Grafische Benutzeroberflächen mit JavaFX

Jens Kosiol
Wintersemester 23/24

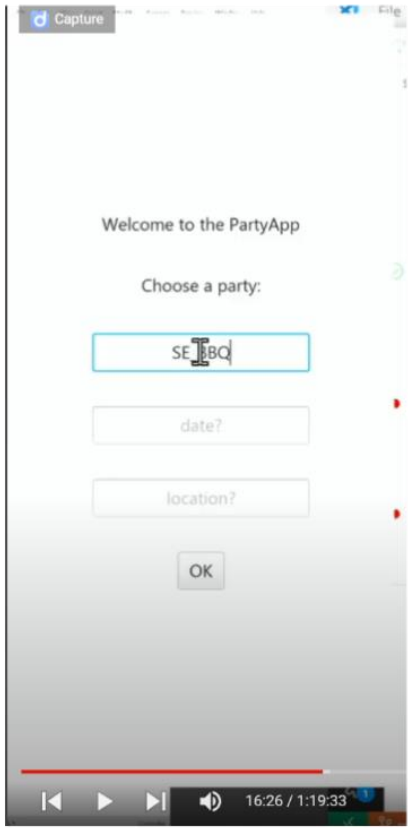
Wichtige Aspekte der GUI-Entwicklung

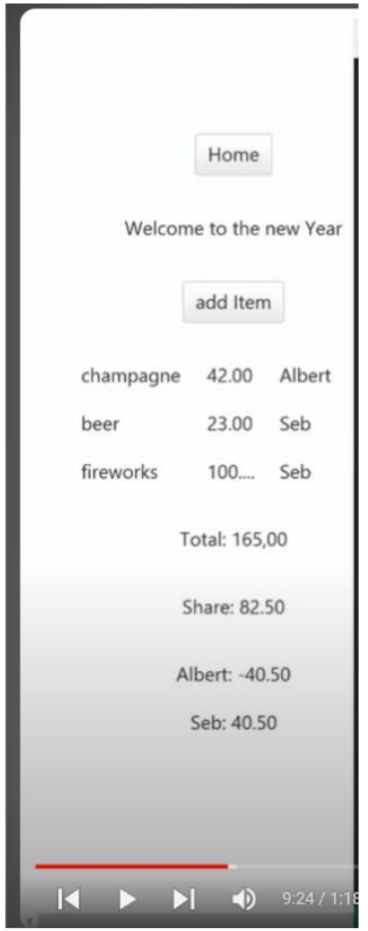
- Verwendung eines GUI-Toolkits (widget toolkit, widget library)
  - stellt Layout- und Kontrollelemente (widgets) zur Verfügung
  - managt die Interaktion mit dem Betriebssystem
- Model-View-Controller Architektur
  - Entwurfsmuster zur Organisation der Implementierung einer GUI
- Event-basierte Programmierung
  - Events (Mausbewegungen, Tastaturanschläge, Werteänderungen, …) steuern den Programmablauf und die Anzeige.

Vorlesungsbeispiel

PartyApp

- Kleine App zur Verwaltung von Feiern
- Startseite bietet Überblick über angelegte Feiern und bietet die Möglichkeit, neue Feiern anzulegen
- In einer Feier können Gäste eingetragen werden, was sie zur Party mitbringen und ihre Kosten
- Es wird eine Anzeige generiert, wer wieviel Geld erhält bzw. bezahlen muss

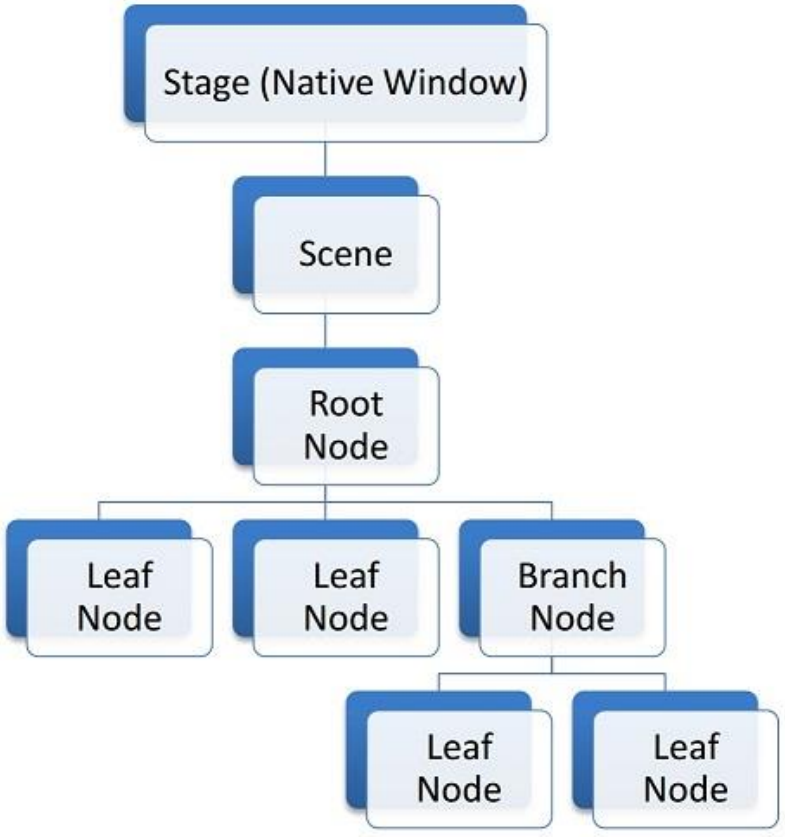




Java GUI-Toolkits und JavaFX

- Für Java stehen eine ganze Reihe von GUI-Toolkits zur Verfügung (AWT, SWT, Swing, JavaFX, …)
- JavaFX (https://openjfx.io/):
  - Framework zur Erstellung von Java-Applikationen, insbesondere solcher mit multimedialen Inhalten und GUI
  - Entwickelt seit 2007 von Oracle (inzwischen Open Source); aktuelle LTS Version: 21.0.1 (Oktober 2023)
  - Unterstützt das Trennen von Logik und Gestaltung
    - Gestaltung über FXML-Files
    - Logik als Java-Code
  - Bietet Mediensupport (Audio, Bilder, Video) und besitzt eine Webkomponente
  - Bietet ein System für Event handling
  - Dokumentation:
    - https://openjfx.io/javadoc/21/index.html (Dokumentation der API)
    - https://docs.oracle.com/javase/8/javase-clienttechnologies.htm (Dokumentation durch Oracle, aber FX 8)
    - https://fxdocs.github.io/docs/html5/ (Open Source Doku durch die Community)

Aufbau einer JavaFX Anwendung



Quelle: https://fxdocs.github.io/docs/html5/#_scene_graph

JavaFX Anwendungen sind als Bäume organisiert:

- Stages sind der Rahmen der Anwendung; sie entsprechen meist Fenstern
  - Eine `primary stage` wird beim Start von der JavaFX runtime automatisch erzeugt
  - Die `show`-Methode blendet eine Stage ein
- Eine Stage zeigt Scenes an: immer nur eine, aber welche kann zur Laufzeit gewechselt werden.
- Einer Scene ist ein Scene Graph zugeordnet: eine Baumstruktur mit den visuellen Elementen der Scene.

Aufbau Scene Graph

- Die Elemente des Scene Graph heißen Nodes.
  - Branch nodes: Knoten, die andere Knoten enthalten (können).
  - Leaf nodes: Knoten, die keine anderen Knoten enthalten können.
- JavaFX stellt verschiedene (visuelle) Komponenten, die als Knoten dienen können, zur Verfügung:
  - Controls stellen Funktionalität zur Verfügung (Button, CheckBox, TextField, …).
  - Layouts steuern die Erscheinung der Elemente, die sie enthalten (Group, Hbox, Vbox, StackPane, …).
  - Charts zur einfachen Erstellung von Charts (BarChart, PieChart, …).
  - Weitere Komponenten zur Wiedergabe von Audio, Video, Webinhalten (WebView) oder 2D und 3D Grafiken.
- Scene Graphs können im Code erstellt werden oder zum Beispiel fxml-Dateien können als Grundlage dienen.

In dieser Vorlesung benutzen wir fxml und erstellen die entsprechenden Dateien mit dem Scene Builder.

Lebenszyklus einer JavaFX Anwendung


```
import javafx.application.Application;
import javafx.stage.Stage;

public class App extends Application {

   @Override
   public void start(Stage primaryStage) throws
Exception {
      primaryStage.setTitle("Party App");
      primaryStage.setWidth(640);
      primaryStage.setHeight(480);
      primaryStage.show();
   }
}

//main-Methode ist optional
public class Main {
   public static void main(String[] args) {
      Application.launch(App.class, args);
   }
}
```

Die Klasse `Application` ist der Startpunkt für jede JavaFX Anwendung.

Lebenszyklus bei Anwendungsstart:

1. JavaFX runtime wird gestartet.
2. Eine Instanz der spezifizierten `Application`-Klasse wird erzeugt.
3. Die Methode `init()` wird aufgerufen; Standardimplementierung tut nichts (darf aber muss nicht überschrieben werden).
4. Die Methode `start(javafx.stage.Stage)` wird aufgerufen – diese Methode muss überschrieben werden!
5. Es wird gewartet, bis die Programmausführung endet:
   - `Platform.exit()` wird aufgerufen
   - Alle Fenster wurden geschlossen und `implicitExit` ist `true`.
6. Die Methode `stop()` wird aufgerufen; Standardimplementierung tut nichts (darf aber muss nicht überschrieben werden)

Bei Verwendung einer `main`-Methode muss dort die `launch`-Methode von `Application` aufgerufen werden.

Zentrale JavaFX-Methoden

Für Klasse `Stage`:

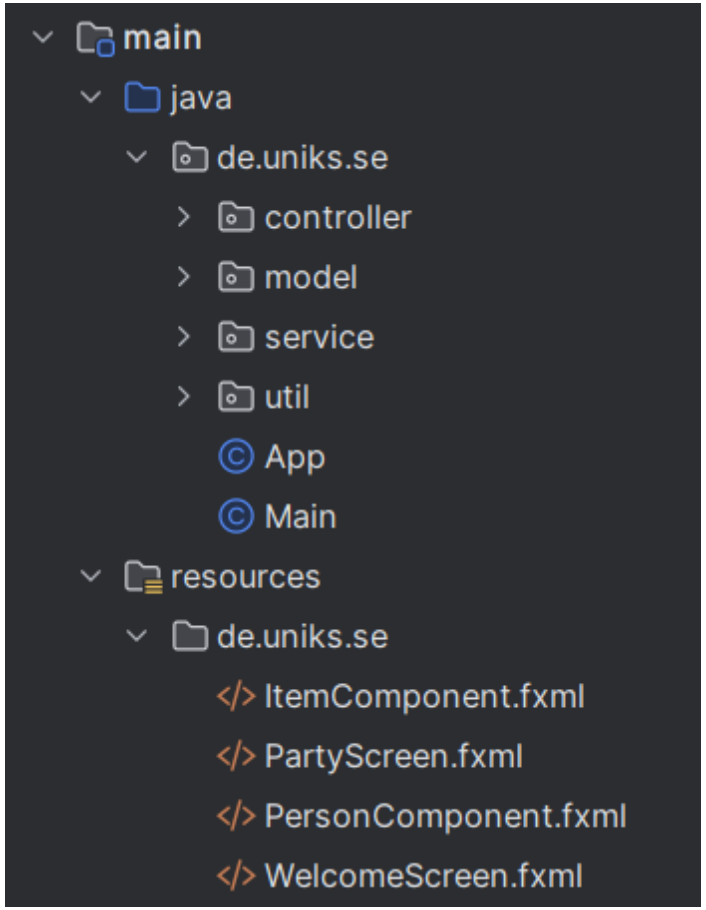
- Konstruktor `Stage()` erzeugt eine neue Instanz
- `close()` schließt eine `Stage`
- `setScene(Scene value)` spezifiziert die Szene der `Stage`
- `setTitle(String title)` setzt den Titel der `Stage`
- `show()` zeigt die Stage an
- …

Für Klasse `Scene`:

- Konstruktor `Scene(Parent root)` erzeugt eine `Scene` mit `root` als Wurzelknoten des zugehörigen Scene Graph
- `setRoot(Parent value)` setzt den Wurzelknoten des zugehörigen Scene Graph auf `value`
- …

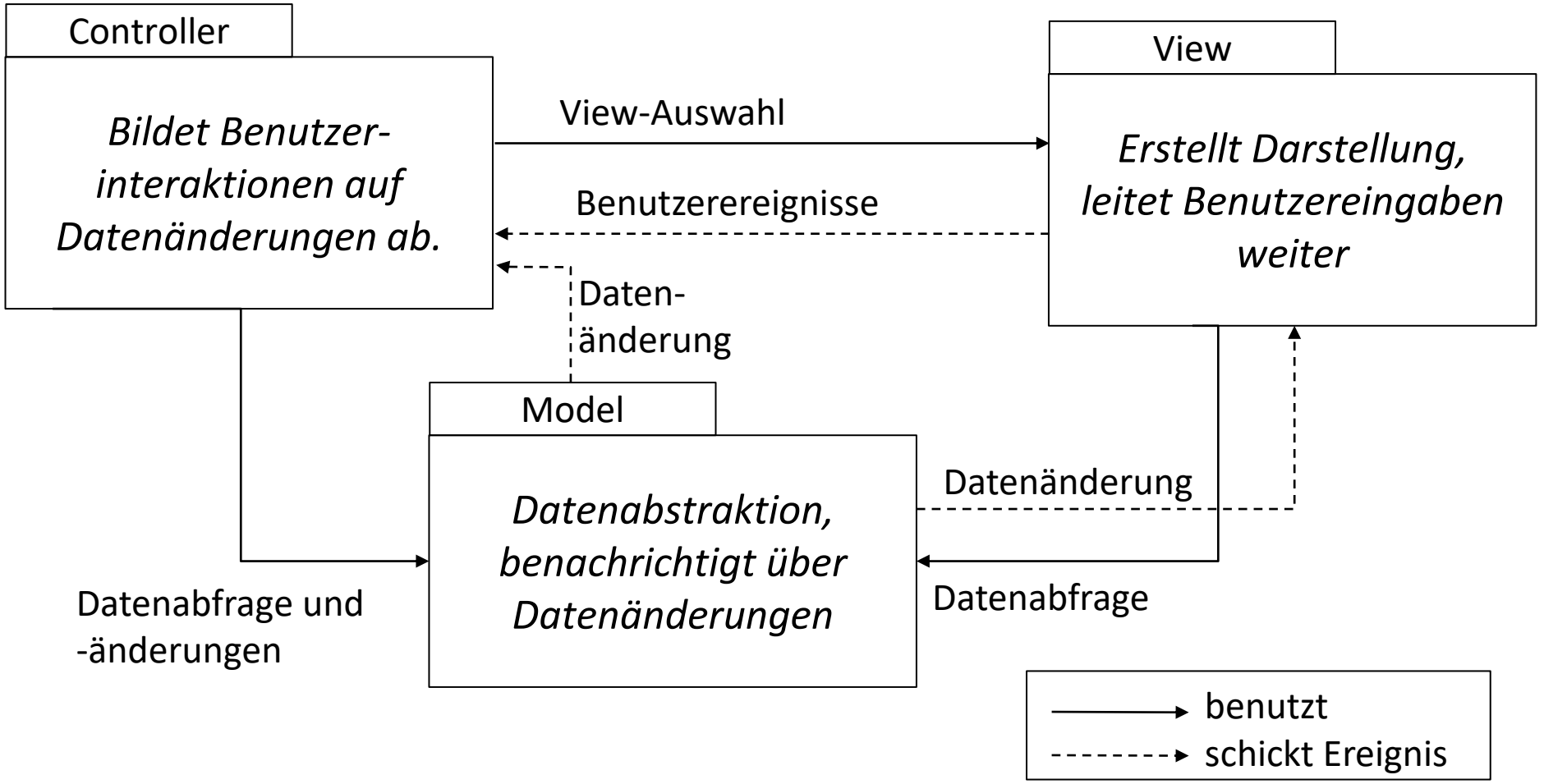
Wechsel des angezeigten Inhalts in einem Fenster: Der entsprechenden `Stage` eine andere `Scene` zuweisen oder der zugeordneten `Scene` einen anderen Wurzelknoten.

Model-View-Controller Architektur



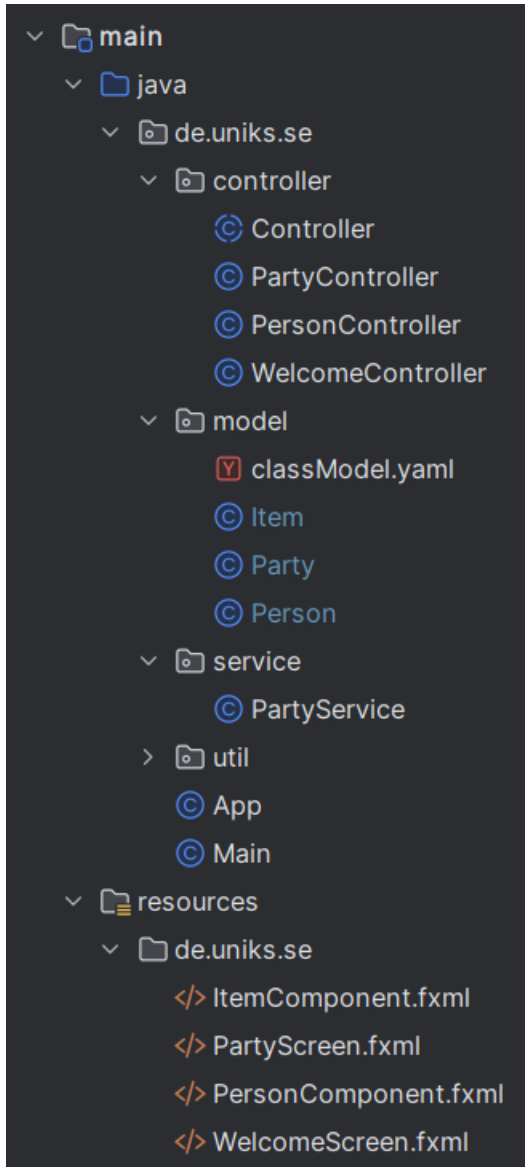
- Model-View-Controller (MVC): Entwurfsmuster, das sehr häufig bei der Entwicklung von (grafischen) Nutzerschnittstellen eingesetzt wird
  - Entwurfsmuster (Pattern) bieten bewährte Schablonen für wiederkehrende Probleme
- Ziel von MVC: Separieren von (grafischer) Darstellung, Interaktion und Daten
- Die Anwendung wird in drei Komponenten strukturiert:
  - Model: Verwaltet Daten und implementiert Operationen auf diesen Daten
  - View: Stellt Daten dar und nimmt Benutzereingaben entgegen
  - Controller: Verwaltet Benutzereingaben und reicht diese an Model (und View) weiter

Model-View-Controller Architektur



Eigenschaften MVC

- Unabhängige Implementierung verschiedener Darstellungen
  - Für dieselben Daten sind verschiedene Darstellungen und Interaktionsformen an der Oberfläche wählbar.
  - Darstellungen und Interaktionen sind erweiter- und austauschbar.
- Datenänderungen wirken sich auf alle Darstellungen aus.
- Liegt inzwischen in diversen Varianten vor



MVC Architektur in Beispielanwendung

- Model: Datenmodell mit den Klassen `Item`, `Party` und `Person` und den grundlegenden zugehörigen Methoden (z.B. Getter und Setter)
- View: Vier fxml-Dateien, die das Aussehen der Anwendung bestimmen
- Controller: Drei konkrete Controller, die die Anzeige steuern

- Service: Klasse `PartyService` enthält Programmlogik (z.B. Anlegen von Items, Parties oder Personen), auf die die Controller zugreifen
- util: Klassen zum Einstieg in die Anwendung und steuern des grundsätzlichen Ablaufs

Steuerung durch Controller


```
public class App extends Application {
   public void start… {
      …
      stage.setScene(scene);
      showWelcomeView();
      …
      stage.show();
   }

   public void showWelcomeView() {
      Controller welcomeController = …
      show(welcomeController);
   }

   public void show(Controller controller) {
      this.controller = controller;
      initAndRender(controller);
   }

   private void initAndRender(Controller controller) {
      stage.getScene().setRoot(controller.render());
   }
}
```

Ausschnitt aus Klasse App.java (vereinfacht)

- Im Prinzip könnten Aufbau und Änderungen des Scene Graph als Skript in der Klasse App (und weiteren Klassen) geschrieben werden.
- Unser Vorgehen:
  - In der `start`-Methode wird der stage eine scene zugewiesen und mit `show()` eingeblendet.
  - Die Klasse App stellt Methoden zum Einblenden von Controllern zur Verfügung; diese setzen die Wurzel der scene neu.
  - Die Controller steuern, was angezeigt wird, den Wechsel zwischen Controllern, …
  - Die Controller erhalten ihre Scene Graphs aus den entsprechenden Views (fxml-Dateien).

Kommunikation zwischen fxml und Controller

Im View (fxml-Datei):

- Jeder fxml-Datei kann ein Controller zugewiesen werden.
- Elementen in der fxml-Datei können IDs zugewiesen werden.
- Elemente in der fxml-Datei können mit Event-Handling Methoden aus dem Controller verknüpft werden.

Im Controller:

- können fxml-Dateien geladen werden;
- können Felder über die IDs mit den entsprechenden Elementen aus der fxml-Datei verknüpft werden;
- können Event-Handling Methoden definiert werden.

Setzen von Controller und Zuweisen von IDs

```
<?import …?>

<VBox … fx:controller="de.uniks.se.controller.PartyController">
   …
   <children>
      <Label fx:id="partyNameLabel" text="PARTYNAME_LABEL" />
      …
      <Label fx:id="partyLocationLabel" text="LOCATION_LABEL" />
   </children>
   …
```

Ausschnitt aus `PartyScreen.fxml`

Setzen des Controllers:

- Controller wird als Attribut `fx:controller` im Wurzel-Element der fxml-Datei gesetzt
- Argument ist der Pfad zur Controller-Klasse

Zuweisen von IDs:

- Über ein Attribut `fx:id` kann einem Element eine ID zugewiesen werden.
- Die IDs müssen eindeutig sein! (Konvention: Kleinschreibung und CamelCase)
- Das Setzen von Controllern und Zuweisen von IDs kann auch im Scene Builder geschehen.

Laden von fxml-Dateien durch Controller

```
public class PartyController extends Controller {
    private Party party;
    private Parent parent;

    @FXML public VBox itemVBox;
    @FXML private VBox personVBox;
    @FXML Label partyNameLabel;
    @FXML Label partyDateLabel;
    …
    public void init() {
        final FXMLLoader loader = new
                FXMLLoader(Main.class.getResource("PartyScreen.fxml"));
        loader.setControllerFactory(c -> this);
        try {
            this.parent = loader.load();
        } catch (IOException e) {
             throw new RuntimeException(e);
          }
    }
}
```

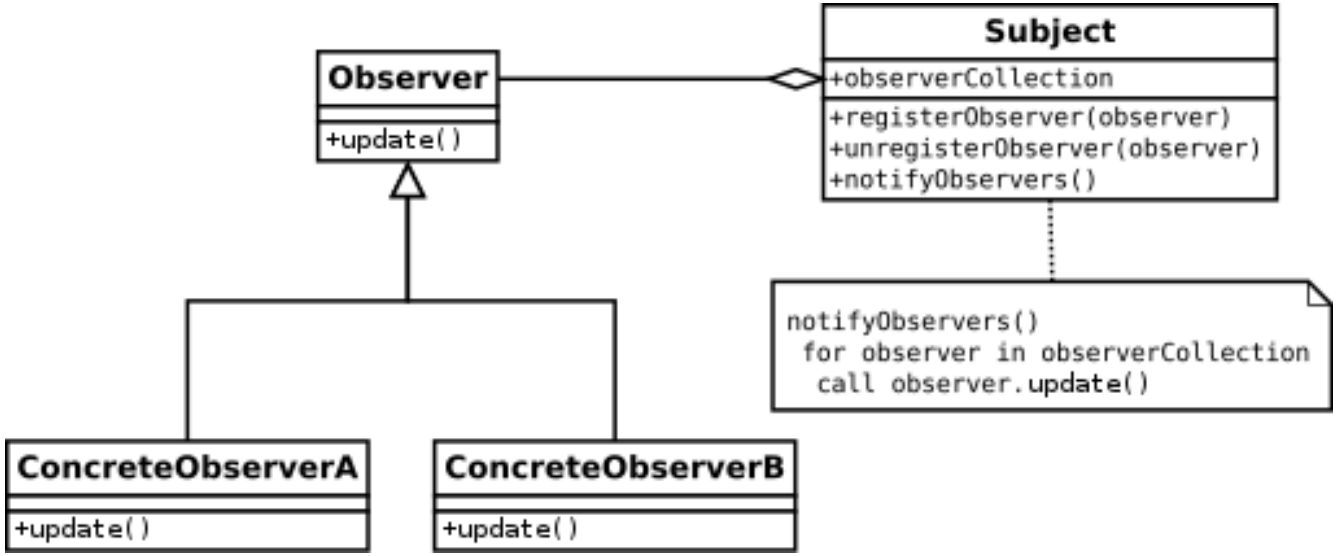
Ausschnitt aus `PartyController.java`

- Die Klasse `FXMLLoader` dient zum Laden von FXML-Objekthierarchien – hierfür stellt sie die `load`-Methode bereit.
- Bei Aufruf der `load`-Methode werden Felder der Controller-Instanz gesetzt: Felder des Controllers, deren Name mit der `fx:id` eines Elements aus der fxml-Datei übereinstimmt und die entweder `public` oder mit `@FXML` annotiert sind, werden mit den Werten aus der fxml-Datei gefüllt.

Informationsaustausch

- Controller müssen erfahren, wenn sich Werte im Datenmodell ändern, um die View aktualisieren zu können.
- Controller müssen erfahren, wenn in der laufenden Anwendung Textfelder ausgefüllt, Buttons geklickt, … werden, und die passenden Methoden bereitstellen bzw. aufrufen, um darauf zu reagieren.
- Etabliertes Pattern zur Implementierung der Reaktion auf solche Events: Observer Pattern
- Java und JavaFX stellen Funktionalität bereit, damit man das Pattern nicht jedes Mal vollständig ausimplementieren muss.

Observer Pattern (Grundidee)



[Quelle: Gregorybleiker, CC BY-SA 4.0, via Wikimedia Commons]

PropertyChangeSupport in Java


```
public class MyBean {
    private final PropertyChangeSupport pcs = new PropertyChangeSupport(this);
    public void addPropertyChangeListener(PropertyChangeListener listener) {
        this.pcs.addPropertyChangeListener(listener);
    }

    public void removePropertyChangeListener(PropertyChangeListener listener) {
        this.pcs.removePropertyChangeListener(listener);
    }

    private String value;
    public String getValue() { return this.value; }
    public void setValue(String newValue) {
        String oldValue = this.value;
        this.value = newValue;
        this.pcs.firePropertyChange("value", oldValue, newValue);
    }
    [...]
}
```

Quelle:
https://docs.oracle.com/en/java/javase/21/docs/api/java.desktop/java/beans/PropertyChangeSupport.html

- Einer Klasse kann ein Objekt vom Typ `PropertyChangeSupport` zugewiesen werden.
- Dem Objekt vom Typ `PropertyChangeSupport` können `PropertyChangeListener` hinzugefügt (und auch wieder entfernt) werden:
  - `addPropertyChangeListener(PropertyChangeListener listener)` für einen allgemeinen Listener;
  - `addPropertyChangeListener(String propertyName, PropertyChangeListener listener)` für einen Listener für spezielle Eigenschaft.
- Wenn Daten sich geändert haben, kann ein `PropertyChangeEvent` gefeuert werden.
  - Parameter der `firePropertyChange`-Methode kann ein `PropertyChangeEvent` sein oder der Name der Eigenschaft (als String), die sich geändert hat, und der alte und neue Wert.
  - Alle registrierten `PropertyChangeListener` (für die gefeuerte Eigenschaft) werden informiert.

PropertyChangeListener

- `PropertyChangeListener` ist ein funktionales Interface: es stellt die abstrakte Methode `propertyChange(PropertyChangeEvent evt)` zur Verfügung. Statt eines `PropertyChangeListeners` kann also ein Lambda-Ausdruck übergeben werden (seit Java 8).
- Wird ein `PropertyChangeEvent evt` gefeuert, wird für alle registrierten `Listener` die Methode `propertyChange` mit `evt` als Eingabe ausgeführt.
- Wichtige Methoden von `PropertyChangeEvent`:
  - `getNewValue()`
  - `getOldValue()`
  - `getSource()`

Listener in JavaFX

```
public class PartyController extends Controller {
    …
    public Parent render() {
        …
        guestNameTextField.textProperty().addListener(
            this::checkItemCreateStatus);
        …
    }
    private void checkItemCreateStatus… {
        …
    }
}
```

Auszug aus `PartyController.java` (vereinfacht)

- JavaFX bietet auch eigene Listener an (`ValueChangeListener`, `ListChangeListener`, …)
- Allgemeinste Variante:
  - JavaFX stellt Interface `ObservableValue` zur Verfügung; dies bietet die Methode `addListener(ChangeListener<? super T> listener)` an.
  - `ChangeListener<T>` ist ein funktionales Interface.
  - JavaFX Control-Elemente haben häufig Properties, die das Interface `ObservableValue` implementieren.

Action Handler I

```
<VBox fx:controller="com.foo.MyController"
   xmlns:fx="http://javafx.com/fxml">
   <children>
      <Button text="Click Me!"
         onAction="#handleButtonAction"/>
   </children>
</VBox>
```

```
package com.foo;

public class MyController {
   public void handleButtonAction(ActionEvent event) {
      System.out.println("You clicked me!");
   }
}
```

Quelle: https://openjfx.io/javadoc/21/javafx.fxml/javafx/fxml/doc-files/introduction_to_fxml.html#controllers

- Controls (z.B. `Button`, `MenuItem`, `TextField`) können in der fxml-Datei mit einem `onAction`-Attribut versehen werden.
- Syntax: `onAction="#<name>"`; Konvention: der Name startet mit “handle” und beschreibt dann die Aktion, die zu handeln ist.
- Der Controller der fxml-Datei implementiert dann eine Methode mit dem entsprechenden Namen und `ActionEvent` als Parameter (entweder public oder annotiert mit @FXML).
- Semantik: Wenn die zur Control passende Aktion (Mausklick, Drücken der Enter-Taste) durch einen Nutzer ausgeführt wird, wird die entsprechende Methode aufgerufen.

Action Handler II

```
public class PartyController extends Controller {
   @FXML Button addItemButton;

   …

   public Parent render() {
      …
      addItemButton.setOnAction(this::handleAddItem);
      …
   }

   private void handleAddItem(ActionEvent evt) {
      …
   }
}
```

Auszug aus PartyController.java (vereinfacht)

- Control-Elemente wie `Button`, `TextField`, `MenuItem` in JavaFX verfügen auch über eine `setOnAction`-Methode; damit können Action Handler im Controller definiert werden, ohne in der fxml-Datei das onAction-Attribut setzen zu müssen.
- Parameter von `setOnAction` ist von Typ `EventHandler<ActionEvent>`.
  - `EventHandler` ist wieder ein funktionales Interface mit abstrakter Methode `handle(T event)`.
  - Es kann also ein Lambda-Ausdruck als Parameter verwendet werden.