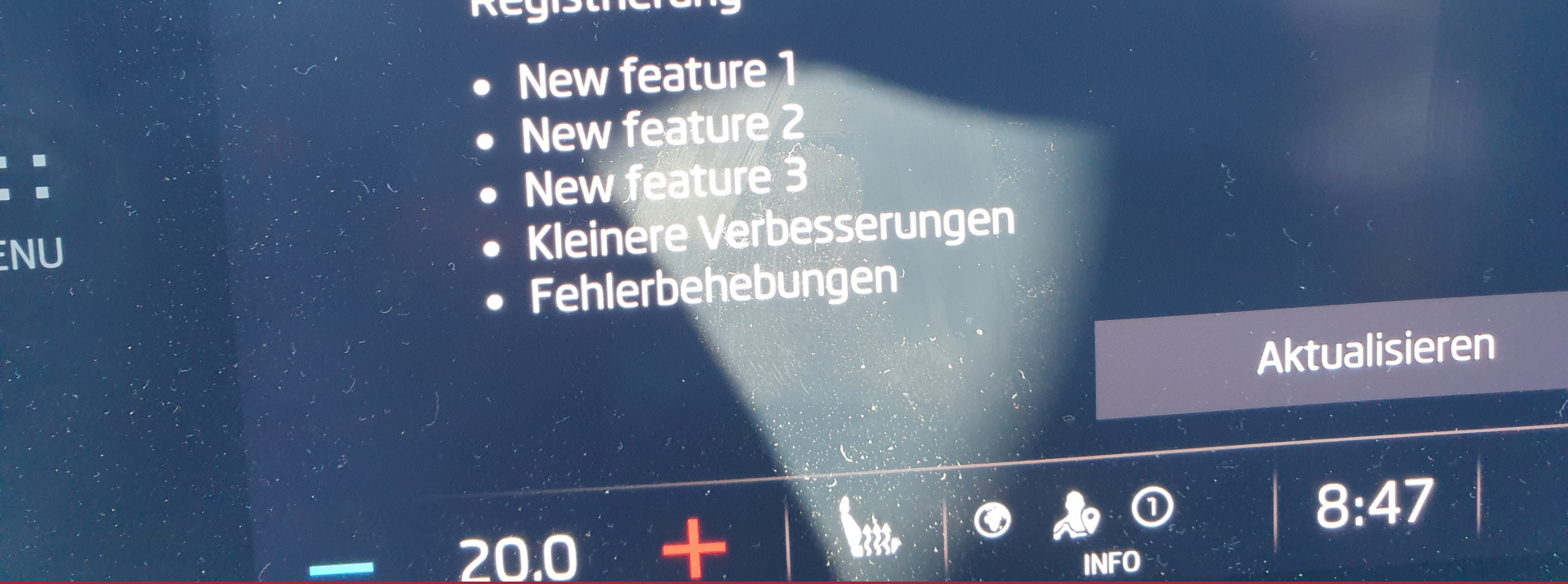


09-GUI-2-EventHandling

Objektorientierte Programmierung | Matthias Tichy

Software Engineering
Programming Languages

universität ulm

Lernziele

- Events
- Event Handling
- Zustands
- Zustandsautomaten

Let's play - Demo

Was ist neu?

- Icons
- RadioButtons
- Action!

- Image verarbeitet bmp, gif, jpg,png
- ImageView ist Node zur Bilddarstellung

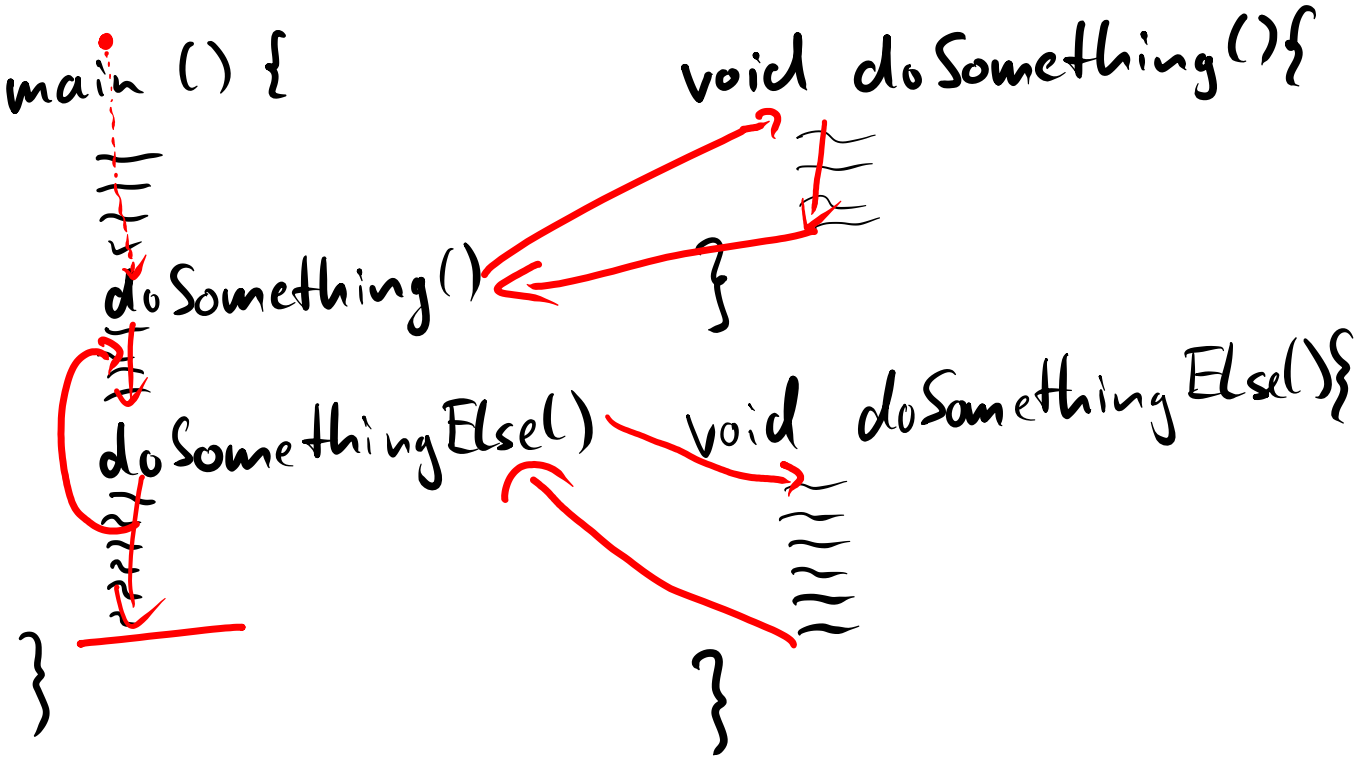
- ToggleGroup gruppiert z.B. Radiobuttons

Icons als Resources

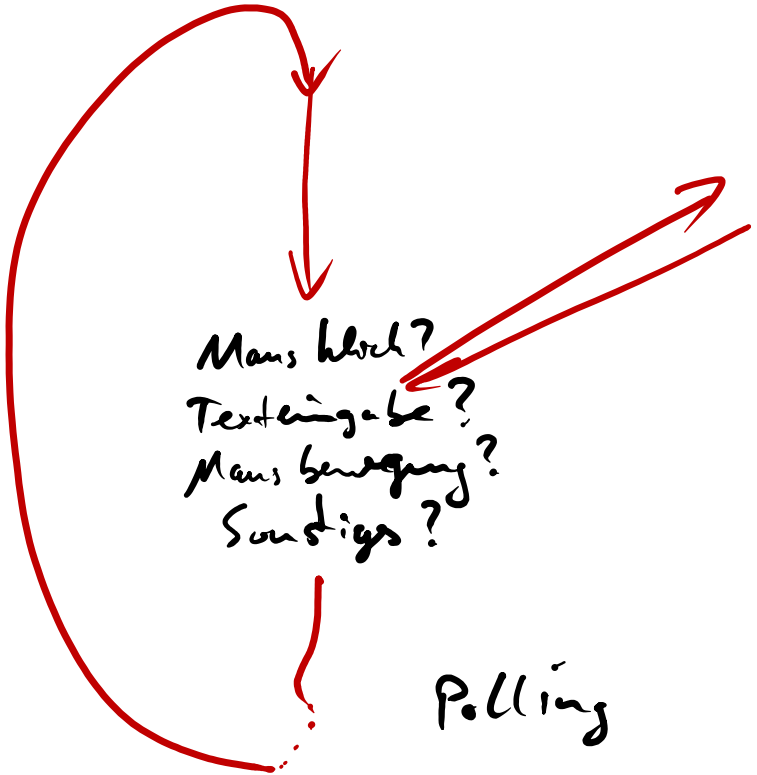
- absolute Pfade auf Datei problematisch
  `C:\Users\alex\workspace_pvs\FX-smiley-game\...`
- relative Pfade nur, wenn immer gleich
  `..\..\resources\open.png`
- Besser: als Resource speichern und mit Anwendung „einpacken“ lassen
- Zugriff über:
  `<mainClass>.getclass().getResource("<Pfad>")`

Programmablauf

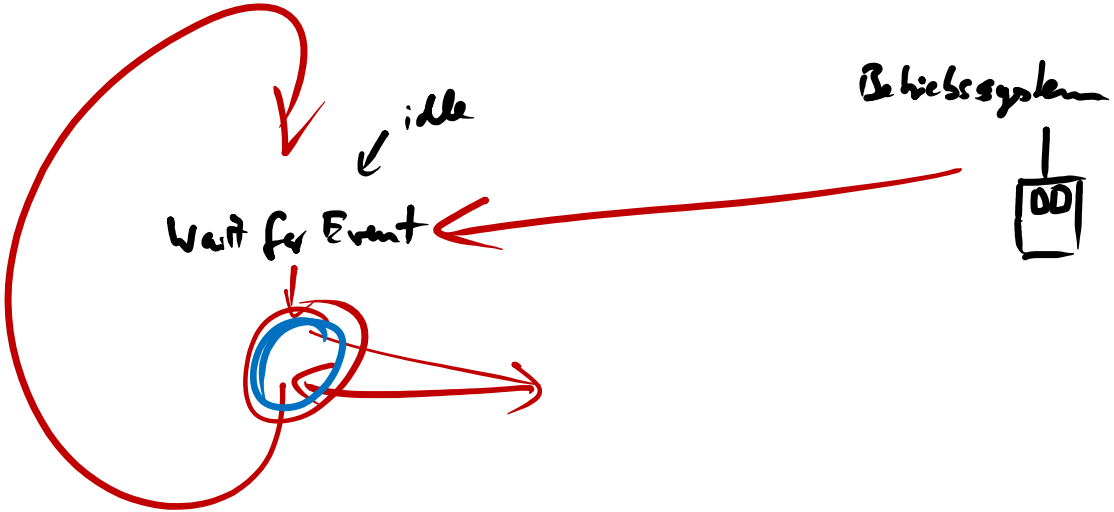
Programmablauf



Programmablauf



Ereignisbasierte Programmierung



An wen soll Ereignis geschickt werden?

Wichtige Aspekte

- Ereignis tritt unabhängig von Empfänger auf
- Empfänger interessiert evtl. gar nicht, woher das Ereignis kommt.
- Es kann mehrere Empfänger für ein Ereignis geben
- Ereignisnachrichten können auch getypt sein ➔ Empfänger kann sich nur für bestimmte Typen interessieren

Wer macht's?

1. jede Komponente erhält alle Ereignisse und filtert selbst

2. nur die Komponenten, die das Ereignis interessiert, werden aufgerufen

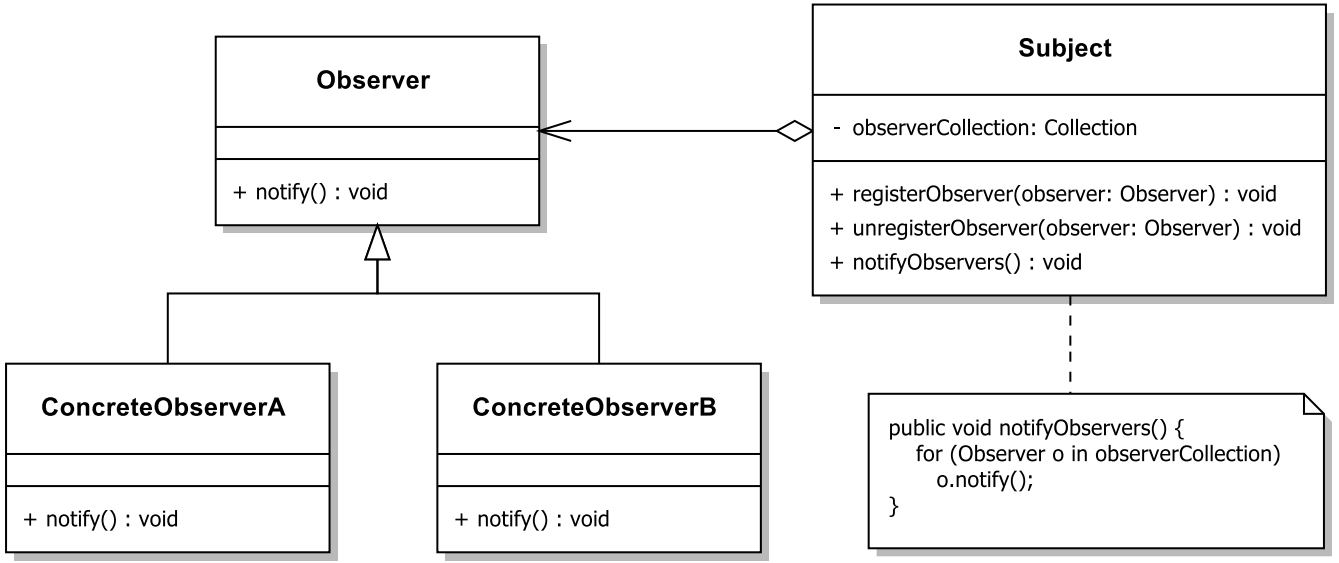
➔ woher weiß die Hauptschleife das?

Anmeldung für Informationen

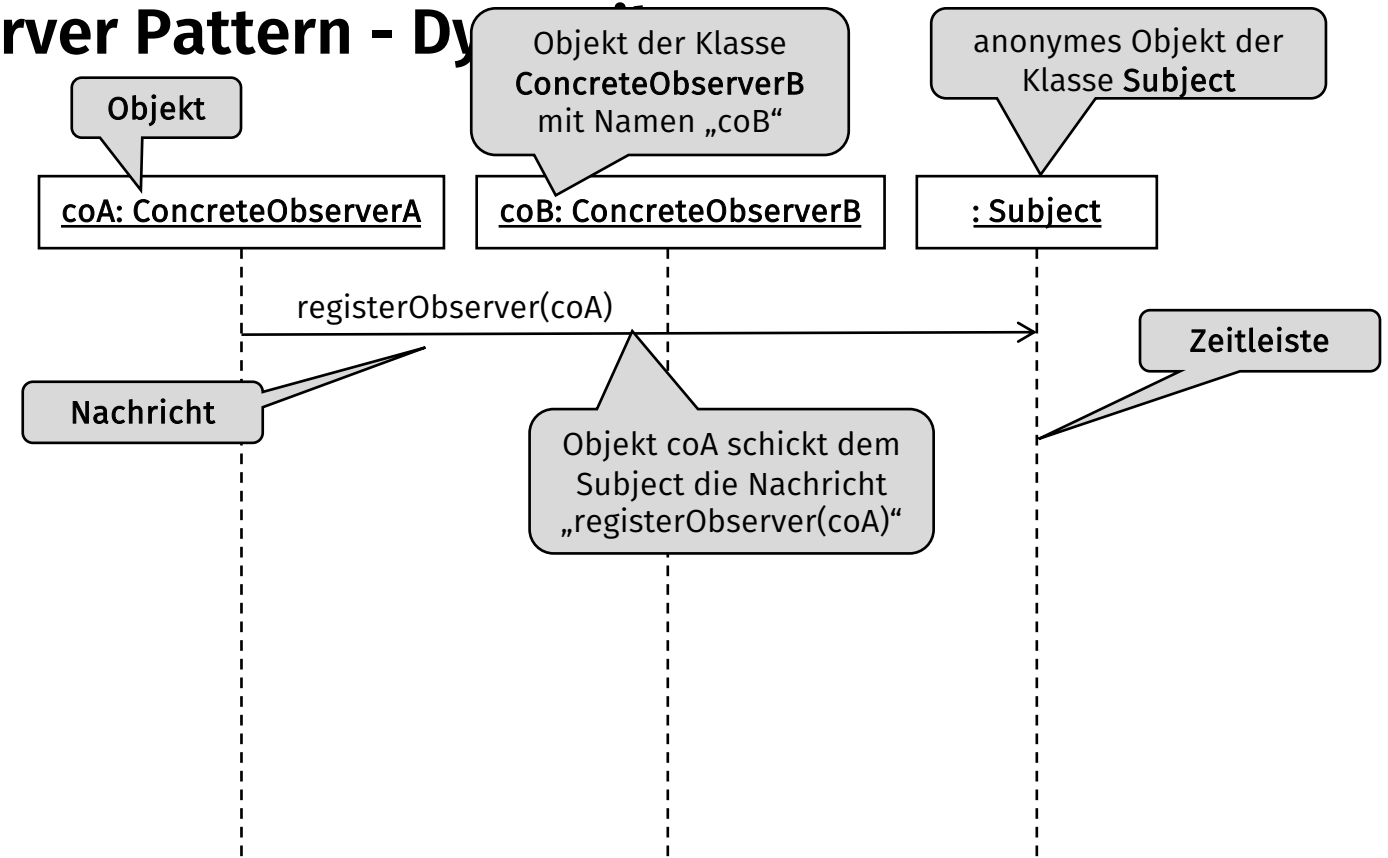
- Beispiel Newsletter
- Eintragen in eine Liste
- Bei jeder neuen Info bekommt man eine Nachricht
- Kann man lesen, muss man aber nicht

- Ebenso bei Mailinglisten, Twitter, etc.

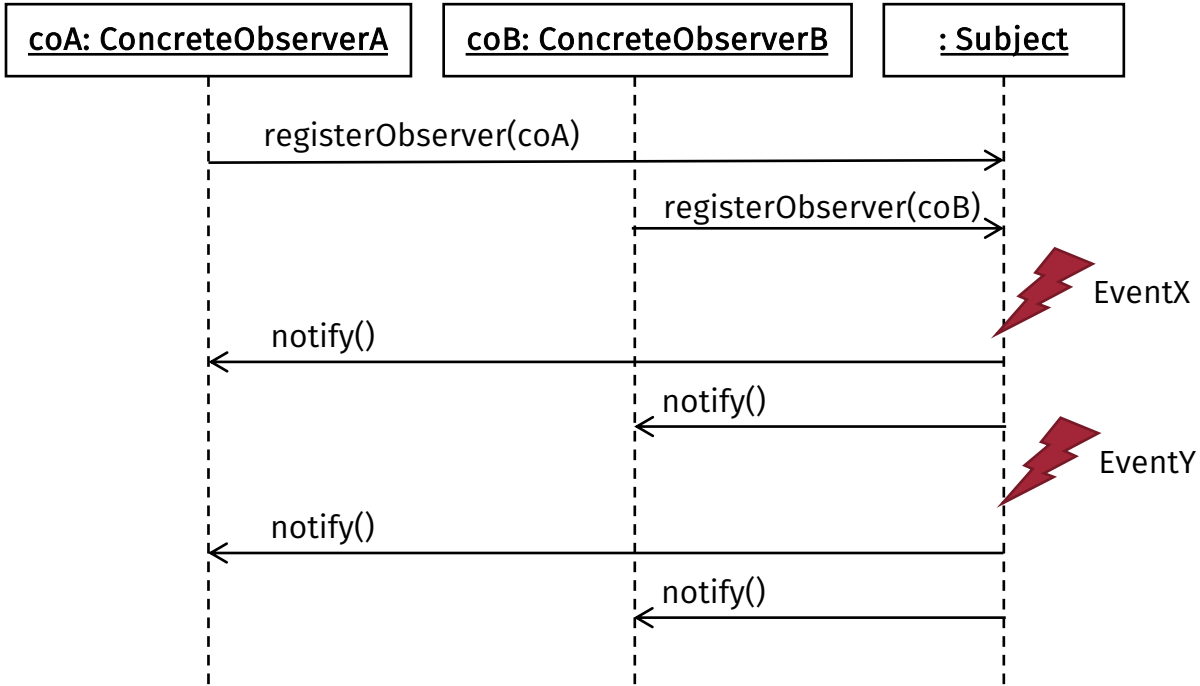
Observer Pattern



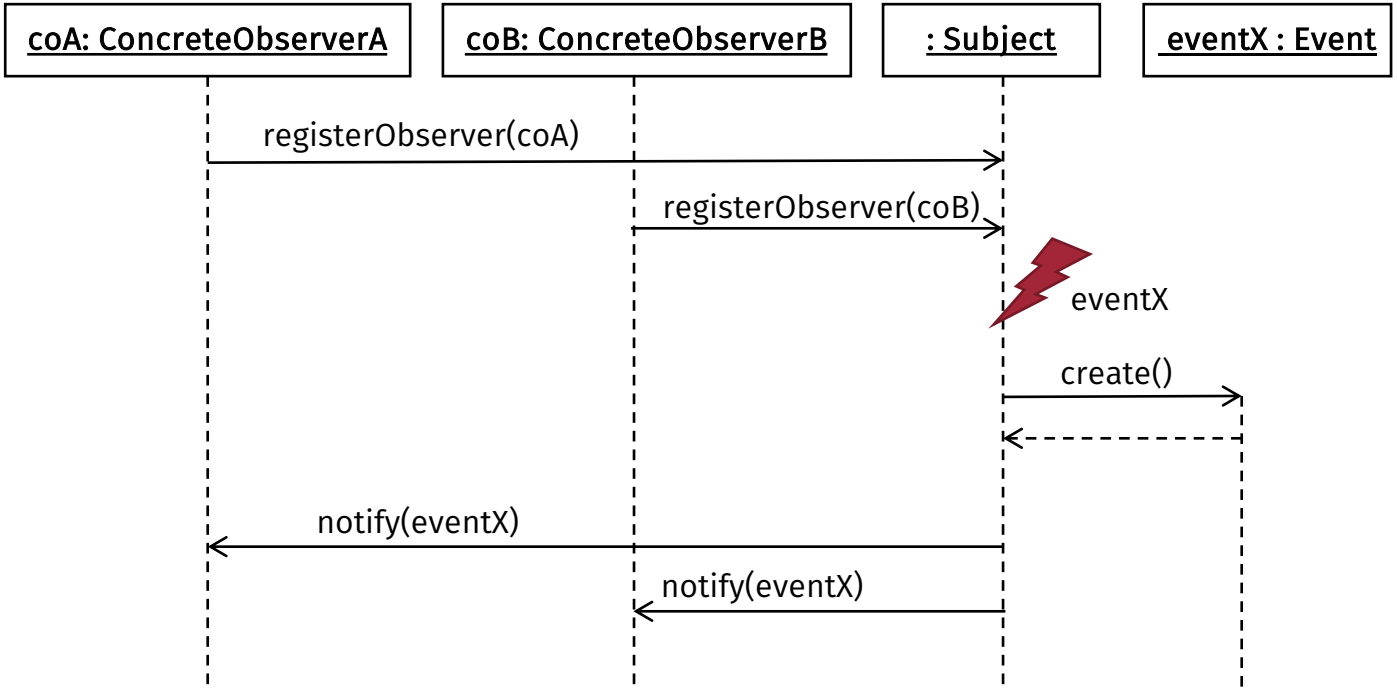
Observer Pattern - Dy



Observer Pattern - Dynamik



Observer Pattern – häufige Anpassung



Observer-Pattern in Java

- Interface `Observer`:

```
void update(Observable o, Object arg)
```

- Klasse `Observable`

```
public void addObserver(Observer o)
public void deleteObserver(Observer o)
public void notifyObservers()
public void notifyObservers(Object arg)
public void deleteObservers()
protected void setChanged()
protected void clearChanged()
public boolean hasChanged()
public int countObservers()
```

Collections?

- "normale" Collections nicht "Observable"
- FXCollections sind “Wrapper" der java.collections, allerdings observable (ObservableList, ObservableArray, ObservableMap, etc.)

- Package: `javafx.collections`
- `Class: FXCollections`

Probleme?

- Immer alles nur Object ➔ keine Generizität ➔ keine Typsicherheit, nur Typecasts

- nur genau eine update-Methode ➔ Unterscheidung der Ereignisse nur über Ereignisobjekt ➔ unschöne Fallunterscheidung in update-Methode

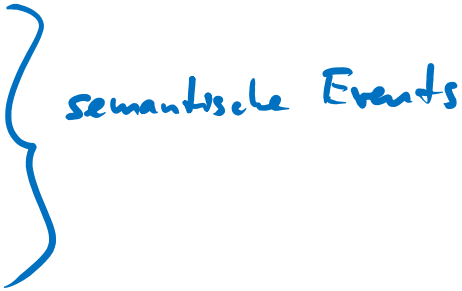
Schöner: EventHandler

- Ereignisobjekte heißen XXXEvent
- Java-Schnittstellen mit generischem Eventhandler (Funktionsreferenz)
- Registrierung eines Handlers für ein bestimmtes Event auf einem bestimmten Objekt

Typische Ereignisse

- Tastatureingabe
- Mausbewegung
- Mausklick

- Schließen eines Fensters
- Vergrößern/Verkleinern
- Buttonklick
- Änderung des (Text-)inhalts



Wie läuft das jetzt in JavaFX?

- Jedes Event hat folgende Inhalte:
  - EventType (siehe nächste Folie)
  - Source
  - Target (z.B. Scene oder Node)

- Jeder Handler implementiert das EventHandler-Interface:

```
public interface EventHandler<T extends Event>
                 extends EventListener {
  void handle(T event);
}
```

Typeinschränkung: Typparameter T muss von Event erben

Wie läuft das jetzt in JavaFX?

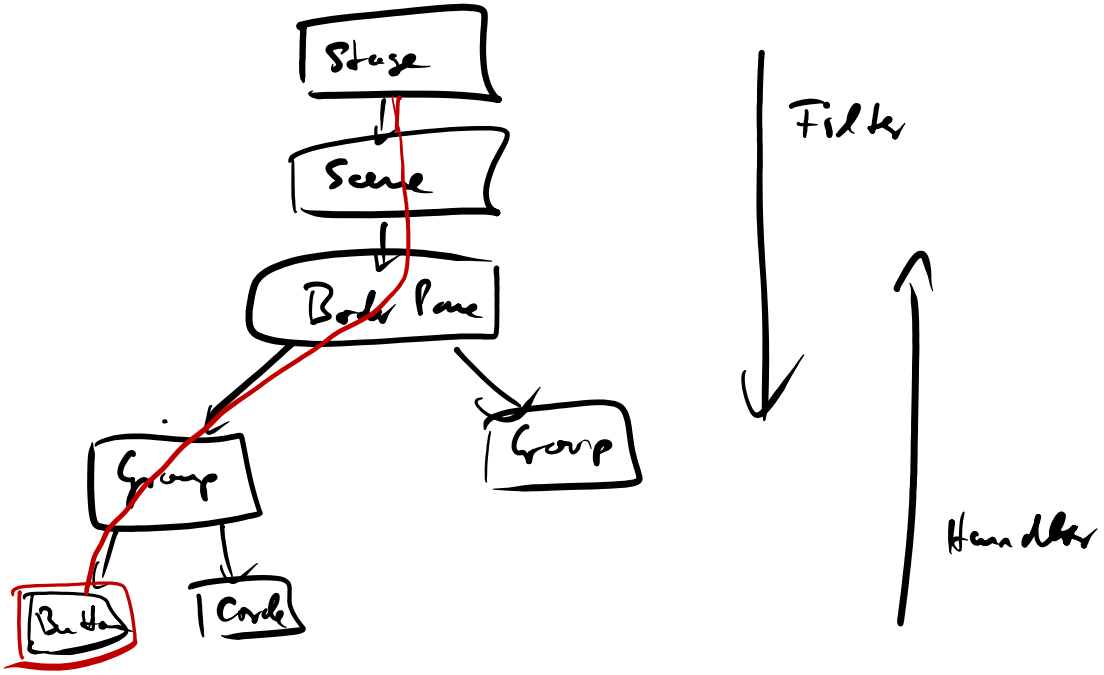
hierarchische Struktur von EventTypes (Auszug)

- `Event.ANY (alle Events)`
  - `ActionEvent.ANY`
  - `InputEvent.ANY`
    - `KeyEvent.ANY`
      - `KeyEvent.KEY_PRESSED`
      - `KeyEvent.KEY_RELEASED`
      - `KeyEvent.KEY_TYPED`
    - `MouseEvent.ANY`
      - `MouseEvent.MOUSE_PRESSED`
      - `MouseEvent.MOUSE_RELEASED`
      - …
    - …
  - `WindowEvent.ANY`
    - …

Event Verarbeitung

1. Auswahl des Targets
2. Konstruktion eines Pfades im Scene Graph
3. Event Capturing Phase
4. Event Bubbling Phase

Event Verarbeitung



Registrierung

- In der Klasse Node:
  - `addEventFilter(EventType, EventHandler)`
  - `removeEventFilter(EventType, EventHandler)`

  - `addEventHandler(EventType, EventHandler)`
  - `removeEventHandler(EventType, EventHandler)`
  - `setEventHandler(EventType, EventHandler)`

Vereinfachung

- viele vordefinierte Methoden (Auszug):

```
setOnKeyPressed (EventHandler<? super KeyEvent> value)
setOnKeyReleased
setOnKeyTyped
setOnMouseClicked
setOnMouseEntered
setOnMouseExited
setOnMouseMoved
setOnMousePressed
setOnMouseReleased
setOnSwipeDown
setOnSwipeLeft
setOnSwipeRight
setOnSwipeUp
setOnInputMethodTextChanged
...
```

Warum komplexe Hierarchie?

- Jeder bekommt nur das, was ihn interessiert
- auf jede Änderung kann reagiert werden
- spezielle Ereignisse für besondere Komponenten, z.B.:
  - TableView, TreeView
- Parameter "Event" liefert gezielte Informationen zum Event
- Eine Klasse kann mehrere Interfaces implementieren

Im Code

EventHandler implementieren und für entsprechendes Event registrieren

Implementierung eines EventHandlers über

- eigene Klasse mit Interface
- Anonyme Klasse
- Funktionsparameter (anonyme Funktion)

Probleme?

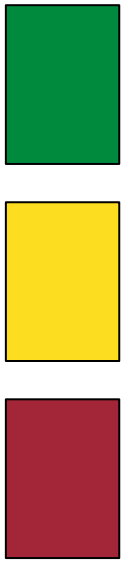
- Welche Probleme können beim Programmieren auftreten?
  - viele Aktionen zustandsabhängig
  - aktueller Zustand aber verteilt und daher nicht ganz einfach einsehbar, geschweige denn konsistent umsetzbar.

Lösungen?

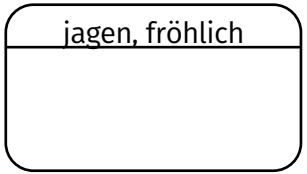
- Zustandsautomaten können helfen
- Zumindest als Dokumentation und Realisierungshilfe

Zustandsautomat...

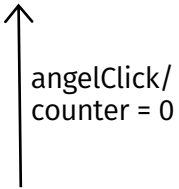
- ... kenne ich
- ... habe ich schon einmal gehört
- ... was soll das sein?



Zustandsautomaten



- Zustand



- Transition
  - Event [Condition] / Action
  - jede Kombination in Ordnung:
    E [C] /A E[C] E/A [C]/A

Syntax?

- Starttransition

Zustandsautomaten

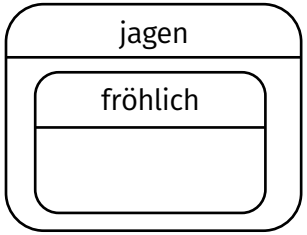
- typisches Vorgehen:
- Zustände identifizieren
- Startzustand definieren
- mögliche Transitionen von jedem Zustand nach jedem anderen Zustand überlegen
  - Komme ich dorthin?
  - Und wenn ja, wie?
  - Was kann im aktuellen Zustand alles passieren und wo komme ich dann hin?
- Optimierung durch hierarchische und parallele Zustände

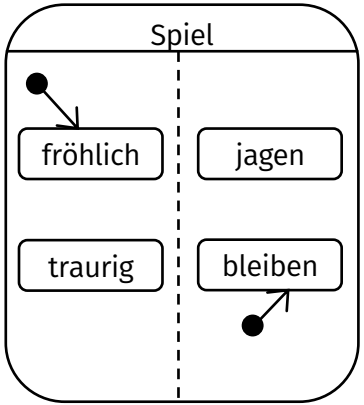
Murmelgruppe

- Murmelgruppe – 5 Minuten
- Bearbeiten Sie mit Ihrem Nachbarn (~2-3 Personen) folgende Aufgabe:

- Definieren Sie eine Zustandsmaschine für den Zustand des Smileys:
  - Der Smiley ist entweder fröhlich oder traurig, bleibt stehen („stehen-Modus“) oder springt weg („jagen-Modus“).
  - Die Stehen-/Jagen-Buttons wechseln zwischen den Modi.
  - Wenn der Angel-Button gedrückt wird, wird der Smiley fröhlich, wenn der Devil-button gedrückt wird, wird der Smiley traurig.
  - Wenn min. 5 mal hintereinander der Devil-button gedrückt wird, verbleibt der Smiley im „jagen-Modus“ und es wird „oh noo“ angezeigt. Wenn der Angel-Button gedrückt wird, wird dieser Modus wieder verlassen.

hierarchische Zustandsautomaten

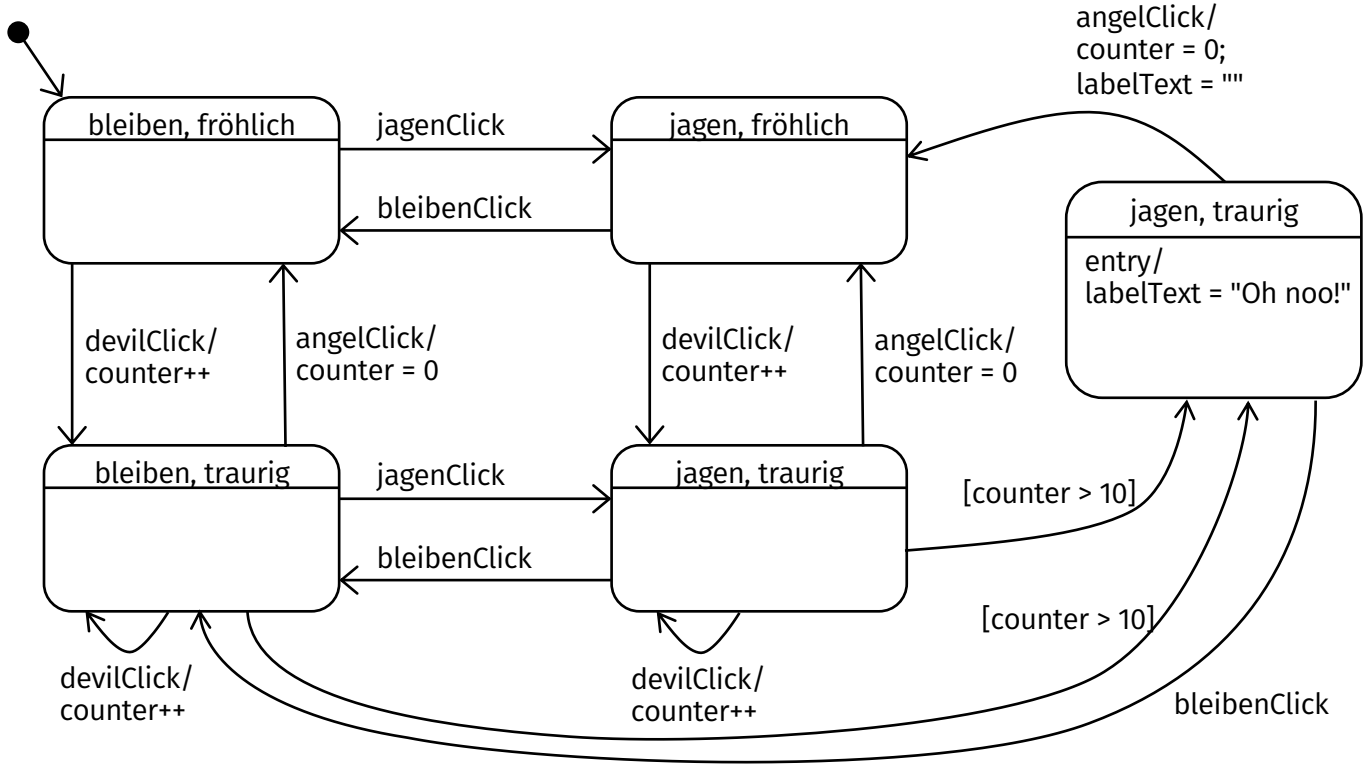




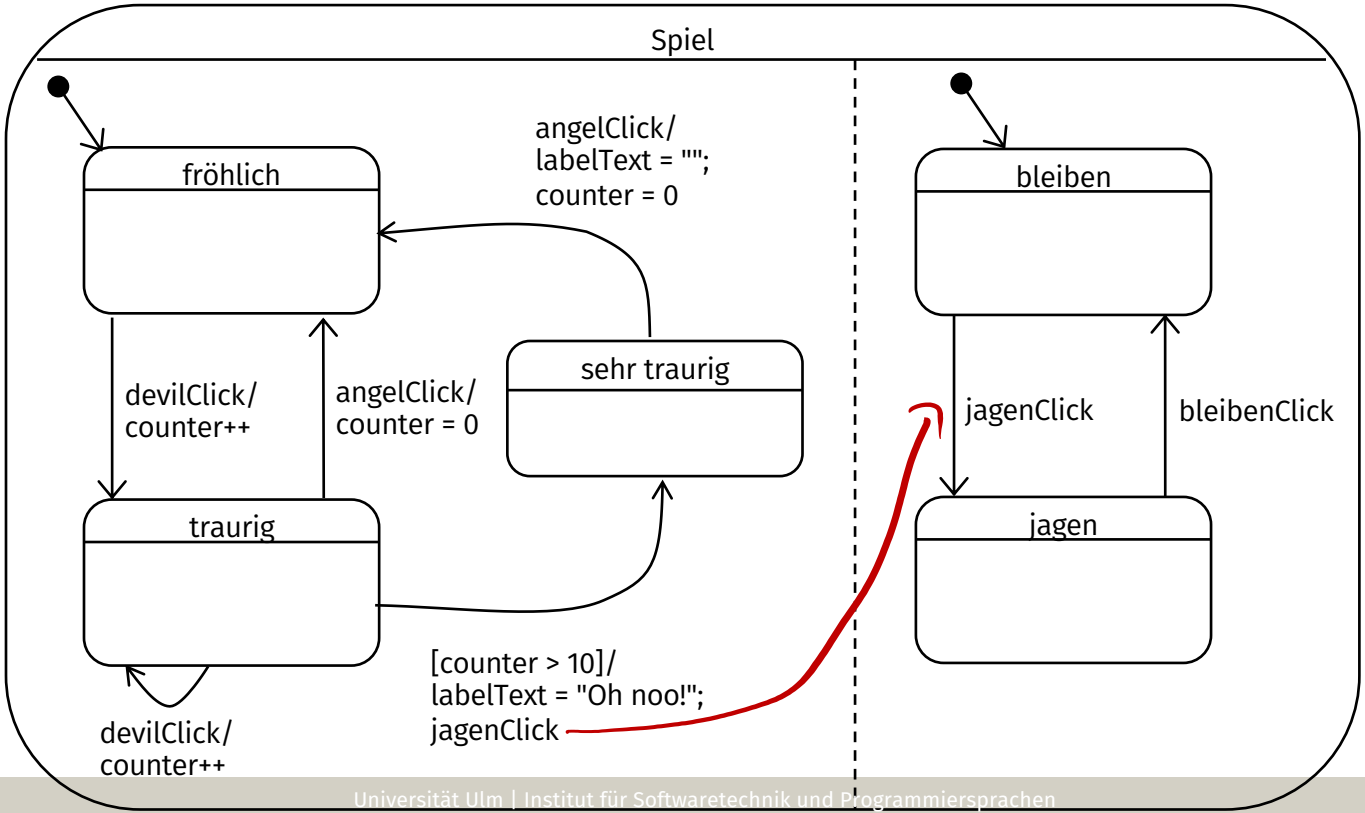
- hierarchische Zustände:
  ➔ weniger Transitionen

- parallele Zustände:
  ➔ weniger Zustände

Zustandsautomaten

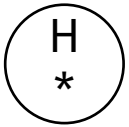


hierarchische Zustandsautomaten



hierarchische Zustandsautomaten – Specials

- History-Connector



- Deep-History Connector

- entry/exit-Ereignisse
  entry/<Action>
  exit/<Action>

hierarchische Zustandsautomaten – Probleme?

- Prioritäten
- Zeit
  - Übergang nach einer Sekunde
  - Wie lange dauert ein Übergang?

- Semantik(en)!
- Vergleich von (damals) 21 Varianten:
  Von der Beeck, Michael. "A Comparison of Statecharts Variants." Formal techniques in real-time and fault-tolerant systems. Springer Berlin Heidelberg, 1994.

Lernziele

- Events
- Event Handling
- Zustands
- Zustandsautomaten