValue, 10))
   );
   op2.textProperty().addListener(
      (observable, oldValue, newValue) -> value2.set(intFromString(newValue, 10))
   );
   value2.addListener(
      (observable, oldValue, newValue) -> op2.setText(stringFromInt(newValue, 10))
   );
   sum.textProperty().bind(value1.add(value2).asString());
   op1Hex.textProperty().addListener(
      (observable, oldValue, newValue) -> value1.set(intFromString(newValue, 16))
   );
   value1.addListener(
      (observable, oldValue, newValue) -> op1Hex.setText(stringFromInt(newValue, 16))
   );
   op2Hex.textProperty().addListener(
      (observable, oldValue, newValue) -> value2.set(intFromString(newValue, 16))
   );
   value2.addListener(
      (observable, oldValue, newValue) -> op2Hex.setText(stringFromInt(newValue, 16))
   );
   sumHex.textProperty().bind(value1.add(value2).asString("%x"));
}
```

s. auch separates ch.fhnw.swa.mvc.adder_hex_solution.zip

3. Makro MVC

Verteilt man den vorhandenen Code der einfachen Version auf drei Komponenten (Klassen), erhält man:

```
public class Model {
   public final IntegerProperty value1 = new SimpleIntegerProperty();
   public final IntegerProperty value2 = new SimpleIntegerProperty();
   public final NumberBinding sumVal = value1.add(value2);
}

public class View {
   private Model model = null;
   @FXML private TextField op1;
   @FXML private TextField op2;
   @FXML private TextField sum;

   public void setModel(Model model) {
      if (this.model != null) throw new IllegalStateException();
      this.model = model;
      model.value1.addListener(
         (observable, oldValue, newValue) -> op1.textProperty().set(newValue.toString())
      );
      model.value2.addListener(
         (observable, oldValue, newValue) -> op2.textProperty().set(newValue.toString())
      );
      sum.textProperty().bind(model.sumVal.asString());
   }
```

```
   public Model model() { return model; }
   public TextField op1() { return op1; }
   public TextField op2() { return op2; }
   public TextField sum() { return sum; }
}

public class Controller {
   public Controller(final View view) {
      Model m = view.model();
      view.op1().textProperty().addListener(
         (observable, oldValue, newValue) -> m.value1.set(intFromString(newValue, 10))
      );
      view.op2().textProperty().addListener(
         (observable, oldValue, newValue) -> m.value2.set(intFromString(newValue, 10))
      );
   }

   private int intFromString(String s, int base) {
      try { return Integer.parseInt(s, base); }
      catch (NumberFormatException e) { return 0; }
   }
}
```

Betrachtet man bei dieser Lösung die Importe der Klasse `Model`, sieht man, dass die Model-Implementierung von der JavaFX-Technologie abhängig ist:

```
import javafx.beans.binding.NumberBinding;
import javafx.beans.property.IntegerProperty;
import javafx.beans.property.SimpleIntegerProperty;
```

Für ein reines Domain Model (ausschliesslich für Datenhaltung und Business Logic verantwortlich) ist eine solche Abhängigkeit von einem bestimmten UI-Toolkit nicht gewünscht. Man müsste in diesem Fall die Klasse `Model` unabhängig von `javafx`-Elementen programmieren; z.B. mit einem eigenen `Listener`-Mechanismus:

```
public final class Model {
   private int value1, value2, sumVal;
   private List<Listener> listeners = new ArrayList<>();

   public interface Listener { public void modelStateChanged(); }

   public void addListener(Listener l) { listeners.add(l); }
   public void removeListener(Listener l) { listeners.remove(l); }

   public int getValue1() { return value1; }
   public void setValue1(int value1) { this.value1 = value1; update(); }
   public int getValue2() { return value2; }
   public void setValue2(int value2) { this.value2 = value2; update(); }
   public int getSumVal() { return sumVal; }

   private void update() {
      int sum = value1 + value2;
      if (sum != sumVal) {
         sumVal = sum;
         for (Listener l : listeners.toArray(new Listener[listeners.size()]))
            l.modelStateChanged();
      }
   }
}
```

Die `View`-Klasse hat dann die Verantwortung, die JavaFX-Properties mit diesem `Listener` nachzuführen:

```
public class View {
   private Model model = null;
   @FXML private TextField op1;
   @FXML private TextField op2;
   @FXML private TextField sum;
```

```
   public void setModel(Model model) {
      if (this.model != null) throw new IllegalStateException();
      this.model = model;
      model.addListener(() -> {
         op1.textProperty().set(Integer.toString(model.getValue1()));
         op2.textProperty().set(Integer.toString(model.getValue2()));
         sum.textProperty().set(Integer.toString(model.getSumVal()));
      });
   }

   public Model model() { return model; }
   public TextField op1() { return op1; }
   public TextField op2() { return op2; }
   public TextField sum() { return sum; }
}
```

Und der `Controller` muss direkt die Setter des `Models` aufrufen:

```
public class Controller {
   public Controller(final View view) {
      Model m = view.model();
      view.op1().textProperty().addListener(
         (observable, oldValue, newValue) -> m.setValue1(intFromString(newValue, 10))
      );
      view.op2().textProperty().addListener(
         (observable, oldValue, newValue) -> m.setValue2(intFromString(newValue, 10))
      );
   }

   private int intFromString(String s, int base) {
      try { return Integer.parseInt(s, base); }
      catch (NumberFormatException e) { return 0; }
   }
}
```

s. auch separates ch.fhnw.swa.mvc.adder_mvc_solution.zip

4. BidirectionalBindings mit JavaFX – Extra-Aufgabe für Interessierte