Programmiermethodik

State, Observer & MVC

SS 2002

Thomas Kühne

kuehne@informatik.tu-darmstadt.de

http://www.informatik.uni-mannheim.de/informatik/softwaretechnik

State pattern

Das Problem

- Das Verhalten eines Objekts ist abhängig von seinem Zustand, der sich zur Laufzeit ändert
- Mögliche Lösungen
  - » Objekt kann seinen Typ (Klasse) dynamisch wechseln (ist in Java nicht möglich)
  - » Objektzustand wird z.B. durch Konstaten aufgezählt und Fallunterscheidungen werden benutzt, um je nach Zustand das richtige Verhalten auszuwählen

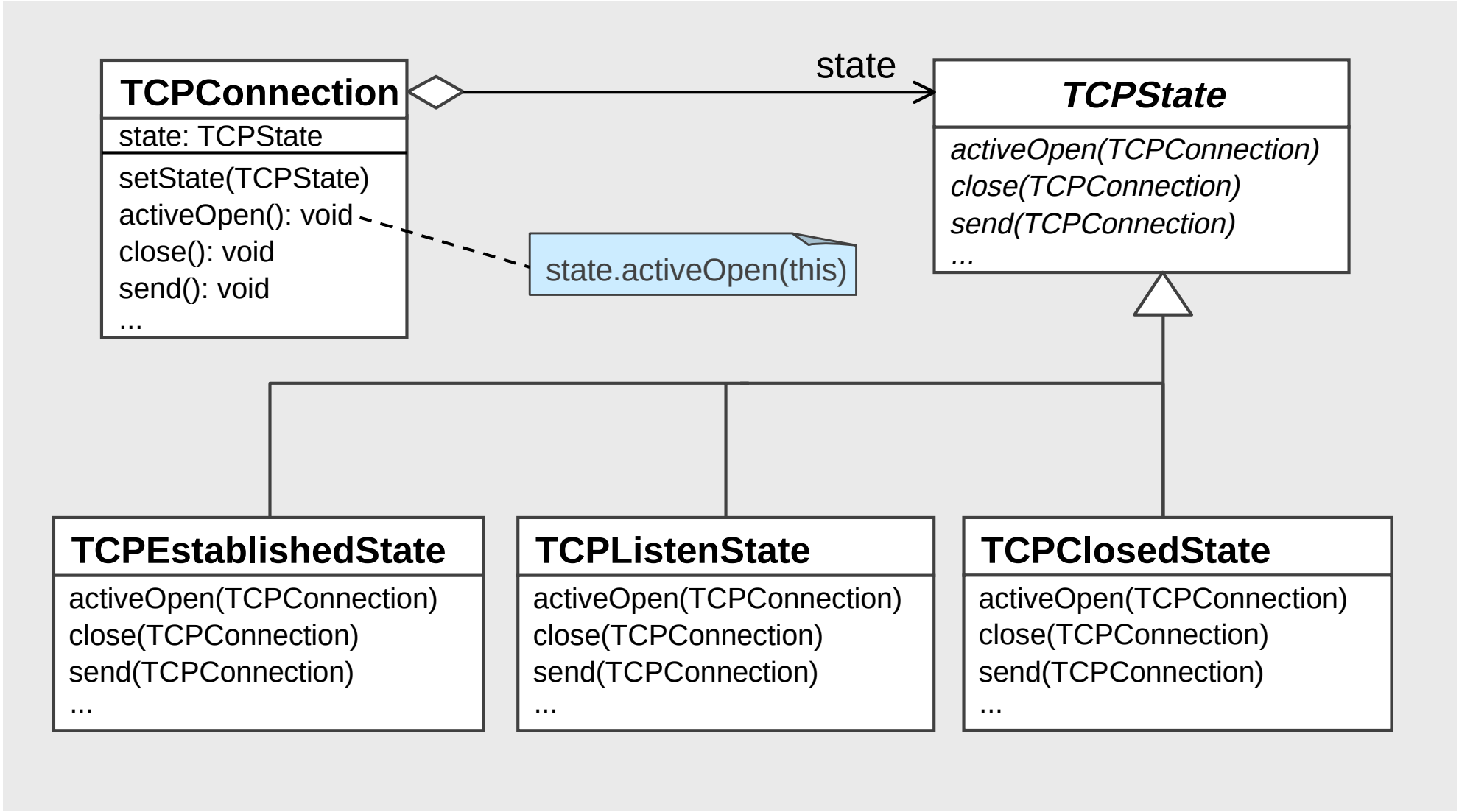
Fallunterscheidungen müssen aber bei jeder Erweiterung der Zustandsmenge – an mehreren Stellen – geändert werden

State pattern

Lösung

- Auslagern des zustandsabhängigen Verhaltens in ein dafür zuständiges Objekt
  - » Objektwechsel statt Typwechsel
- Für jeden möglichen Zustand ein eigenes Zustandsobjekt vorsehen, welches das entsprechende Verhalten implementiert
  - » verschiedene Aggregate statt verschiedener Unterklassen
- Bei Zustandswechseln das ausgelagerte Objekt austauschen
  - » manuelles Auswechseln statt spätes Binden

State: Beispiel



State: Beispiel

- Immer wenn die Verbindung ihren Zustand ändert, wechselt TCPConnection das Zustandsobjekt
- Wenn der Status der Verbindung, z.B., von "erstellt" zu "lauschen" wechselt, dann ersetzt TCPConnection seine TCPEstablishedState-Instanz mit einer TCPListenState-Instanz

State: Beispielcode

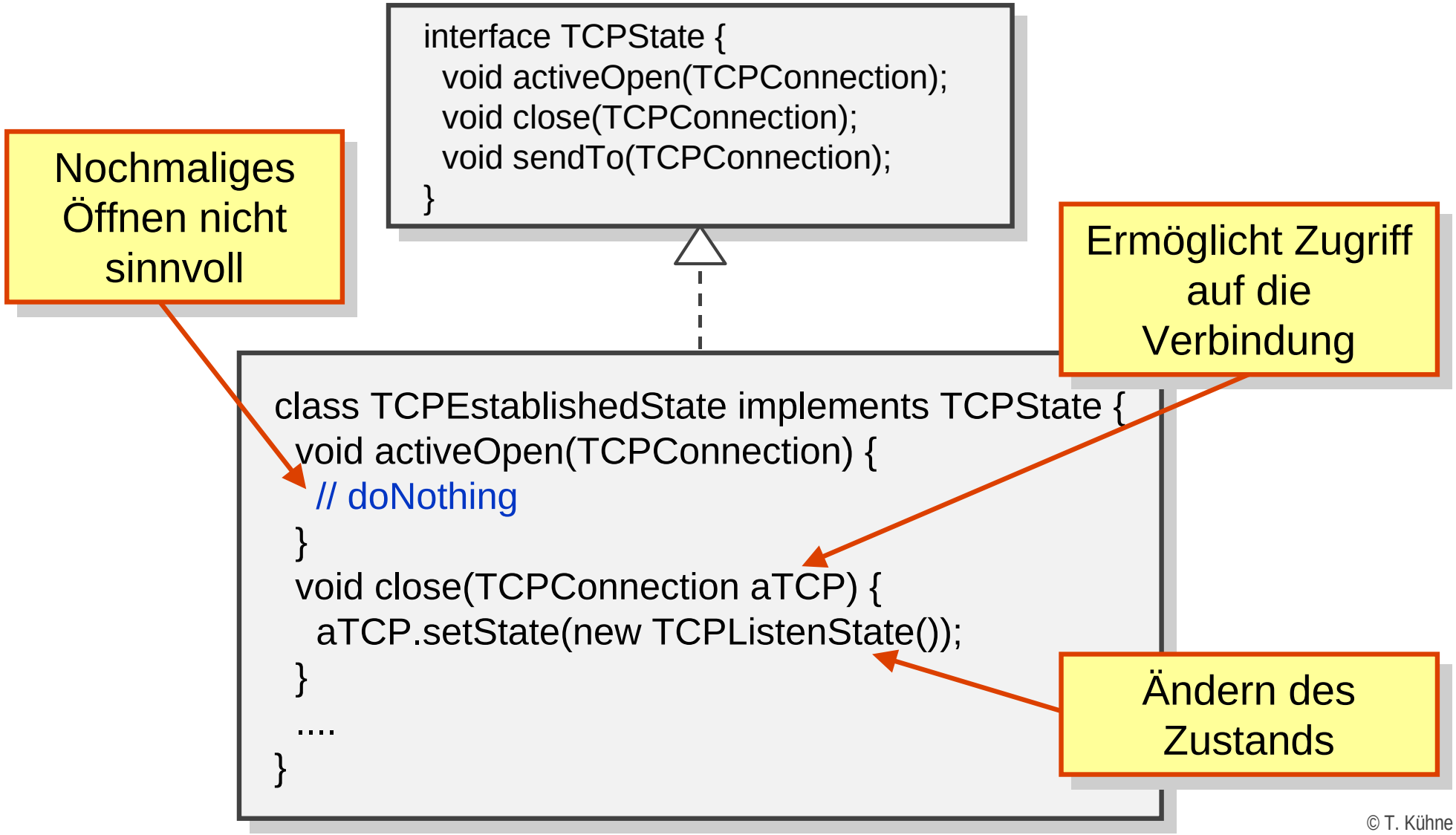
```
class TCPConnection {
    private TCPState state;

    public TCPConnection()
      // Set the connection‘s default state
      setState(new TCPClosedState());
    }
    public void activeOpen() {
      state.activeOpen(this);
    }
    public void close() {
      state.close(this);
    }
    ...
}
```

Zustandsinstanz

Setzen des Anfangszustands

Weiterleiten der zustandsabhängigen Operation

State: Beispielcode



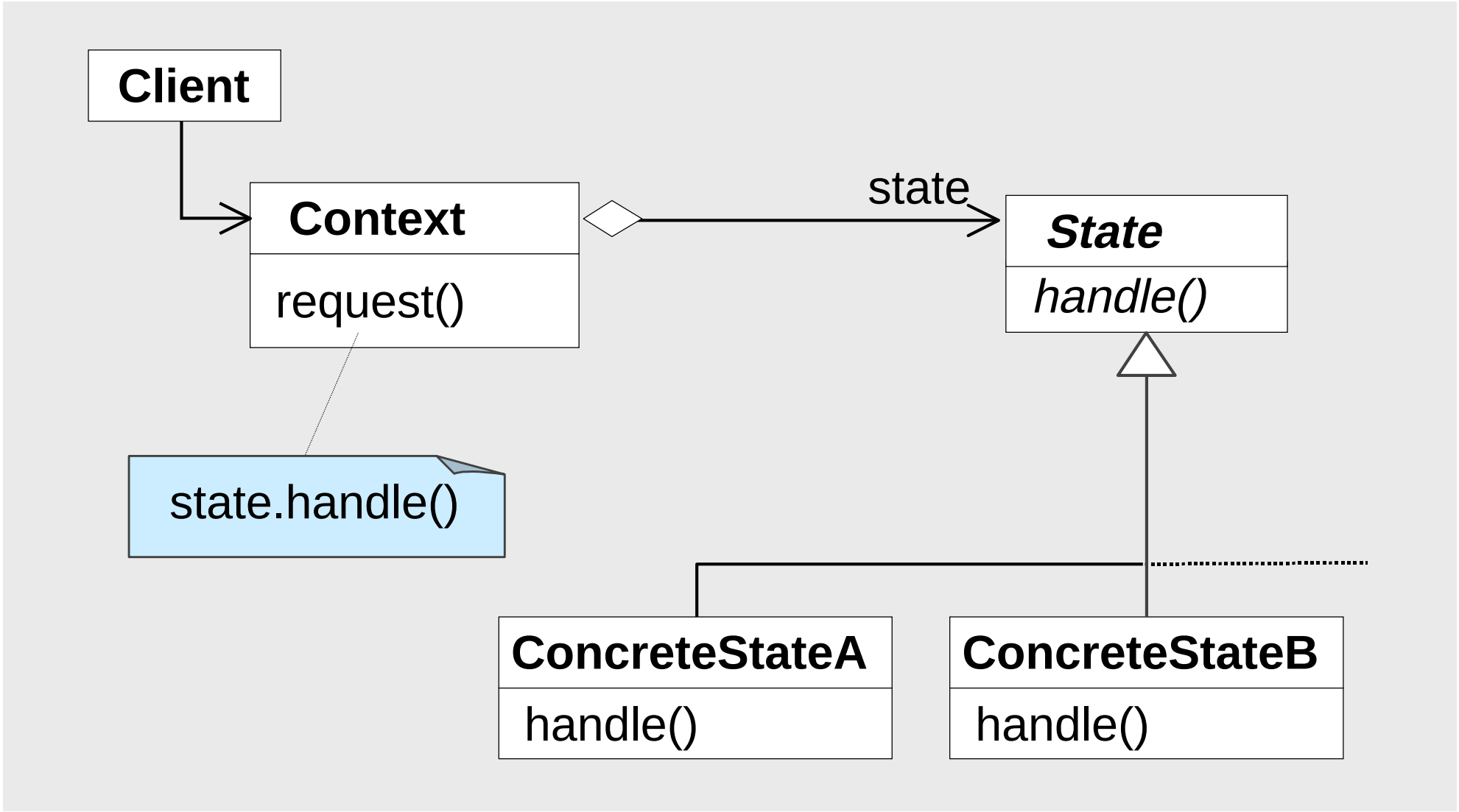
State: Beispielcode

```
interface TCPState {
  void activeOpen(TCPConnection);
  void close(TCPConnection);
  void sendTo(TCPConnection);
}
```

```
class TCPListenState implements TCPState {
    void activeOpen(TCPConnection) {
    // doNothing
    }
    void close(TCPConnection aTCP) {
      aTCP.setState(new TCPClosedState());
    }
  }
```

Geändertes Verhalten gegenüber "EstablishedState"

State: Struktur



State: Kollaborationen

- *Context* delegiert zustandsspezifische Anfragen an das aktuelle *State* Objekt
- Ein *Context* kann sich selbst als Argument an ein *State* Objekt übergeben. Damit können *State* Objekte auf den Kontext zugreifen (falls nötig)
- *Context* ist die primäre Schnittstelle für Klienten. Für Klienten ist das Vorhandensein von *State* Objekten transparent
- Sowohl *Context* als auch *State* Objekte können darüber entscheiden welcher Zustand welchem folgt und unter welchen Umständen dies geschieht

State: Konsequenzen

- Partitionierung und Lokalisierung von zustandsabhängigen Verhalten
  - » Verhalten, das mit einem bestimmten Zustand assoziiert ist, wird in einer Klasse lokalisiert
  - » Es ist einfach (durch lokale Erweiterung) einen neuen Zustand hinzuzufügen
  - » Die Duplikation von fragilen Fallunterscheidungen wird vermieden

State: Implementierung

- Zustandsübergänge
  1. Können durch *Kontext* festgelegt werden
  2. Festlegung durch *State* Objekte ist sehr flexibel erfordert aber eine *Kontext* Schnittstelle, um dies zu ermöglichen
  3. Werden die Übergänge in einer Tabelle (Zustand x Eingabe ⇒ Zustand) abgelegt, so können sie ohne Programmieren einfach durch Datenmanipulation geändert werden

Observer pattern

Das Problem

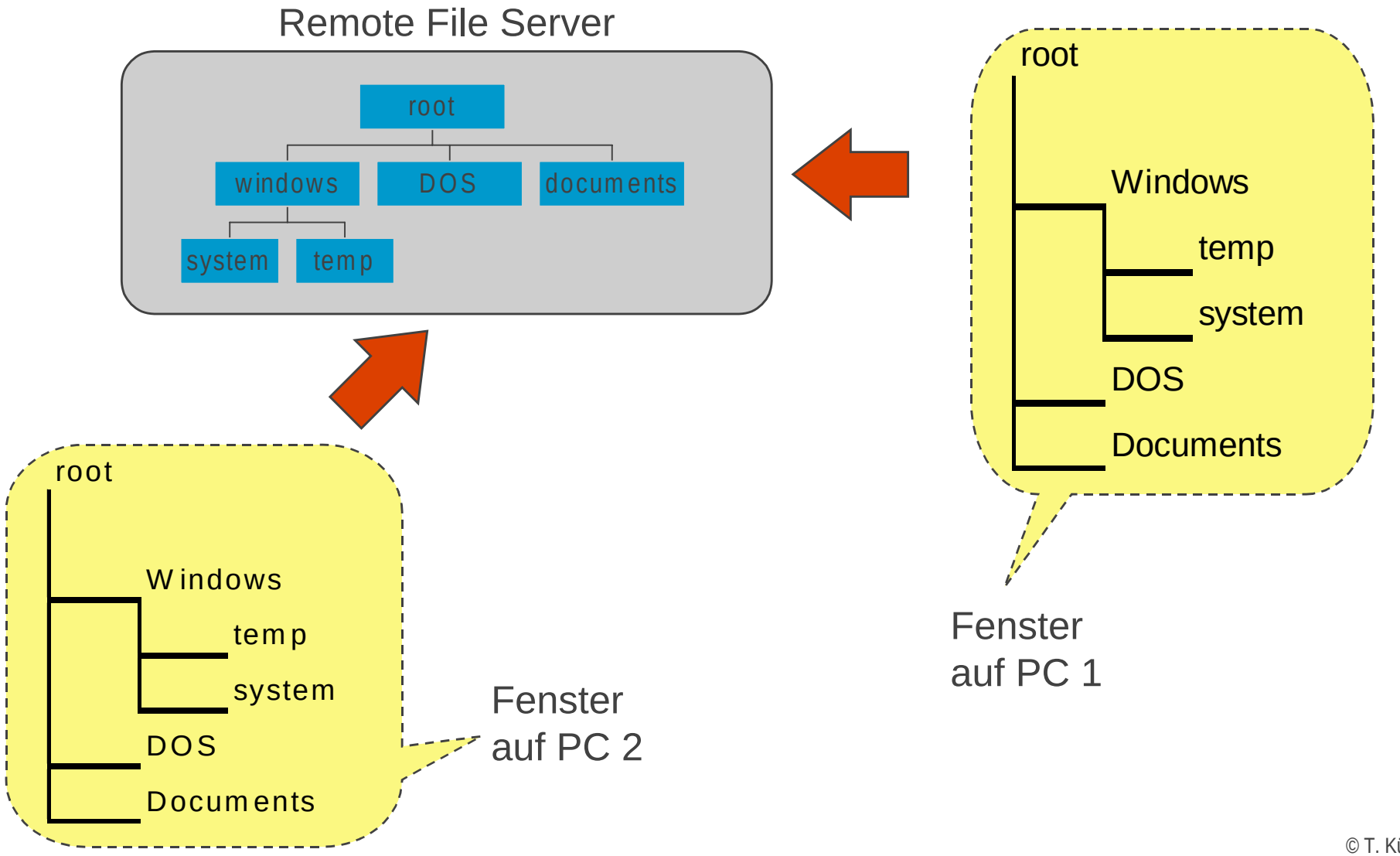
- Objektorientierte Dekomposition löst Anforderungen durch Verteilung von Verantwortlichkeiten auf verschiedene Objekte
- Zustandsänderungen in einem Objekt müssen daher u.U. an kollaborierende Objekte weitergeleitet werden

Eine direkte Benachrichtigung würde die kollaborierenden Objekte stark aneinander koppeln und sie damit für eine unabhängige Benutzung unbrauchbar machen

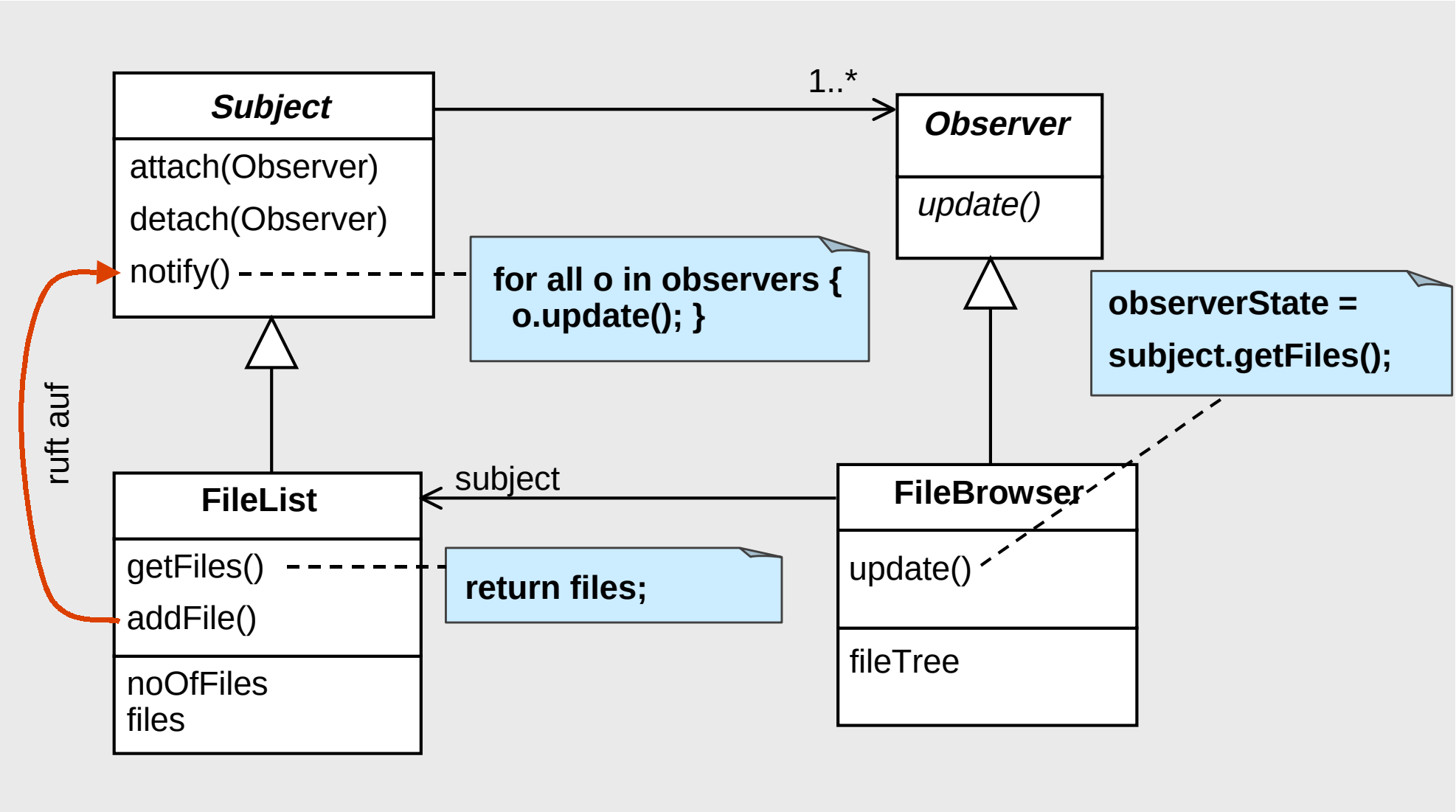
Observer pattern

- Aufgabenstellung: Entkopplung eines Objekts (typischerweise ein Datenmodell) von den interessierten Klienten
- Zu berücksichtigen:
  - » Das Subjekt soll seine Beobachter nicht kennen
  - » Die Anzahl der Beobachter ist nicht vorherbestimmt
  - » Möglicherweise werden in der Zukunft neue Beobachter hinzugefügt
  - » Ein Lösung bei der Beobachter ständig selbstätig den Zustands des Subjekts abfragen (Polling) ist zu ineffizient

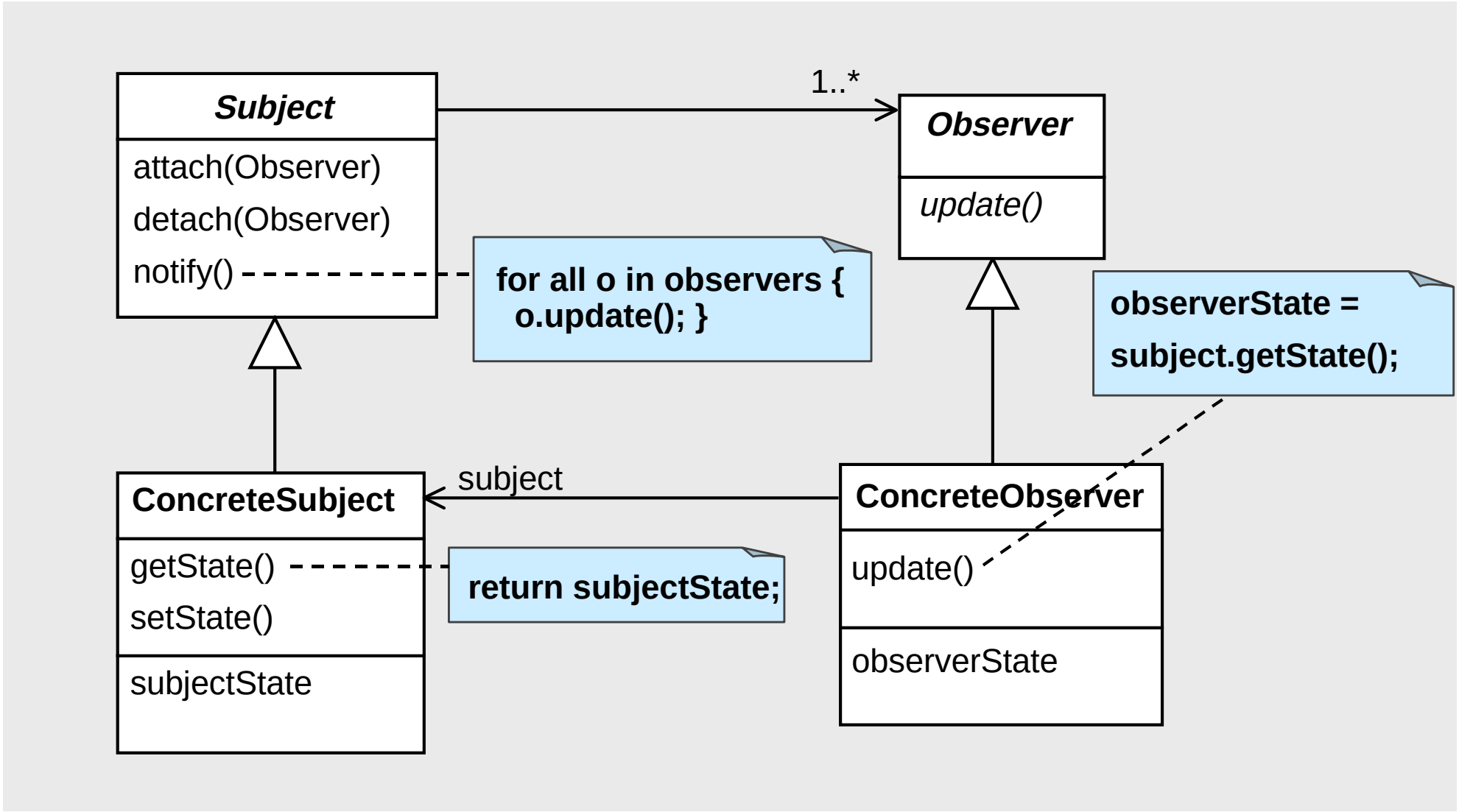
Observer: Beispiel



Observer: Beispielstruktur



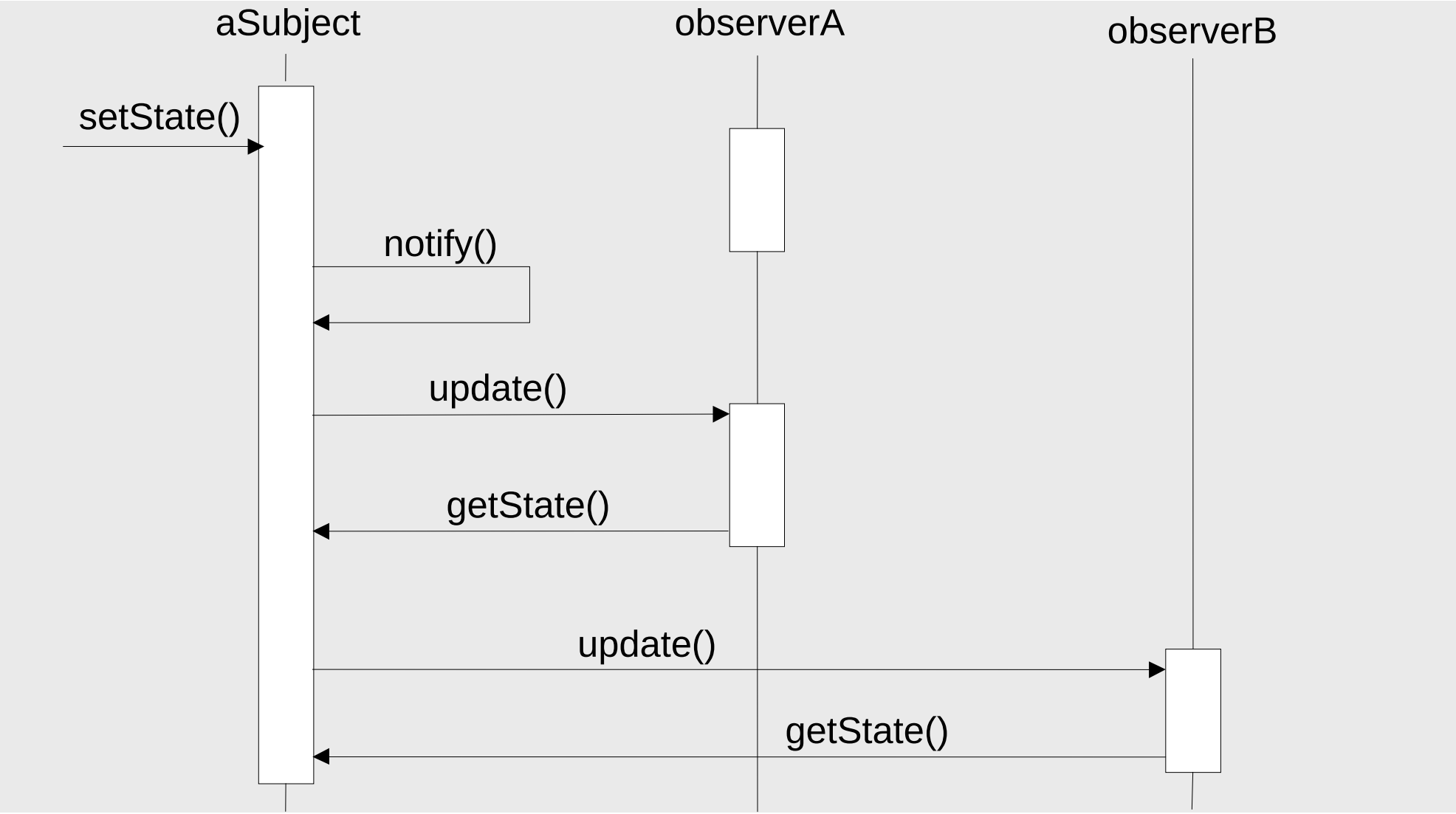
Observer: Beispielstruktur



Observer: Beteiligte

- **Subject**
  - » besitzt eine Liste von *Observer* Objekten und bietet eine Schnittstelle zum Verwalten und Benachrichtigen (notify) von *Observer*-Objekten
- **Observer**
  - » definiert eine update Schnittstelle zur Benachrichtigung von *Observer* Objekten über Subjektänderungen
- **ConcreteSubject**
  - » speichert Zustand und löst die Benachrichtigung von *Observer* Objekten durch Aufruf der notify Operation aus
- **ConcreteObserver**
  - » hat eine Referenz zu einem *ConcreteSubject* und besitzt eigenen (Darstellungs-) Zustand, der mit dem Subjektzustand gekoppelt ist
  - » implementiert die *Observer* "update" Operation, die den Darstellungszustand auf den neusten Stand bringen soll

Observer: Ablauf



Observer: Java Unterstützng

- Die Java Bibliothek unterstützt das Observer pattern:
  - » Observer ⇒ java.util.Observer
  - » Subject ⇒ java.util.Observable

interface java.util.Observer

public abstract void update(Observable o, Object arg)

This method is called whenever the observed object is changed. An application calls an observable object's notifyObservers method to have all the object's observers notified of the change.

Parameters:
o - the observed object.
arg - an argument passed to the notifyObservers method

Observer in Java

class java.util.Observable ("Subject")

addObserver(Observer)
Adds an observer to the observer list.

deleteObserver(Observer)
Deletes an observer from the observer list.

setChanged()
Sets a flag to note an observable change

clearChanged()
Clears an observable change.

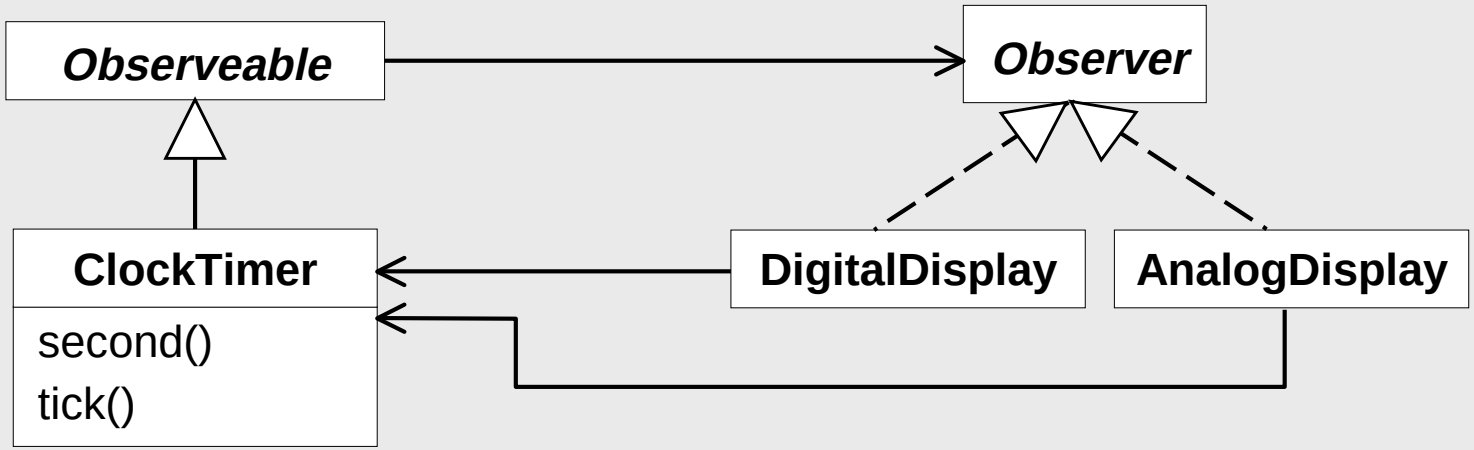
notifyObservers()
Notifies all observers if an observable change occurs.

notifyObservers(Object)
Notifies all observers of the specified observable change which occurred.

class java.util.Observer

update(Observable o, Object arg)
Update own state to reflect changes in the subject's state

Observer: Beispielcode




```
class ClockTimer extends Observable {
    private int seconds;

    public ClockTimer() { seconds=0; }
    public int seconds() { return seconds; }

    public void tick() {
        seconds=(seconds+1)%60;
        setChanged();
        notifyObservers();
        }
    }
```

```
class DigitalDisplay implements Observer {
  private boolean on;

  public void update(Observable subject,
                     Object message) {

    on = (subject.seconds()%1)==0;
    // display colon depending on "on" status
    ...
   }
```

Observer: Konsequenzen

- Entkopplung von Subject und Observer Komponenten
- Unterstützung für "Broadcast" Kommunikation
  - » Beim Aufruf von "notify" muß man den Empfänger nicht kennen
  - » Zahlreiche Empfänger können mit einem Schlag erreicht werden
- Unerwartete Updates
  - » Gefahr von kaskadierenden Updates
  - » Unter Umständen kann es schwierig sein festzustellen was sich im Subjekt geändert hat
  - » Teilweise unnötiges Informieren von Beobachtern, die nur an bestimmten Änderungen interessiert sind
  - » Eine individuelle Benachrichtigung von Beobachtern ist durch die festgelegte Schnittstelle (update) nicht möglich

Observer: Implementierung

- Wer veranlaßt die Benachrichtigung?
  - » Subjekt: Klient muß sich nicht kümmern
  - » Klient: Nicht jede feingranulare Subjektänderung muß weitergeleitet werden
- Bestimmung des geänderten Aspekts
  - » pull model: Beobachter fragt beim Subjekt nach
  - » push model: Subjekt liefert zusätzliche Aspektinformationen bei der update Nachricht

Model-View-Controller

MVC Dreiteilung separiert:

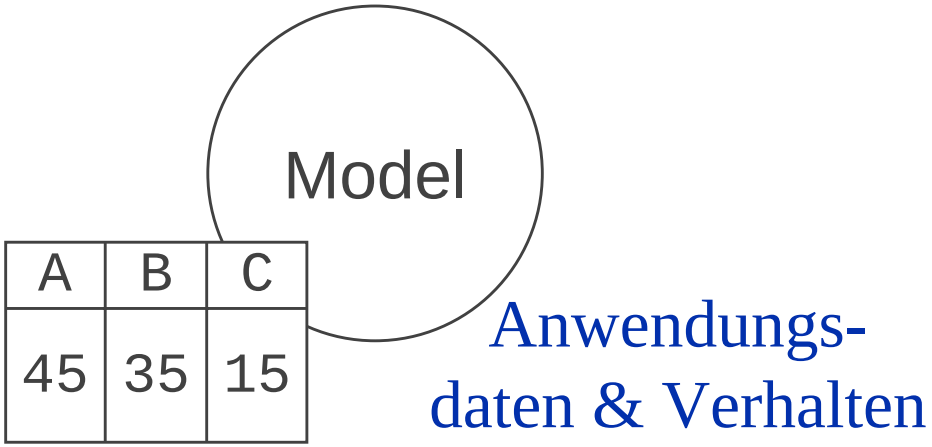
- Komponenten, die ein Modell der Anwendungsdomäne darstellen
  - » Konten, Portfolios, Aktienverlauf
- Präsentation der Daten an den Benutzer
  - » Kontoauszug, Performanzgraph
- Bedienungselement mit denen der Benutzer interagiert
  - » Knöpfe, Menüs, direkte Manipulation

Model-View-Controller

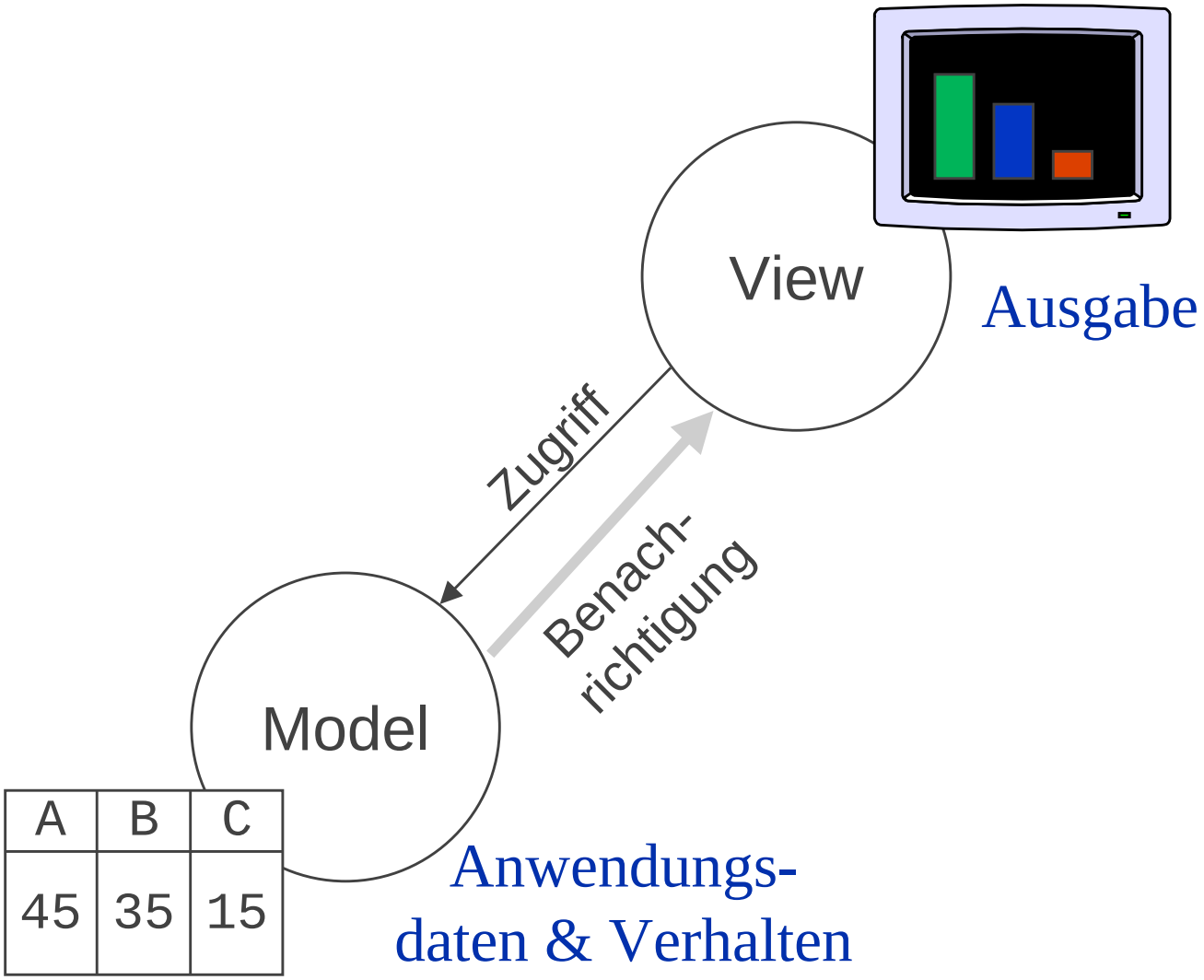
CRC Sicht

- Controller
  - » interpretiert Benutzereingaben
  - » beeinflußt ggf. die Darstellung (View)
- View
  - » stellt die Daten (Model) dar
  - » transformiert Koordinaten
- Model
  - » hält die Anwendungsdaten
  - » benachrichtigt Beobachter (View) bei Änderungen

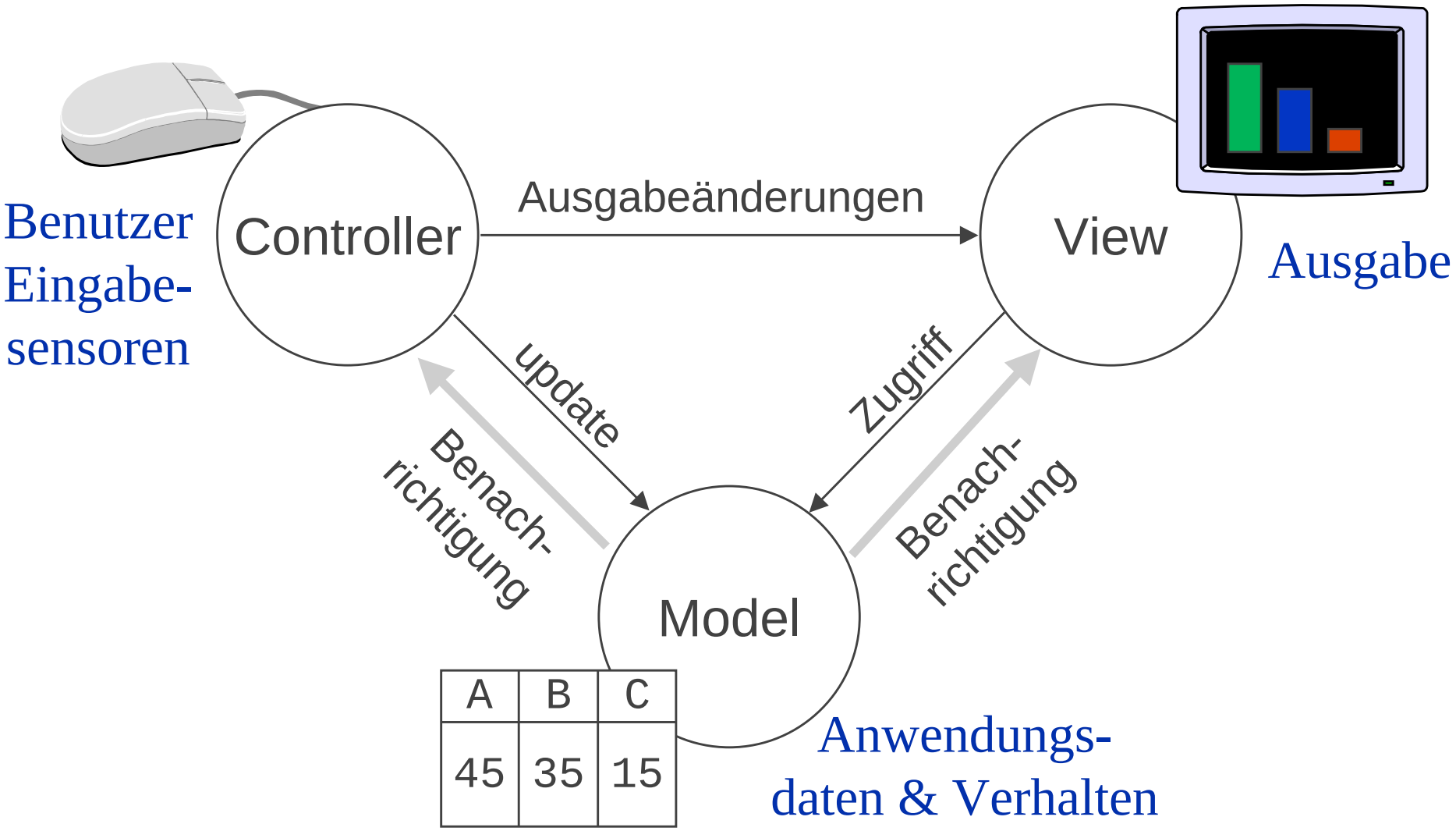
Model-View-Controller



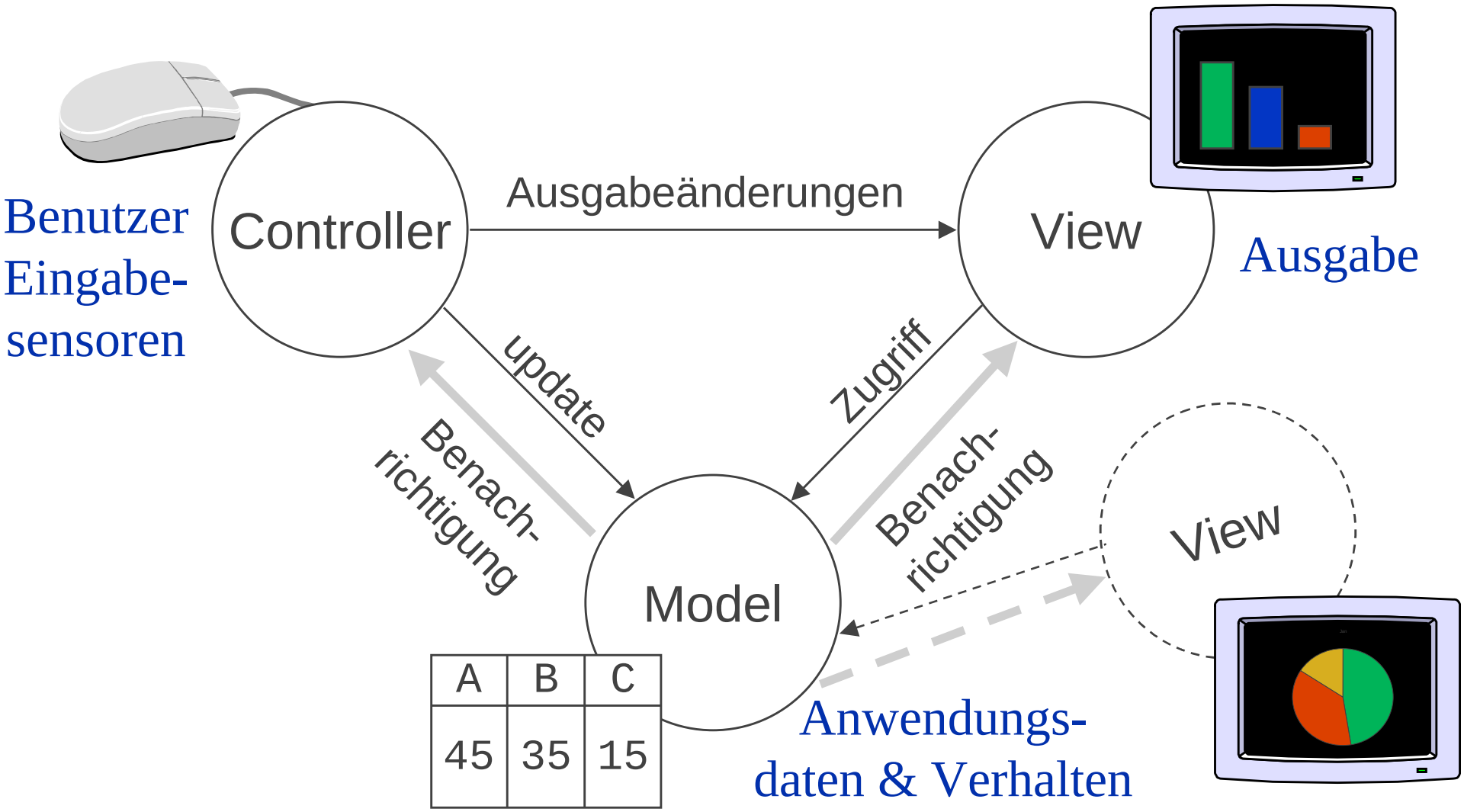
Model-View-Controller



Model-View-Controller



Model-View-Controller



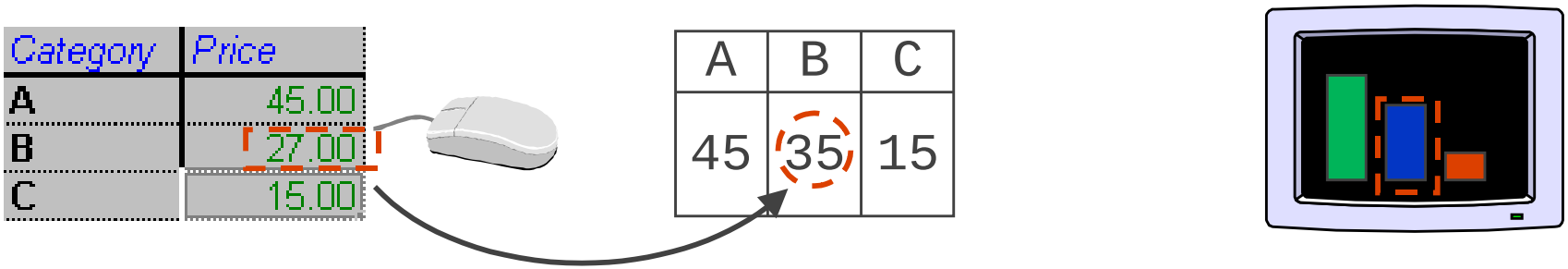
Model-View-Controller

Category	Price
A	45.00
B	27.00
C	15.00

A	B	C
45	35	15

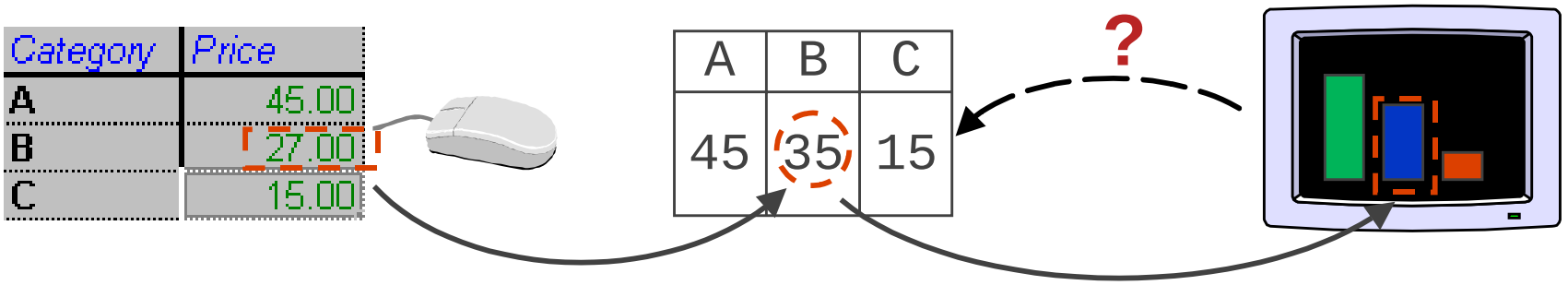
1. Benutzer klickt Zelle an
2. Zellen Controller behandelt Edierung
3. Controller schickt Model die Nachricht `newB(data)`

Model-View-Controller



1. Benutzer klickt Zelle an
2. Zellen Controller behandelt Edierung
3. Controller schickt Model die Nachricht `setB(data)`
4. Model ändert sich
5. Model schickt `notify` Nachricht an sich selbst

Model-View-Controller



1. Benutzer klickt Zelle an
2. Zellen Controller behandelt Edierung
3. Controller schickt Model die Nachricht `newB(data)`
4. Model ändert sich
5. Model schickt `notify` Nachricht an sich selbst
6. View erhält `update` Nachricht
7. View fragt Model nach Werten
8. View malt sich selbst neu

Model-View-Controller

Vorteile

- separiert Model und Darstellung
  - » Modell hängt nicht mehr vom verwendeten GUI ab
- Viele Sichten auf geteilte Daten möglich
- Sichten lassen sich leicht austauschen und hinzufügen
- Code in MVC-Oberklassen wird wiederverwendet

Model-View-Controller

Ereignisse statt Nachrichten

- Der Aufruf der "notify" Operation entspricht eher dem Auslösen eines Ereignisses (einem Signal) als dem Aufruf einer Methode
  - » Empfänger ist nicht bekannt
  - » beliebige Empfänger, die sich vorher dafür registrieren, können auf das Ereignis reagieren
- Dieses Paradigma läßt sich auch für die Controller-Seite verwenden ($\rightarrow$ SWING)
  - » Benutzerinteraktionen als Ereignisse, für die sich Eingabekomponenten anmelden können
  - » Üblicherweise keine Trennung zwischen Controller & View

Architektur & Softwaretechnik

Software designers are in a similar position to architects and civil engineers, particularly those concerned with the design of large heterogeneous constructions, such as towns and industrial plants. It therefore seems natural that we should turn to these subjects for ideas about how to attack the design problem. As one single example of such a source of ideas I would like to mention Christopher Alexander:

Notes on the Synthesis of Form

Peter Naur

Frühe Grundlage für die Software-Entwurfsmuster