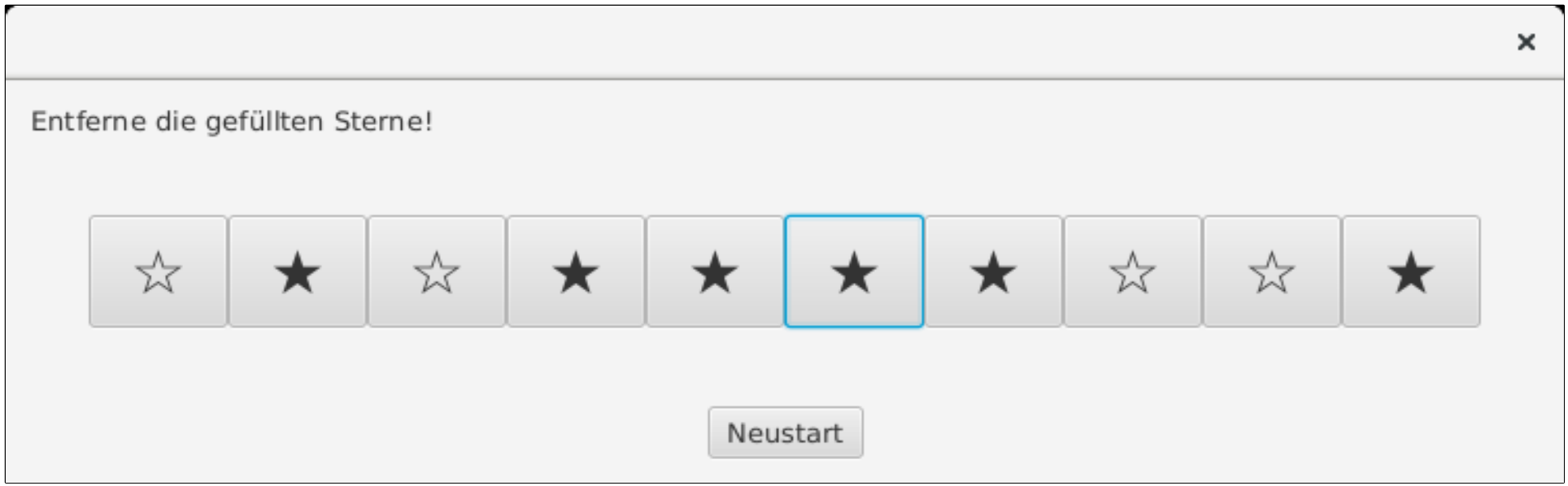
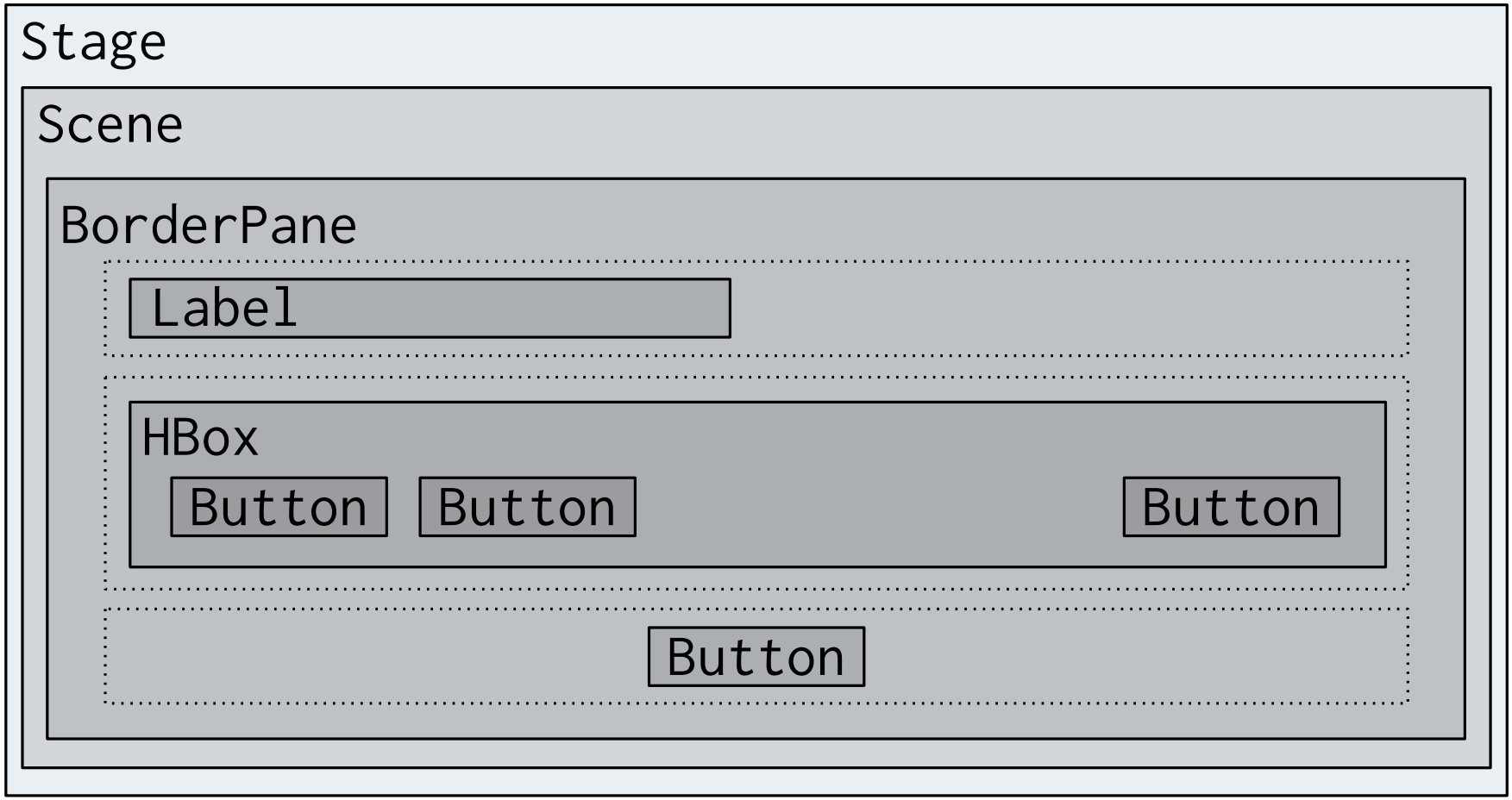
JavaFX
Beispiel „Lights Out“ (Teil 1: Ansicht)

Beispiel „Lights Out“

Als Beispiel eines vollständigen Programms entwickeln wir eine einfache lineare Variante von Lights Out.
Siehe: `https://en.wikipedia.org/wiki/Lights_Out_(game)`



Struktur der Anzeige



- `BorderPane`, `Label`, `Button`, `HBox` sind Unterklassen von `Node`.
- Ein `BorderPane`-Objekt kann fünf `Node`-Objekte enthalten (center, top, bottom, left, right). Benutze `HBox`, um mehrere `Node`-Objekte zu einem zusammenzufassen.

Implementierung der Anzeige

```
public class View0 extends Application {

  public void start(Stage stage) {

      Button restartButton = new Button("Neustart");
      Button messageLabel = new Label("Entferne [...]");

      HBox hbox = new HBox();
      for (int i = 0; i < 10; i++) {
        Button button = new Button();
        hbox.getChildren().add(button);
      }

      BorderPane borderPane = new BorderPane();
      borderPane.setBottom(restartButton);
      borderPane.setCenter(hbox);
      borderPane.setTop(messageLabel);

      Scene scene = new Scene(borderPane, 750, 200);
      stage.setScene(scene);
      stage.show();
  }
}
```

Reagieren auf Ereignisse

Um die Programmlogik zu implementieren, müssen wir auf das Drücken der Buttons reagieren.

Programmcode, der bei Eintreten des Knopfdruck-Ereignisses ausgeführt wird, kann so registriert werden:

```
public class View0 extends Application {

  public void start(Stage stage) {

      Button restartButton = new Button("Neustart");

      restartButton.setOnAction(event -> {
         // println hier nur zum Test,
         // dass der Code auch ausgefuehrt wird
         System.out.println("Button Neustart gedrueckt.");
      });
      ...
```

Implementierung der Programmlogik

Es bleibt noch zu implementieren:

- Datenmodell und die zugehörige Logik: welche Lichter sind an, welche sind aus und was passiert, wenn man ein Licht an- oder ausschaltet.
- Ablauf der Benutzerinteraktion und Verbindung der JavaFX-Anzeige mit den Daten im Modell.

Es gibt viele Möglichkeiten, das zu implementieren. Unstrukturieres Vorgehen kann sehr leicht zu kompliziertem Spaghetticode führen.

Praktisch alle Programme mit grafischer Benutzerschnittstelle sind nach dem *Model-View-Controller*-Pattern strukturiert.

Design Patterns:
Observer, Model-View-Controller

Was sind Design Patterns?

Design Patterns (Entwurfsmuster) sind Erfahrungswerte des Entwurfs objektorientierter Programme.

Ziele des objektorientierten Entwurfs

- **Korrektheit**: Der Entwurf sollte dabei helfen, Programmierfehler zu vermeiden.
- **Wiederverwendbarkeit**: Man sollte ein Problem nur einmal lösen müssen.
- **Verständlichkeit**: Programme sollten lesbar und verständlich sein, z.B. um Teamarbeit zu erleichtern und Fehler zu vermeiden.
- **Effizienz**: Der Entwurf sollte eine effiziente Entwicklung erlauben und zu effizienten Programmen führen.

Grundprinzipien des Entwurfs

Es gibt eine Reihe von Grundprinzipien des Entwurfs.

Beispiel:

Single Responsibilty Prinziple

- Jede Klasse/Methode/Variable hat eine einzige Aufgabe.
- Es sollte nie mehr als einen Grund geben, eine Klasse/Methode/Variable zu ändern.

  Beispiel: Programm zur Ausgabe von Buchhaltungsdaten
  - mögliche Änderungsgründe:
    - Formel zur Berechnung der Steuer hat sich geändert
    - Schriftart der Ausgabe soll größer sein
  - Änderungen wegen dieser Gründe sollten verschiedene Klassen betreffen.

Erfahrungswerte

Erfahrungswerte spielen im objektorientierten Entwurf eine große Rolle.

Aufzeichung von Erfahrungswerten:

- **Design Patterns (Entwurfsmuster)**: Beschreibung bewährter Ansätze zur Lösung verschiedener Probleme
- **Anti-Patterns**: problematische Entwicklungsmuster, die man vermeiden sollte
- **Design Smells**: Kriterien, die auf Probleme am Entwurf hindeuten
- **Code Smells**: Kriterien, die auf Probleme am Programmcode hindeuten
- ...

Design Patterns

Ein **Design Pattern** dokumentiert einen Ansatz zur Lösung eines Problems, das häufig in verschiedenen Kontexten auftritt.

Design Patterns geben häufig benutzten Lösungsansätzen einen Namen und erleichern so die Kommunikation.

Design Patterns zeichnen die Erfahrungen zu Vor- und Nachteilen eines Lösungsansatzes auf.

Entwicklung Graphischer Benutzeroberflächen

Grundprinzip: Trenne Programm in Datenmodell und Ansicht

Zuständigkeiten (vgl. Single-Responsibility-Principle):

- **Modell**: Datenhaltung und -verarbeitung
- **Ansicht**: Anzeige der Daten

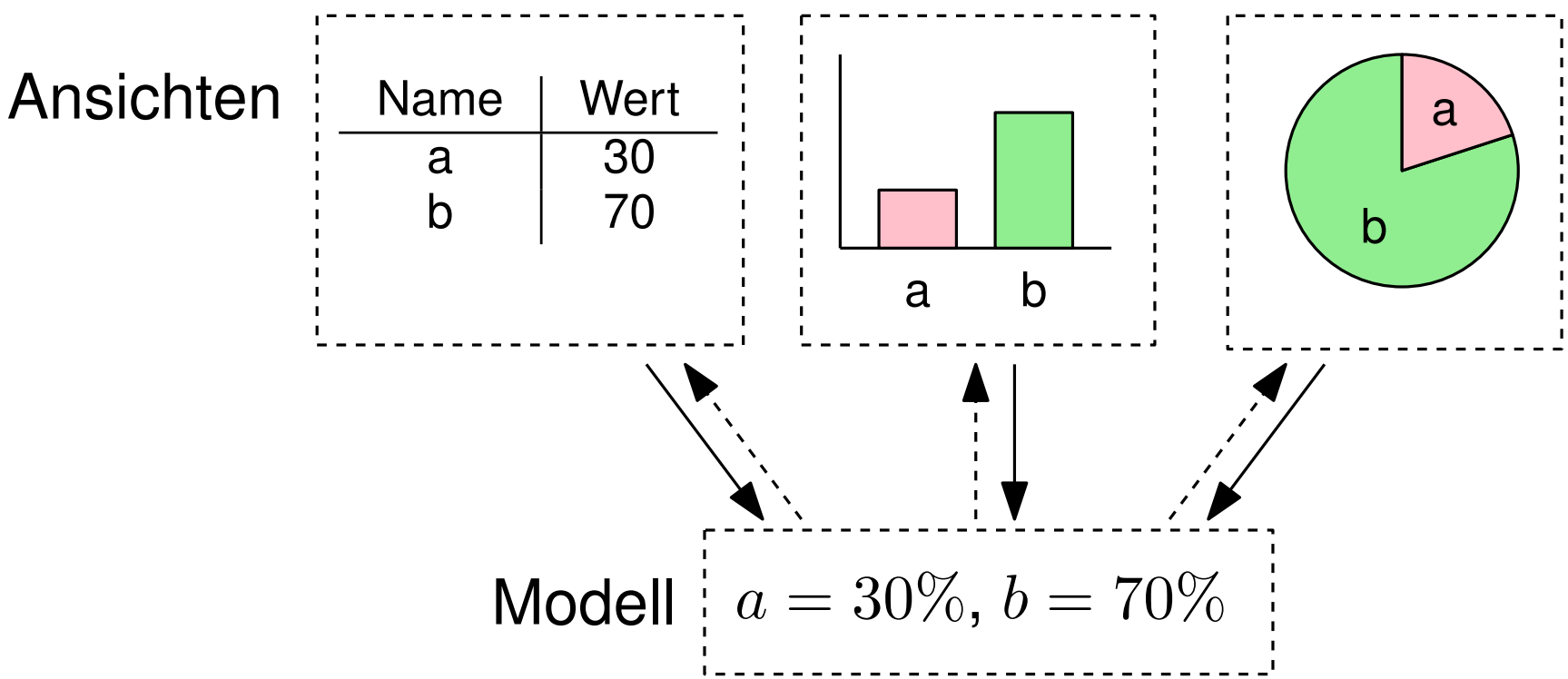
Relationen:

- Die Ansicht hängt vom Modell ab und muss darauf zugreifen.
- Das Datenmodell wird unabhängig von der Anzeige entwickelt.
- Verschiedene GUIs können dasselbe Modell benutzen.

Observer-Pattern

Problem:

- Jede Ansicht muss aktualisiert werden, wenn sich Daten im Modell ändern.
- Wie kann das Modell die Ansichten von Änderungen an Werten informieren, wenn es unabhängig entwickelt wird?



Observer-Pattern

Lösung:

Das Modell verschickt Änderungsbenachrichtigungen mit einem Abonnentensystem.

- Beliebige Interessenten (z.B. Anzeigeklassen) können sich beim Modell als Zuhörer (Observer) registrieren.
- Bei jeder Änderung der Daten benachrichtigt das Modell alle registrierten Zuhörer.

Observer-Pattern in Java

Verschiedene Implementierungen in Java

- `java.util.Observer`: Veraltet (wegen Probleme mit Multithreading und Reihenfolge), für das Praktikum immernoch verwendbar
- `javafx.beans.Observable` und `javafx.beans.InvalidationListener`: Starke JavaFX-Abhängigkeit, darum schlechter für JavaFX-unabhängige Projekte wiederverwertbar
- `java.util.concurrent.Flow`: Allgemeinere, dafür etwa komplexere Lösung

Observer-Alternativen

Alternativen zu Observer sind

- Periodische Aktualisierungen (bei einigen Projekten ohnehin notwendig)
- Explizite Aktualisierungen bei Knopfdrücken (gerade für kleine und mittelgroße Pojekte ok, bei großen Projekten problematisch)

Model-View-Controller

Model-View-Controller ist ein Pattern zur Implementierung graphischer Benutzeroberflächen.

- bereits in den 70er Jahren als eines der ersten Entwurfsmuster formuliert
- seitdem in verschiedenen Varianten den aktuellen Softwaressystemen angepasst
- fester Architekturbestandteil GUI-basierter Betriebssysteme und GUI-Frameworks

Trennung in drei Zuständigkeitsbereiche:

1. Datenrepräsentation (Model)
2. Anzeige von Daten (View)
3. Kontrolle der Benutzerinteraktion (Controller)

Model-View-Controller: Rollen

Modell

- repräsentiert Daten
- implementiert Algorithmen
- vollkommen unabhängig von der Benutzerschnittstelle

View

- zeigt Benutzer bestimmte Daten und Steuerelemente an

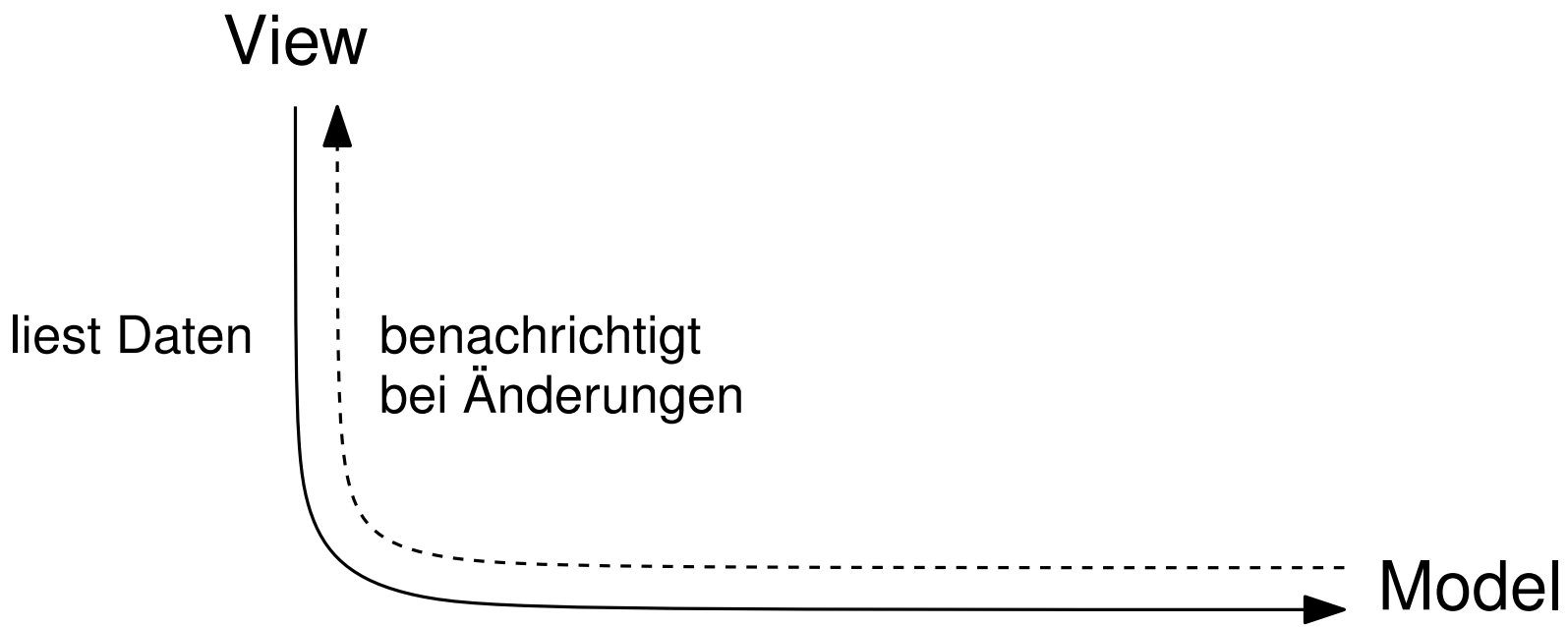
Controller

- implementiert Ablauflogik der Benutzterschnittstelle

Modell ist vom Rest strikt getrennt.
Controller und View sind eng gekoppelt.

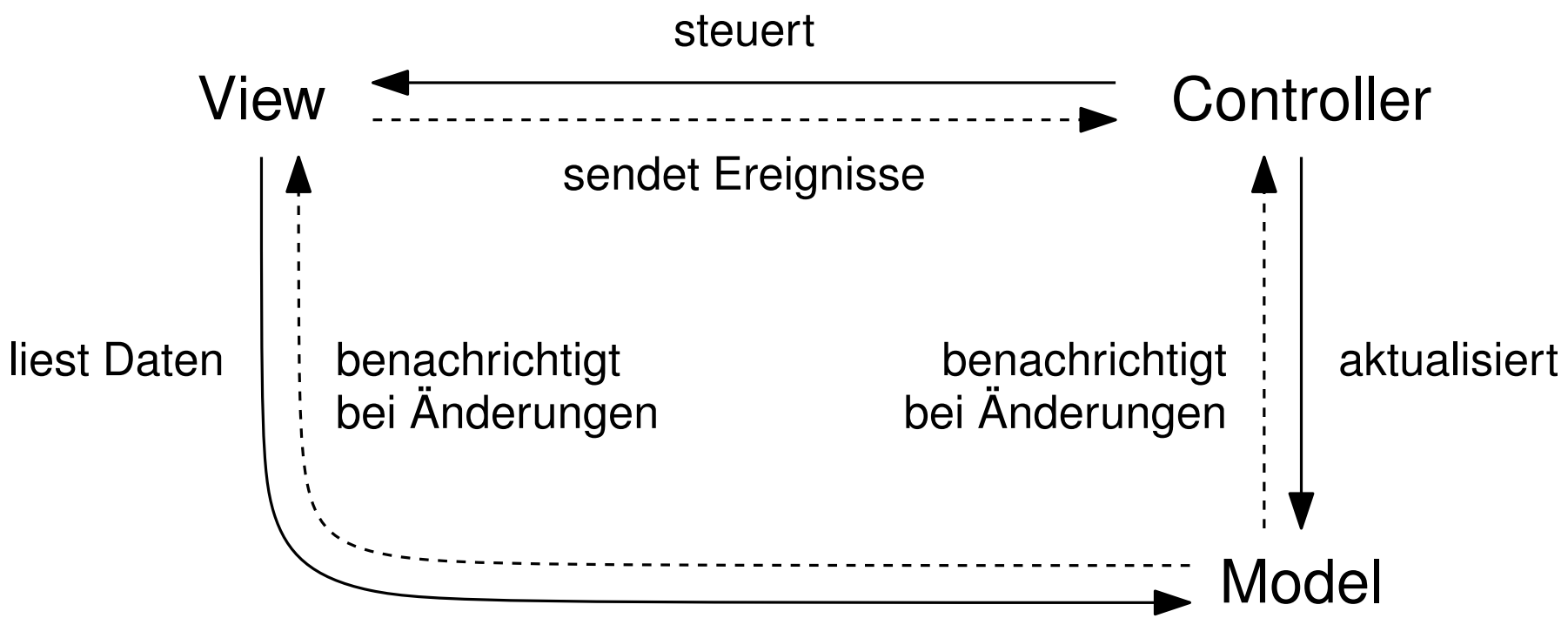
Model-View-Controller: Relationen

- View zeigt eine Auswahl der Daten des Modells an.
- View ist Observer des Modells, damit die Daten stets aktuell sind.



Model-View-Controller: Relationen

- Benutzereigaben im View führen zu Ereignissen, die der Controller behandelt.
- Controller plant Änderungen am Modell und führt sie aus.
- Controller implementiert Logik der Benutzerschnittstelle.
- Controller erfährt als Observer von Änderungen am Modell



Model-View-Controller

Minimalbeispiel

Ein Countdown wird per Knopfdruck von 10 auf 0 heruntergezählt.
Kurz vor Erreichen der 0 soll eine Warnung angezeigt werden.

- Model: speichert Zähler
- View: zeigt Zähler sowie Button zum Dekrementieren an
- Controller: behandelt Benutzereingabe und öffnet Benachrichtigungsfenster

Hauptprogramm

```
public class Main extends Application {

  @Override
  public void start(Stage stage) throws Exception {

    Model model = new Model();
    View view = new View(model, stage);
    Controller controller = new Controller(model, view);
  }

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

}
```

Modell

```
public class Model {

  private int countDown;

  public Model() {
    countDown = 10;
  }

  public int getCountDown() {
    return countDown;
  }

  public void tick() {
    if (countDown > 0) {
      countDown--;
    }
  }
}
```

View

```
public class View {

  private Model model;
  private Stage stage;
  private Label label;
  private Button countButton;

  public View(Model model, Stage stage) {
    this.model = model;
    this.stage = stage;
    label = new Label();
    countButton = new Button("Count");
    stage.setScene(new Scene(new VBox(label, countButton)));

    updateLabel();
  }

  private void updateLabel() {
    label.setText("Countdown: " + model.getCountDown());
  }

... getter ...
}
```

View

- hat Kenntnis vom Modell
- implementiert das Interface `Observer`
- registriert sich beim Modell als Observer, um bei Änderungen benachrichtigt zu werden
- aktualisiert die Anzeige, wenn das Modell Änderungen meldet

Controller

```
public class Controller {
  private View view;
  private Model model;

  public Controller(Model model, View view) {
    this.model = model;
    this.view = view;

    // Event-Handler registrieren
    this.view.getCountButton().setOnAction(event -> countAction(event));
    this.view.getStage().show();
  }

  public void countAction(ActionEvent event) {
    model.tick();
    update();
    view.update();
  }
  public void update() {
    if (model.getCountDown() == 1) {
      Alert alert = new Alert(Alert.AlertType.WARNING, "Achtung, gleich!", E
      alert.show();
    }
  }
}
```

Controller

- kontrolliert View bzgl. dessen was über Datenaktualisierung hinaus geht, z.B. Öffnen neuer Fenster
- behandelt Ereignisse, die in der Anzeige entsehen
- plant Änderungen am Modell und führt sie aus

Faustregel: Versuche den Controller so zu schreiben, dass man das Verhalten der Benutzerschnittstelle durch Lesen des Controllers verstehen kann.

Model-View-Control

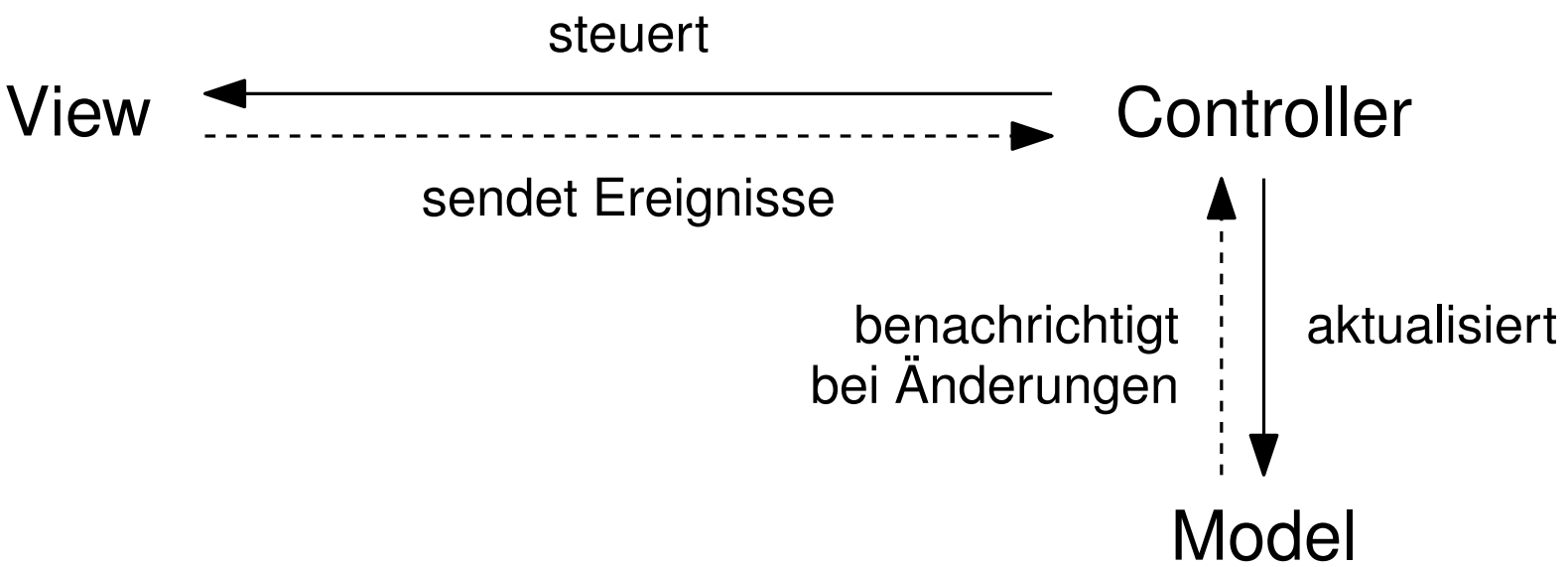
- Grundprinzip: Trennung des Datenmodells von der Anzeige
- Faustregel für das Praktikum:
  Keine `javafx`-Imports im Modell.
- Das MVC-Muster kommt im GUI-Programmen oft mehrfach vor.

  Die Steuerelemente `Label`, `Slider`, usw. entsprechen zum Beispiel selbst dem Muster. Zum Beispiel hat jedes `Label`-Objekt ein Modell, das einen String speichert.

Model-View-Controller: Varianten

Es gibt eine Reihe von Varianten des MVC-Musters.

- Die hier vorgestellte Variante findet man auch unter den Namen “Supervising Controller” und “Supervising Presenter”.
- Andere Varianten verlangen eine stärkere Trennung der Komponenten, z.B. **Model-View-Presenter**.



JavaFX
Beispiel „Lights Out“ (Teil 2: MVC)

Hauptprogramm

```
public class Main extends Application {

  @Override
  public void start(Stage primaryStage) throws Exception {
    Model model = new Model();
    View view = new View(model, primaryStage);
    new Controller(model, view);
  }

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

Den vollständigen Programmcode finden Sie auf der Praktikumshomepage.

Modell

Das Modell speichert den Zustand der Lichter und stellt Methoden zur Veränderung des Zustands bereit.

```
public class Model extends Observable {

  public static int NUMBER_OF_LIGHTS = 10;
  private final boolean[] lights;

  public Model() {
    lights = new boolean[NUMBER_OF_LIGHTS];
    randomiseLights();
  }

  /** Setzt die Lichter auf zufaellige Werte */
  public void randomiseLights() { ... }

  /** Gibt den Wert des i-ten Lichts zurueck */
  public boolean getLight(int i) { ... }

  /** Aendert den Wert des i-ten Lichts und seiner beiden Nachbarn. */
  public void toggle(int i) { ... }

  /** Gibt true zurueck falls alle Lichter aus sind. */
  public boolean allLightsAreOff() { ... }
}
```

View

Die Klasse `View` baut die Ansicht auf.

```
public class View {

  private Stage stage;
  private Label messageLabel;
  private Button[] lightButtons;
  private Button restartButton;
  private Model model;

  public View(Model model, Stage stage) {
    this.model = model;
    this.stage = stage;

    restartButton = new Button("Neustart");
    [...] // Rest des Aufbaus der Anzeige wie vorher.
  }

  public void update(Observable o, Object arg) { ... }

  public Stage getStage() { return stage; }
  public Stage getMessageLabel() { return messageLabel; }
  public Button getLightButton(int i) { return lightButtons[i]; }
  public Button getRestartButton(int i) { return restartButton; }
}
```

Controller

Die Klasse `Controller` steuert die Benutzerinteraktion.

```
public class Controller {

  private Model model;
  private View view;

  public Controller(Model model, View view) {
    this.model = model;
    this.view = view;

    for (int i = 0; i < Model.NUMBER_OF_LIGHTS; i++) {
      final int j = i;
      view.getLightButton(i).setOnAction(event -> model.toggle(j));
    }
    view.getRestartButton().setOnAction(event -> model.randomiseLights());

    view.getMessageLabel().setText("Willkommen! Entferne ...");
    view.getStage().show();
  }

  public void update(Observable o, Object arg) {
    // aktualisiere messageLabel, siehe den vollstaendigen Code
  }
}
```