

Einführung in die Informatik II
Ereignis-basierte
Programmierung

Prof. Bernd Brügge, Ph.D
Institut für Informatik
Technische Universität München

Sommersemester 2001

18.- 20. Juni 2001

Inhalt dieses Vorlesungsblockes

- ❖ Ereignis-basierte Programmierung
 - ◆ Modellierung von *Ereignissen* und *Ereignisempfängern*
 - ◆ Bindung von Ereignissen an Ereignisempfänger
- ❖ Ereignis-basierte interaktive Systeme
- ❖ Benutzerschnittstellen-Programmierung
 - ◆ Modellierung von *graphischen Komponenten*
 - ◆ Java-Klassen für graphische Komponenten
- ❖ Applets
- ❖ Notation zur Modellierung der dynamischen Aspekte eines Systems
 - ◆ UML-Sequenzdiagramme
- ❖ Beobachter-Muster
 - ◆ Java's Ereignismodell beruht auf dem Beobachter-Muster

Ziele dieser Vorlesung

- ❖ Sie verstehen die Grundkonzepte der Ereignis-basierten Programmierung (*Ereignisse, Ereignisempfänger, Ereignisempfänger-Schnittstelle*)
- ❖ Sie können die *Bedienoberfläche* eines interaktiven Systems modellieren und implementieren
- ❖ Sie können Ereignis-basierte interaktive Systeme als *Applets* implementieren.
- ❖ Sie verstehen, dass *Applets die Konzepte von zwei Programmierstilen* verwenden:
 - ◆ Objekt-orientiert: Vererbung, dynamischer Polymorphismus, Methodenüberschreibung
 - ◆ Ereignis-basiert: Ereignisse, Ereignisempfänger-Schnittstellen, Ereignisempfänger
- ❖ Sie können die dynamischen Eigenschaften von Informatik-Systemen mit *Sequenzdiagrammen* modellieren

Warum Ereignis-basierte Programmierung?

- ❖ Wir haben bisher schon eine Art von Ereignissen kennengelernt: Ausnahmen.
 - ◆ Ausnahmen modellieren unvorgesehene Abweichungen im geplanten normalen Kontrollfluss.
- ❖ Bei Ereignis-basierten Systemen ist das Entstehen und die Behandlung von Ereignissen so wichtig, dass sie als normal angesehen werden.
 - ◆ Beispiele von Ereignissen: Eine Mausbewegung, ein Mausklick, das Drücken einer Taste.
- ❖ In der Ereignis-basierten Programmierung *bestimmen Ereignisse den Kontrollfluss* von Informatik-Systemen.
- ❖ Wie modellieren wir Ereignis-basierte Programme?

Ereignis-basierte Systeme

❖ **Definition Ereignis-basiertes System:** Der Kontrollfluss (d.h. die Reihenfolge der Ausführungsschritte) eines Informatik-Systems wird hauptsächlich durch Ereignisse und nicht durch Kontrollstrukturen oder Funktionsaufrufe bestimmt.

❖ Beispiele von Ereignissen:

- ◆ Mausklick
- ◆ Tastendruck
- ◆ Einschieben einer Diskette in ein Laufwerk
- ◆ Schließen eines Fensters
- ◆ Ampel schaltet auf rot
- ◆ Benzintank wird leer
- ◆ Piepen eines Weckers

Ereignisse aus der Lösungsdomäne

Ereignisse aus der Anwendungsdomäne

Kategorisierung von Informatik-Systemen

(aus Info I-Vorlesung 2)

❖ **Wiederholung:** Die Aufgaben, die wir mit Informatik-Systemen bearbeiten, haben wir in 5 Klassen eingeordnet:

- ◆ 1. Berechnung von Funktionen
- ◆ 2. Interaktive Systeme
- ◆ 3. Prozessüberwachung
- ◆ 4. Eingebettete Systeme
- ◆ 5. Adaptive Systeme



Offene Systeme

❖ Offene Systeme lassen sich oft sehr gut Ereignis-basiert modellieren und implementieren.

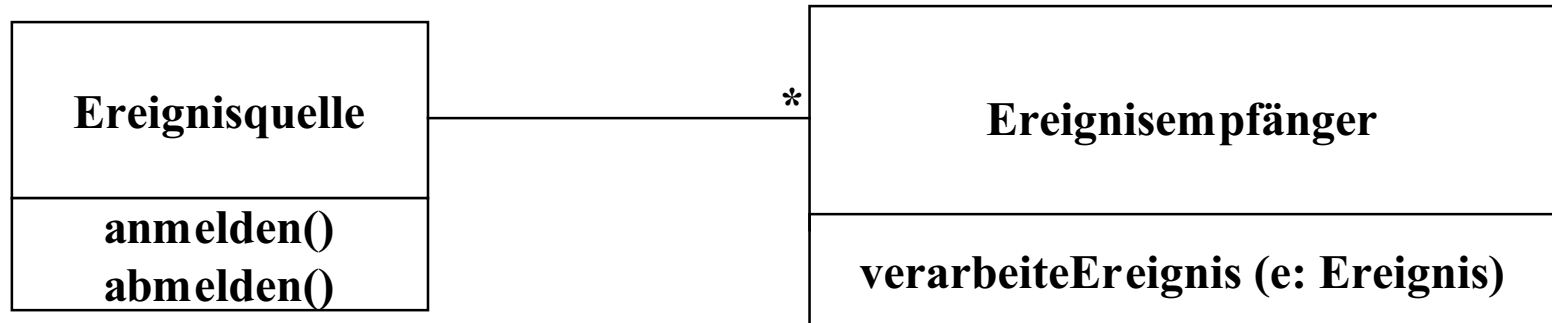
Was ist ein Ereignis?

"Wenn in einem Wald ein Baum umfällt, und niemand hört zu, gibt es dann einen Knall?"

Modellierung von Ereignissen

- ❖ **Realität:** Ein Ereignis ist etwas, was in einem Raum und/oder zu einer Zeit passiert und was innerhalb einer bestimmten Domäne (Anwendungsdomäne, Lösungsdomäne) interessant ist.
- ❖ **Modell:**
 - ◆ Die Menge aller Ereignisse ist durch die Klasse **Ereignis** repräsentiert.
 - ◆ Wann immer ein Ereignis stattfindet, wird ein Objekt vom Typ **Ereignis** instantiiert. Den Erzeuger des Ereignis-Objektes nennen wir *Ereignisquelle (event source)*. Die Instanz nennen wir *Ereignisobjekt*, oft auch kurz *Ereignis* genannt.
 - ◆ Für jedes Ereignisobjekt gibt es einen oder mehrere *Ereignisempfänger (event listener)*, die an dem Ereignis interessiert sind und benachrichtigt werden, nachdem das Ereignis stattgefunden hat.

Modellierung von Ereignissen

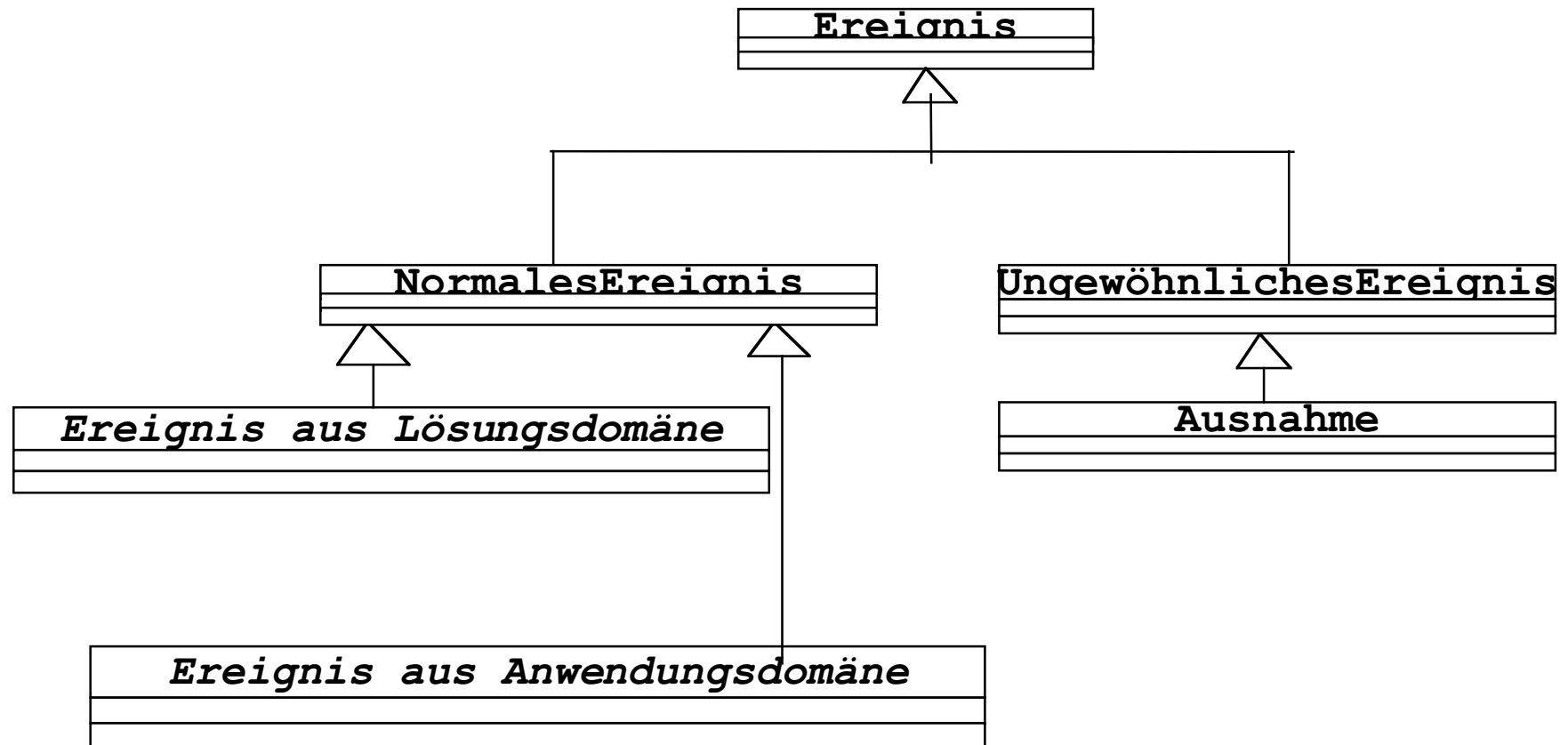


- ❖ Die Verbindung zwischen Ereignisquelle und Ereignisempfänger ist dynamisch.
 - ◆ Ein Ereignisempfänger meldet sich bei der Ereignisquelle mit dem Aufruf der Operation **anmelden ()** an, oder mit dem Aufruf der Operation **abmelden ()** ab.
- ❖ Findet ein Ereignis statt, dann geht die Ereignisquelle nacheinander durch die Menge aller angemeldeten Ereignisempfänger und ruft für jeden Ereignisempfänger die Methode **verarbeiteEreignis ()** auf, wobei das für dieses Ereignis erzeugte Ereignis-Objekt **e** als Argument übergeben wird.

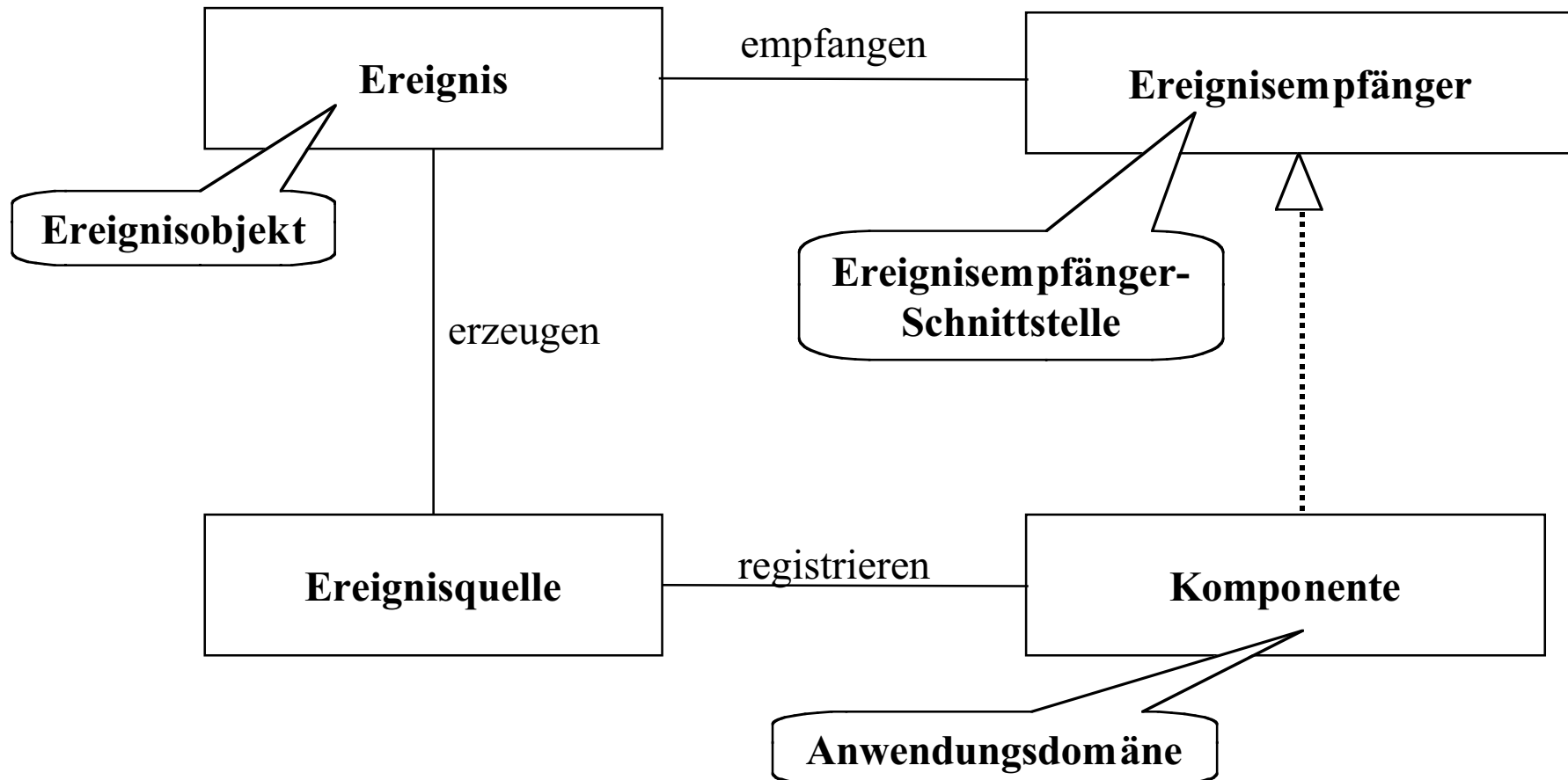
Beispiel

- ❖ Ereignis: Einschieben einer DVD ins Laufwerk
 - ◆ Attribute des Ereignisses: Regionalcode, Name des Films
- ❖ Das Ereignis hat drei Ereignisempfänger.
 1. **Betriebssystem**: erkennt die DVD als neuen Datenträger und ruft das DVD-Spieler-Programm auf, wobei das Ereignisobjekt als Parameter übergeben wird.
 2. **Fenstersystem**: erzeugt ein DVD-Piktogramm mit dem Namen des Films
 3. **DVD-Spieler**: Überprüft den Regionalcode, und fängt, wenn alles ok ist, automatisch an, den Film abzuspielen.

Modellierung von Ereignissen



Allgemeine Struktur von Ereignis-basierten Systemen



Berechnung von Funktionen vs. Interaktive Systeme

- ❖ Bei der Modellierung von Systemen der Klasse "*Berechnung von Funktionen*" haben wir uns auf die Klassen der Anwendungsdomäne konzentriert, und diese dann mit Klassen aus der Lösungsdomäne implementiert.
 - ◆ Diese System-Klasse wird selten Ereignis-basiert modelliert (abgesehen vom Auftreten von Ausnahmen).
- ❖ Bei der Modellierung von interaktiven Systemen brauchen wir mindestens zwei Modelle:
 1. Modell der Anwendungsdomäne
 2. Modell der Interaktion mit dem Benutzer

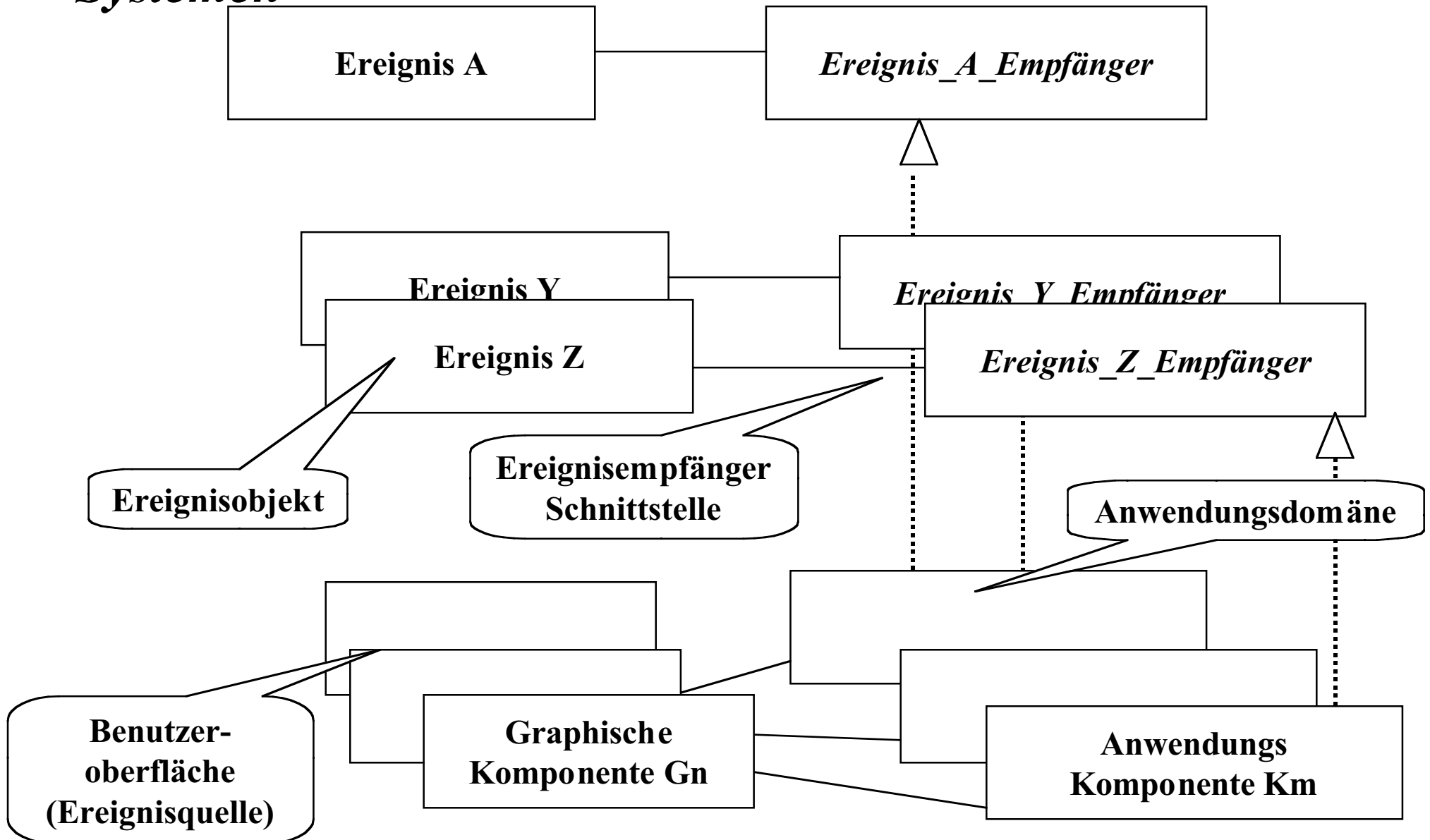
Modell der Interaktion mit dem Benutzer

- ❖ **Definition Bedienoberfläche:** Die Menge aller Komponenten eines interaktiven Systems, die der Interaktion mit Benutzern dienen.
 - ◆ **Graphische Bedienoberfläche** (Graphical User interface, GUI): Die Interaktion erfolgt mit Hilfe von graphischen Komponenten (Knopf, Auswahl-Menü, Tabelle, ...)
 - ◆ **Textuelle Bedienoberfläche** (Command language interface): Die Interaktion erfolgt mit Hilfe von textuellen Komponenten (textuelle Eingabe und Ausgabe).

Ereignis-basierte Interaktive Systeme

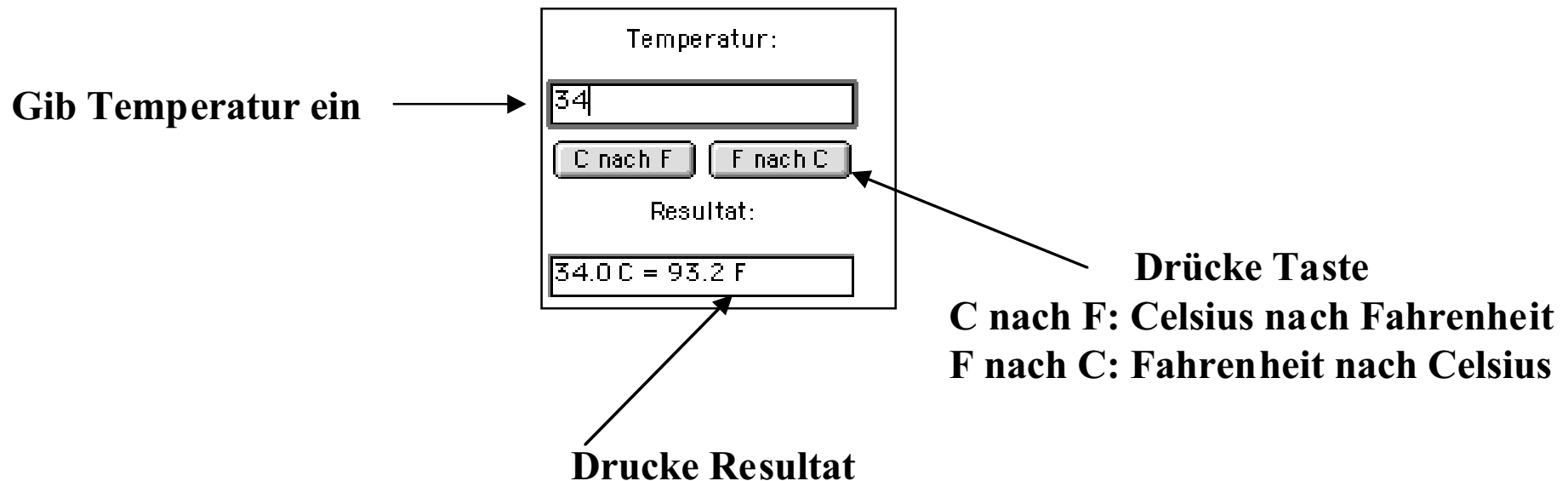
- ❖ Interaktive Systeme werden oft als Ereignis-basierte Systeme modelliert.
- ❖ Bei der Modellierung von Ereignis-basierten interaktiven Systemen brauchen wir folgende vier Modelle:
 - ◆ Modell der Anwendungsdomäne
 - ◆ Modell der Interaktion mit dem Benutzer
 - ◆ Struktur der Bedienoberfläche
 - ◆ Ereignisquellen
 - ◆ Modell der Ereignisse
 - ◆ Modell der Ereignisempfänger

Allgemeine Struktur von Ereignis-basierten Interaktiven Systemen



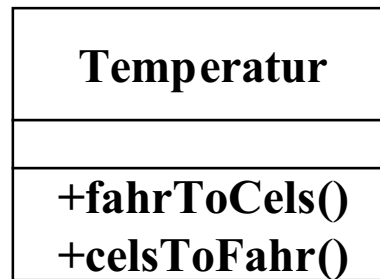
Beispiel der Entwicklung eines Ereignis-basierten interaktiven Systems: Temperatur-Konvertierer

- ❖ *Problembeschreibung*: Ein System soll die Konvertierung von Celsius nach Fahrenheit und von Fahrenheit nach Celsius berechnen.
- ❖ Das System soll die folgende graphische Bedienoberfläche haben.



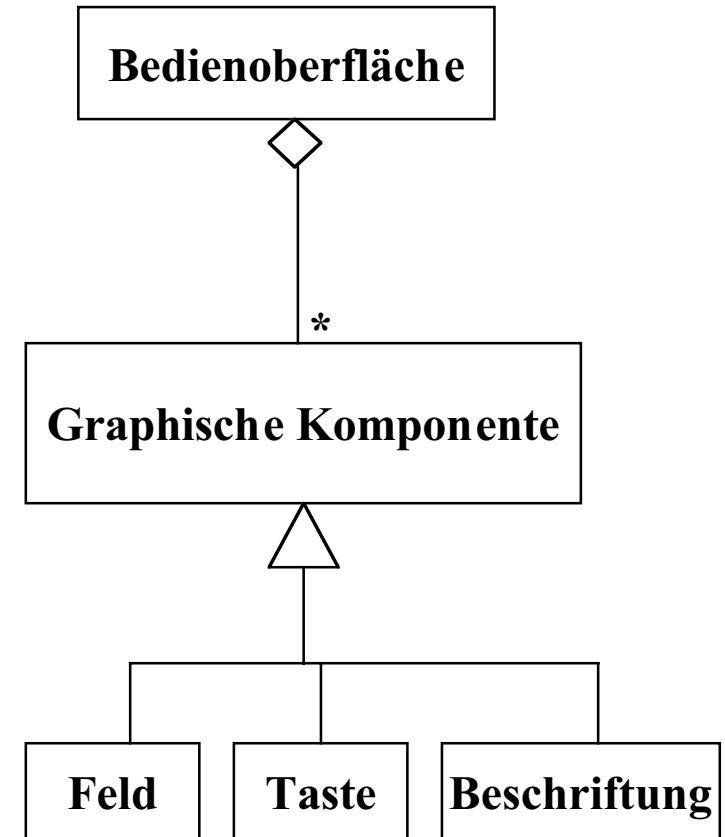
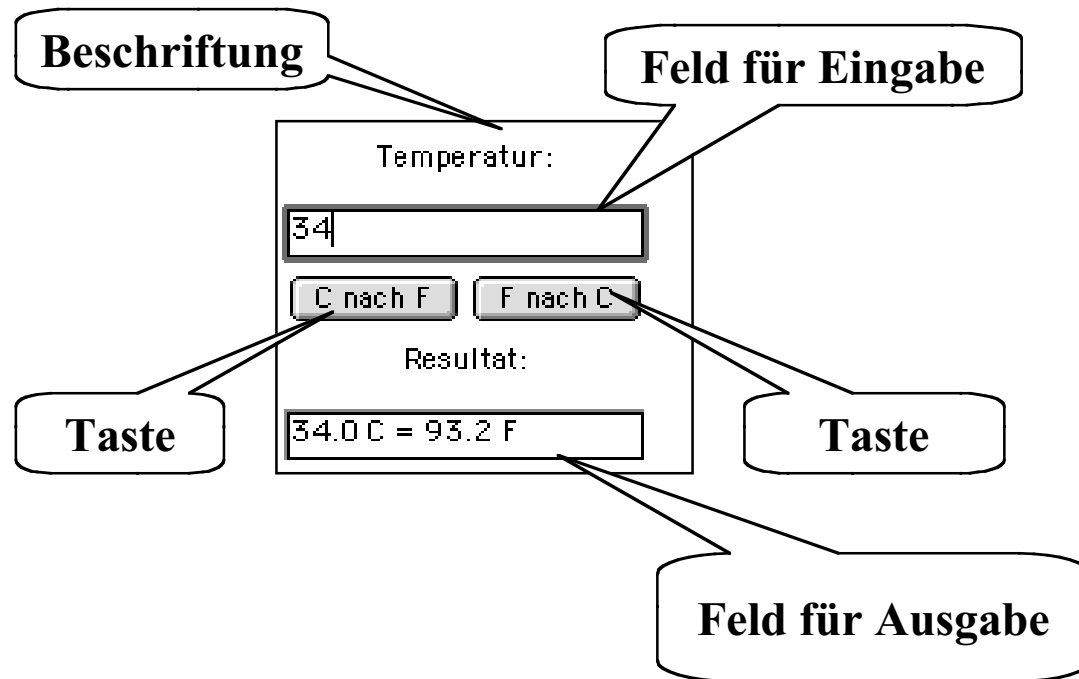
- ❖ Wir lösen das Problem mit einem Ereignis-basierten interaktiven System.

Modellierung der Anwendungsdomäne



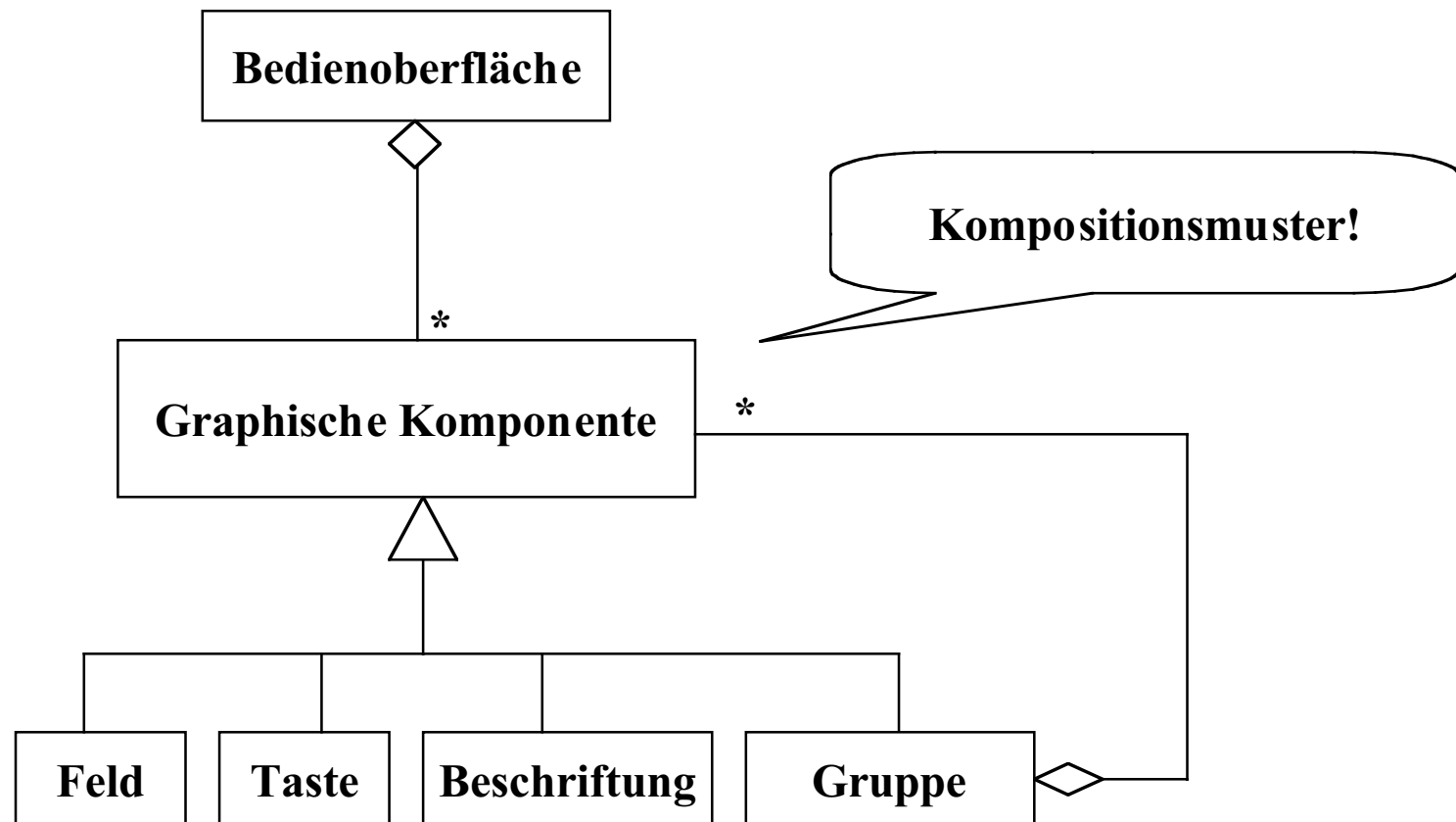
Modellierung der Interaktion mit dem Benutzer

Bedienoberfläche des
Temperaturkonverters

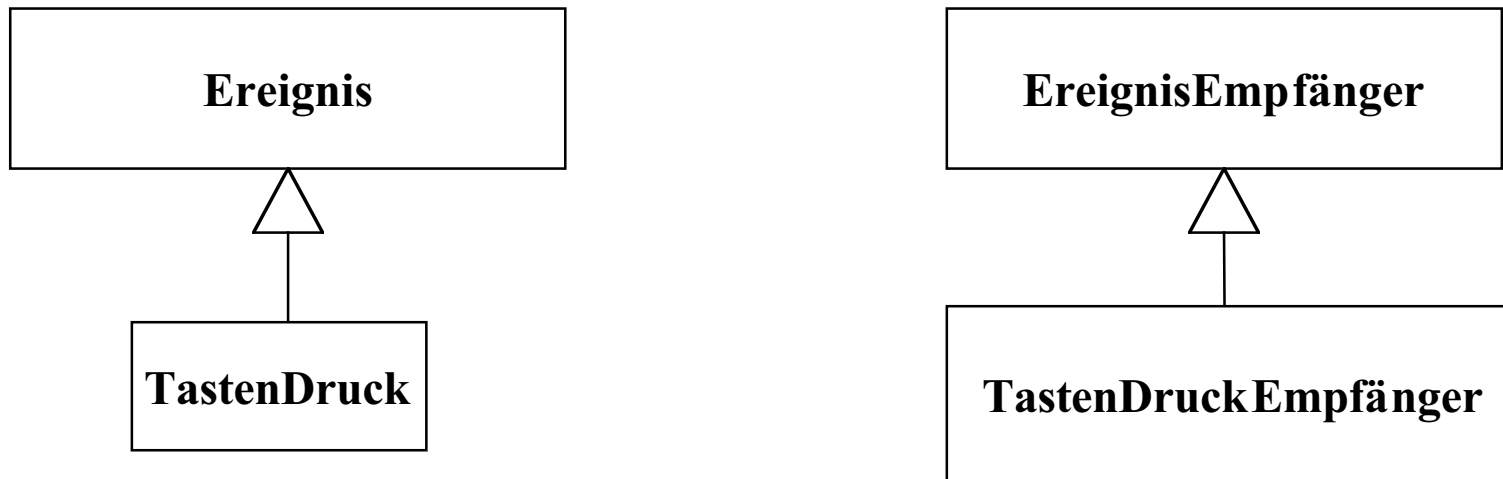


Modellierung von Bedienoberflächen (Verallgemeinerung)

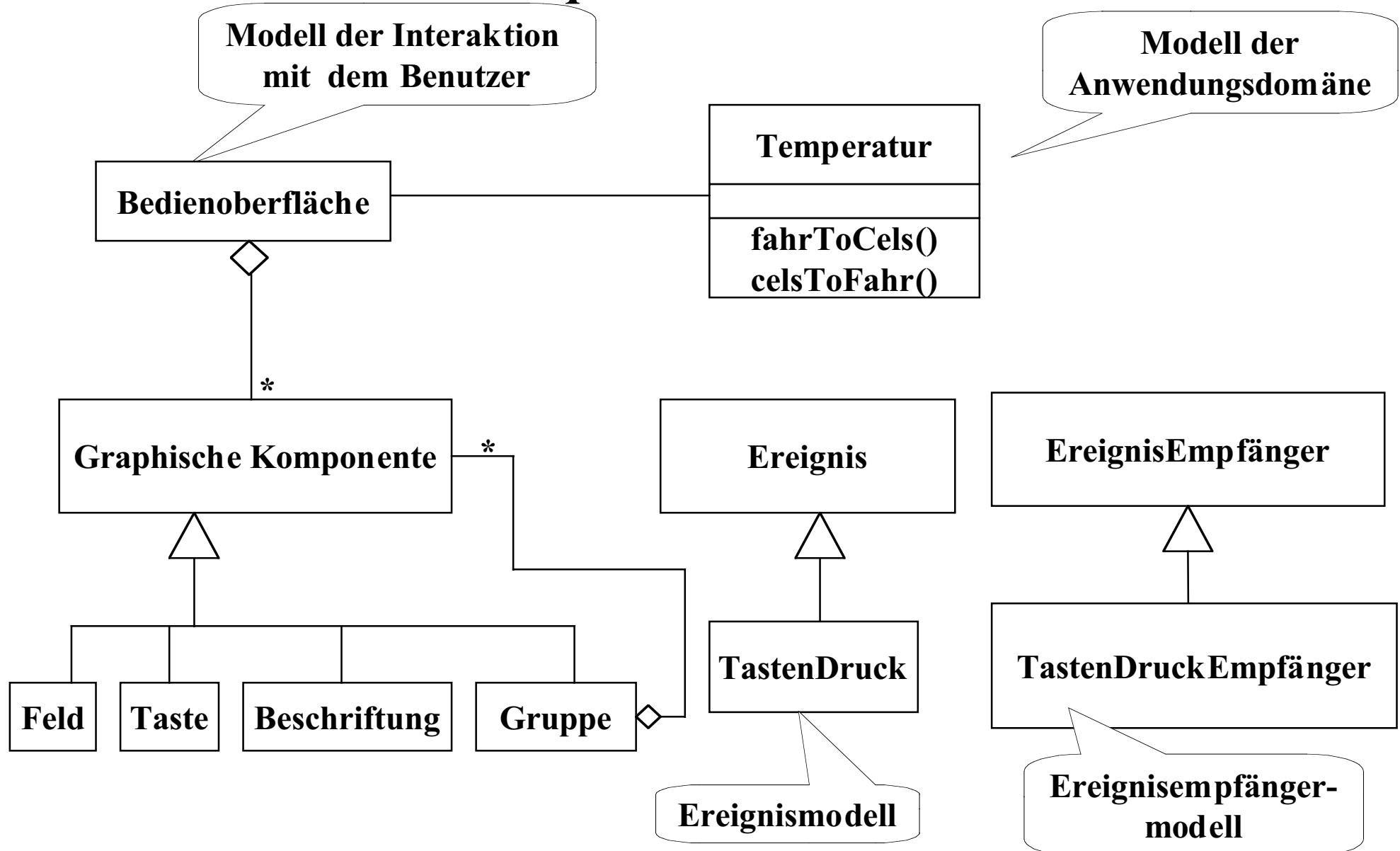
- ❖ Wir wollen graphische Komponenten auch gruppieren können.
- ❖ Dann erhalten wir:



Modellierung der Ereignisse und Ereignisempfänger



Gesamt-Modell des Temperatur-Konvertierers



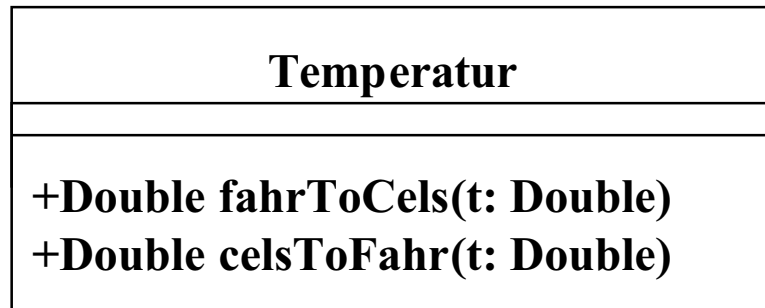
Implementierung des Temperatur-Konvertierers in Java

- ❖ Die Implementierung der Anwendungsdomäne ist trivial (nächste Folie).
- ❖ Für die Implementierung von Bedienoberfläche, Ereignisquellen, Ereignissen und Ereignisempfängern stellt Java interessante Konzepte bereit.
 - ◆ Schauen wir uns deshalb erst einmal an, welche Konzepte aus Java's Klassenbibliotheken wir verwenden können.
- ❖ Für die Modellierung von graphischen Komponenten:
 - ◆ **java.awt** (Abstract Windowing Toolkit) und **javax.swing** enthalten *plattformunabhängige* Komponenten für graphische Bedienoberflächen.
- ❖ Für die Modellierung von Ereignissen und Ereignisempfängern:
 - ◆ **java.util** enthält Basisklassen/-schnittstellen für Java's Ereignismodell

Implementierung der Anwendungsdomäne

Spezifikation:

UML-Modell:



OCL-Modell:

Temperatur::fahrToCels(t:Double)

post: result $5 * (t - 32) / 9$

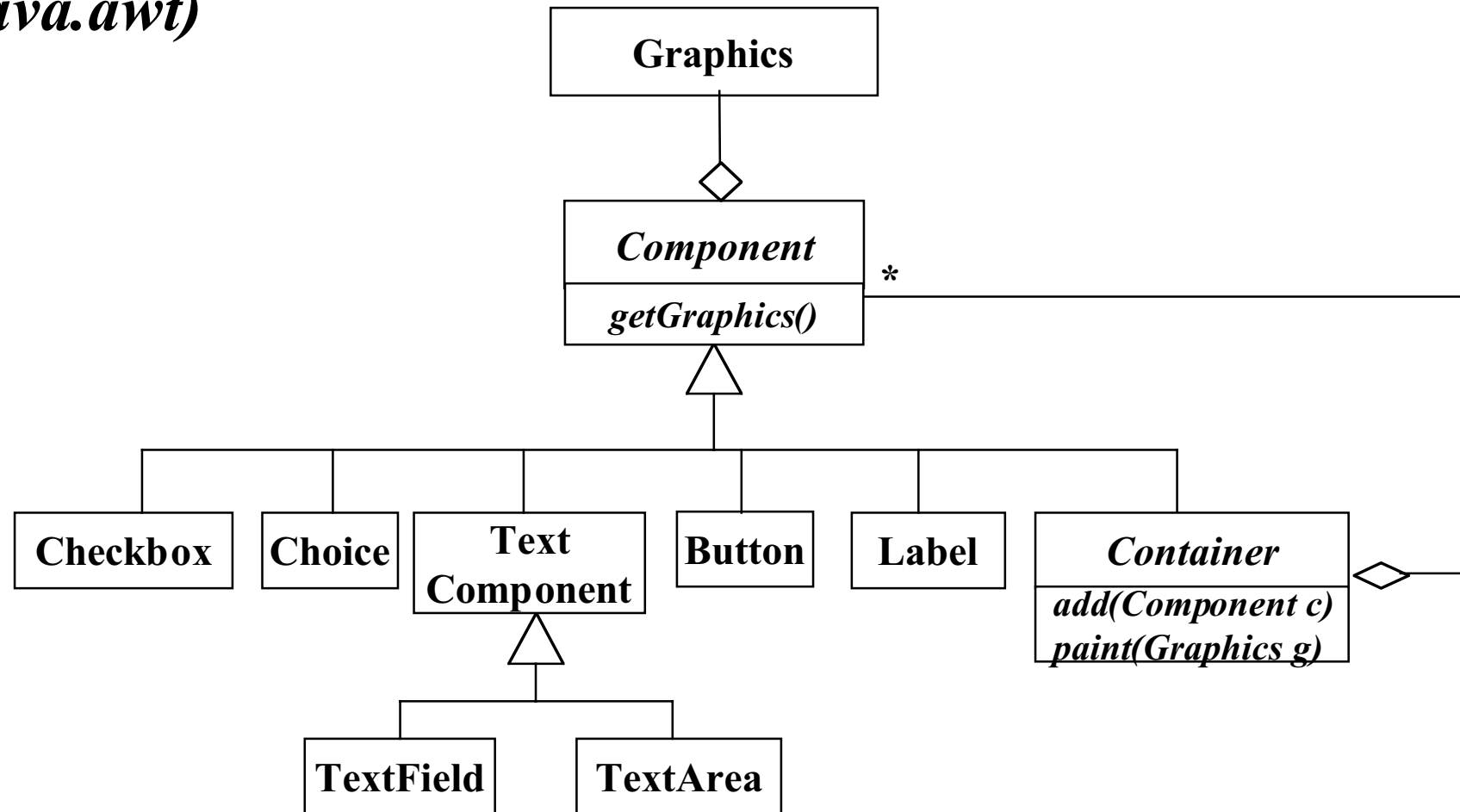
Temperatur::CelsToFahr(t:Double)

post: result $9 * t / 5 + 32$

Implementierung:

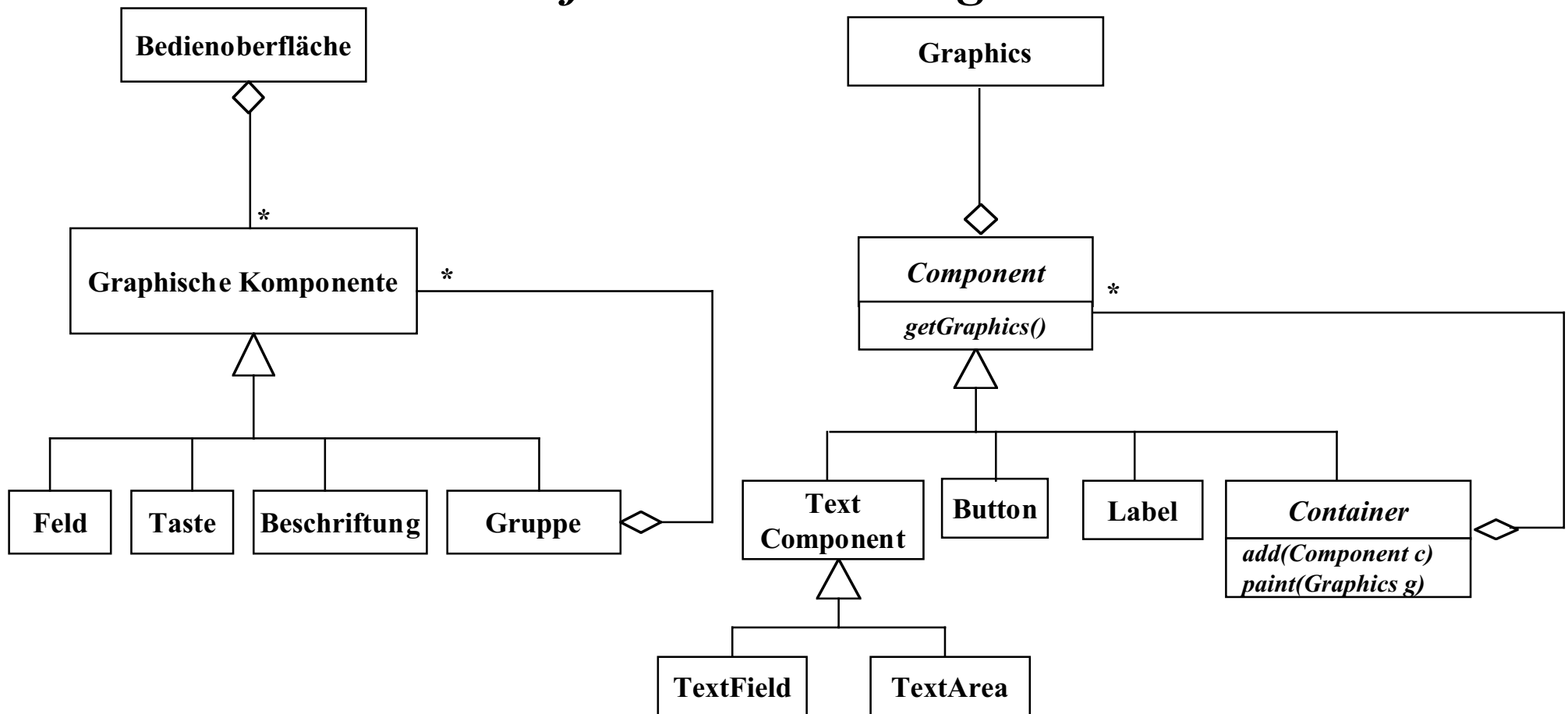
```
public class Temperatur {  
    public Temperatur () {}  
    public double fahrToCels (double t) {  
        return (5.0 * (t- 32.0) / 9.0);  
    }  
    public double celsToFahr (double t) {  
        return (9.0 * t / 5.0 + 32.0);  
    }  
}
```

Implementierung der graphischen Komponenten in Java (*java.awt*)



Wenn wir dieses Klassendiagramm mit dem Modell für den Temperatur-Konvertierer vergleichen, dann sehen wir, dass das Modell der Bedienoberfläche relativ einfach auf bereits in Java existierende Lösungsklassen abgebildet werden kann.

Modell der Bedienoberfläche vs. Lösungsklassen in Java

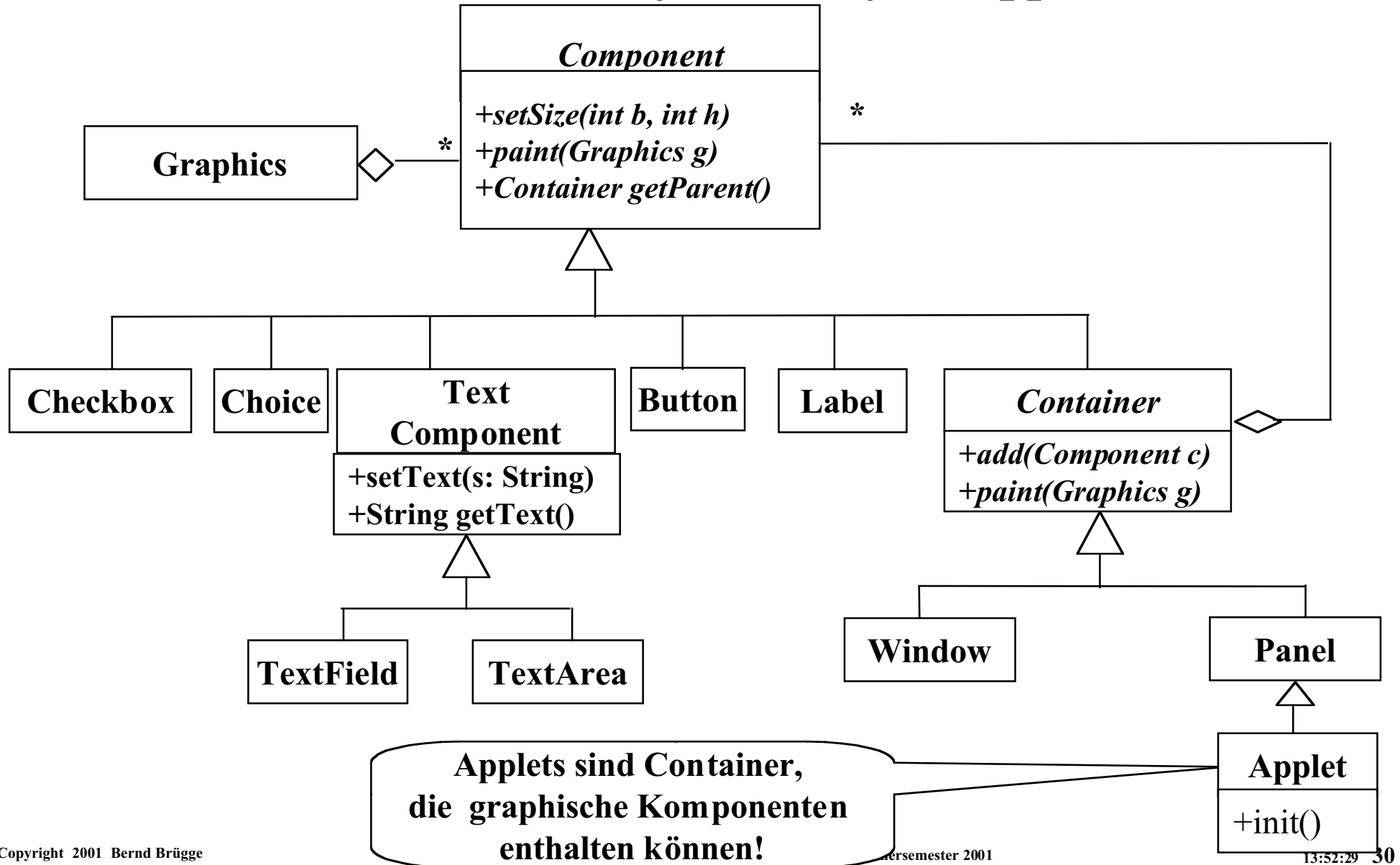


- Die Implementierung ist also relativ problemlos, da Java passende Klassen bereitstellt.
- Die Bedienoberfläche werden wir als Applet implementieren

Java Applets

- ❖ **Definition Applet:** Ein Applet ist eine graphische Java-Komponente, die von einem Web-Browser oder einem Applet Viewer ausgeführt werden kann.
- ❖ Die Klasse **Applet** ist eine Unterklasse von **Container**.
- ❖ Ein Applet muss keine **main ()**-Methode besitzen, um ausgeführt werden zu können.

Die Stellung von Applets in Java's Klassenbibliothek (Illustrativer Ausschnitt aus java.awt, java.applet)



Wichtige Methoden bei der Implementierung von Applets

❖ **java.awt.Component:**

- ◆ **setSize(int b, int h)** Größe der Komponente (Breite **b**, Höhe **h**)
- ◆ **paint(Graphics g)** Zeichnet die Komponente
- ◆ **Container getParent()** Liefert den Container, von dem die Komponente ein Teil ist.

❖ **java.awt.Container:**

- ◆ **add(Component c)** Fügt die graphische Komponente **c** zum Container hinzu.
- ◆ **paint(Graphics g)** Zeichnet alle Komponenten vom Container (überschreibt **paint()** von **Component**)

❖ **java.awt.TextComponent:**

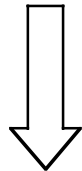
- ◆ **setText(string s)** Setzt Zeichenkette in der Komponente auf **s**
- ◆ **String getText()** Liefert Zeichenkette der Komponente.

❖ **java.applet.Applet:**

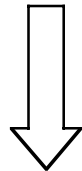
- ◆ **init()** Wird einmal aufgerufen, wenn die Exekution des Applets gestartet wird.

Die Stellung von Applets in Java's Vererbungshierarchie

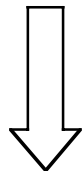
```
public class Component extends Object { ... }
```



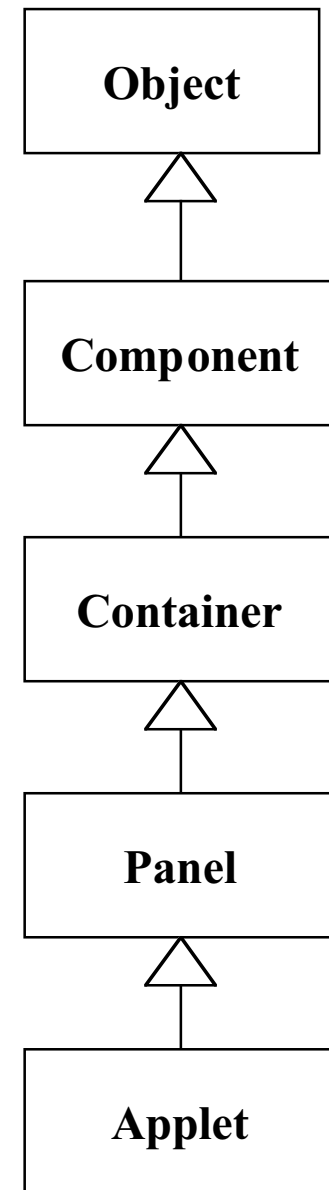
```
public class Container extends Component { ... }
```



```
public class Panel extends Container { ... }
```



```
public class Applet extends Panel { ... }
```



Instantiierung von Applets

- ❖ Applets werden im allgemeinen nicht explizit im Programmtext instantiiert.
 - ◆ Applets werden erst von einem Web-Browser oder Applet Viewer instantiiert.
- ❖ Viele Applet-Methoden haben zwar Standard-Implementierungen, sind aber mit der Absicht entworfen worden, dass sie vom Implementierer überschrieben werden.
- ❖ Üblicherweise überschreiben wir zumindest die **init()**-Methode, und zwar mit der Deklaration und Initialisierung der graphischen Komponenten der Bedienoberfläche für das interaktive System.

Auszug aus der Java Applet-Spezifikation

```
public class Applet extends Panel {  
    public void init();  
    public void destroy();  
    public AudioClip getAudioClip(URL url);  
    public void play(URL url, String name);  
    public void getImage(URL url);  
    ...  
}
```

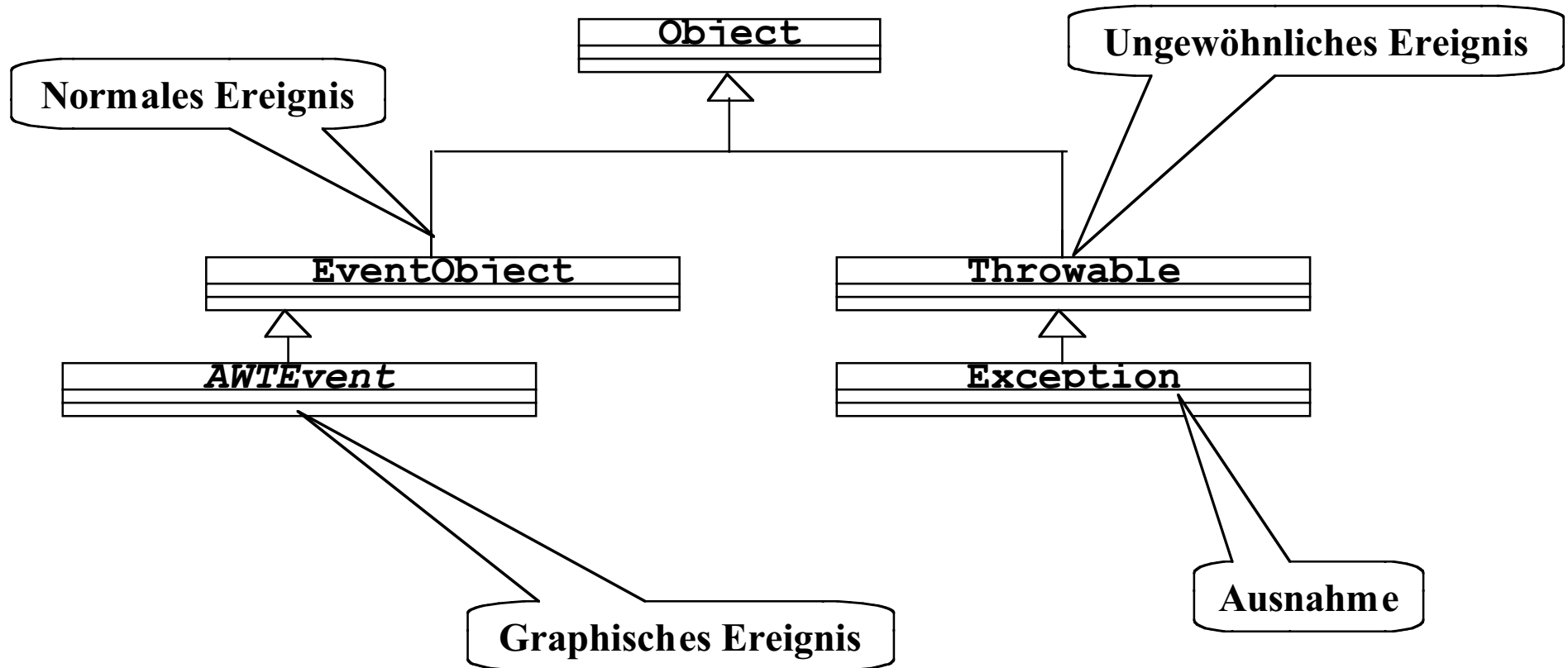
- ❖ Die Klasse **Applet** wird normalerweise nicht direkt instantiiert:
 - ◆ Wir müssen immer eine Unterklasse von **Applet** deklarieren.
- ❖ Typische Deklaration eines Applets:

```
public class MyApplet extends Applet
```

Wo stehen wir?

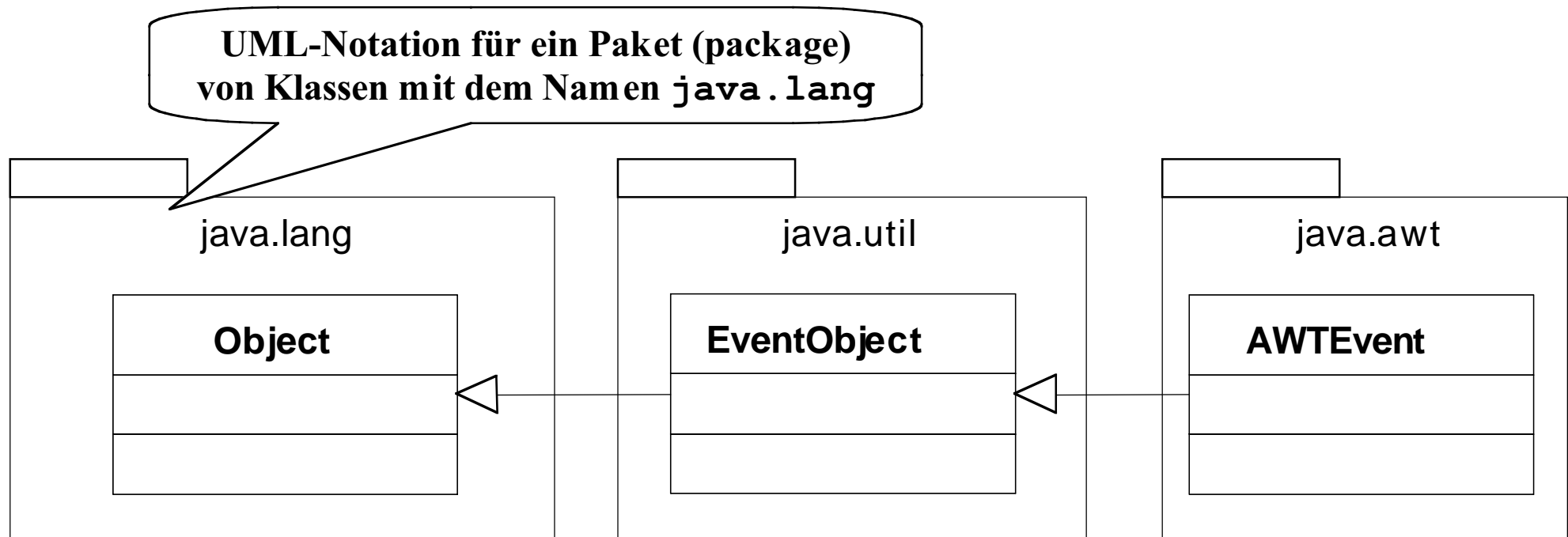
- ❖ Wir implementieren ein Modell eines Ereignis-basierten interaktiven Systems für einen Temperatur-Konvertierer
- ❖ Bei der Modellierung von Ereignis-basierten interaktiven Systemen brauchen wir folgende vier Modelle:
 - ◆ Modell der Anwendungsdomäne
 - ◆ Modell der Interaktion mit dem Benutzer
 - ☞ Modell der Ereignisse
 - ◆ Modell der Ereignisempfänger
- ❖ Wir implementieren dann jedes dieser Modelle in Java:
 - ◆ Implementierung der Anwendungsdomäne
 - ◆ Implementierung der Bedienoberfläche
 - ◆ Implementierung des Ereignismodells
 - ◆ Implementierung der Ereignisempfänger

Implementierung des Ereignismodells in Java

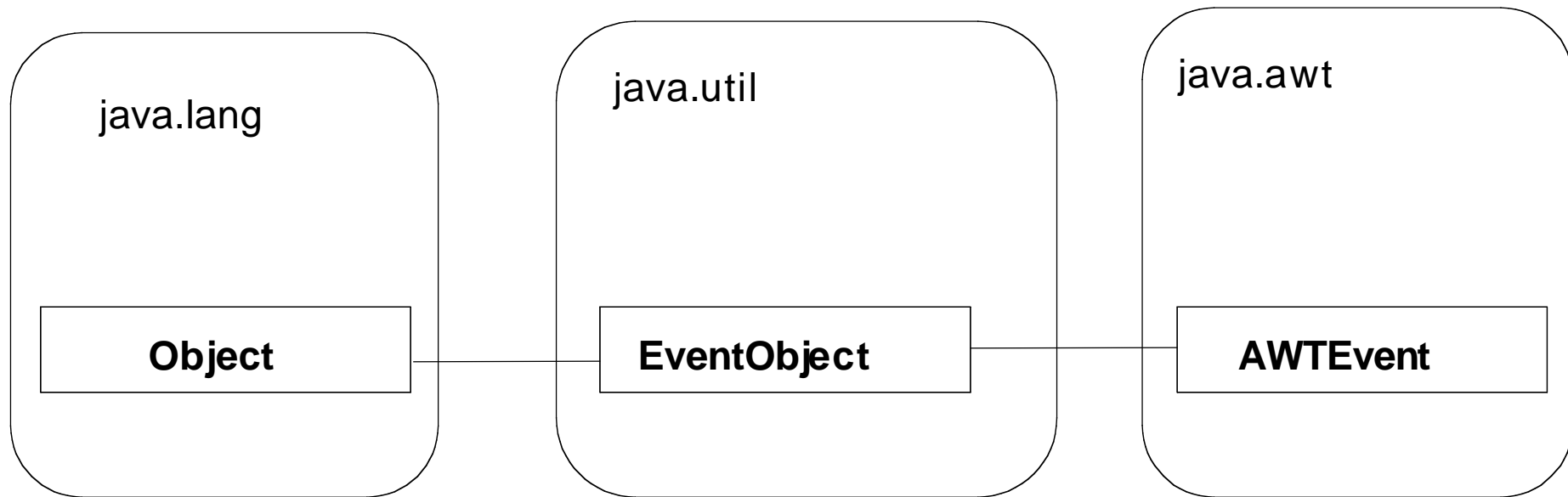


Java's Ereignis-Hierarchie

- ❖ Alle normalen Ereignisse in Java sind Unterklassen der Klasse **EventObject**.



Java's Ereignis-Hierarchie in einer anderen Notation



- ❖ Viele Java-Einführungsbücher und Referenzhandbücher benutzen diese Notation ("*Java, Java, Java*", "*Java in a Nutshell*", usw.)
 - ◆ Diese Notation ist nicht UML-basiert. Bitte nicht benutzen, wenn Sie UML-Klassendiagramme *erstellen*.
- ❖ Wir erwähnen sie hier, um das *Lesen* von Klassendiagrammen in der aktuellen Java-Literatur zu erleichtern.
 - ◆ Viele Autoren stellen zur Zeit auf UML um.

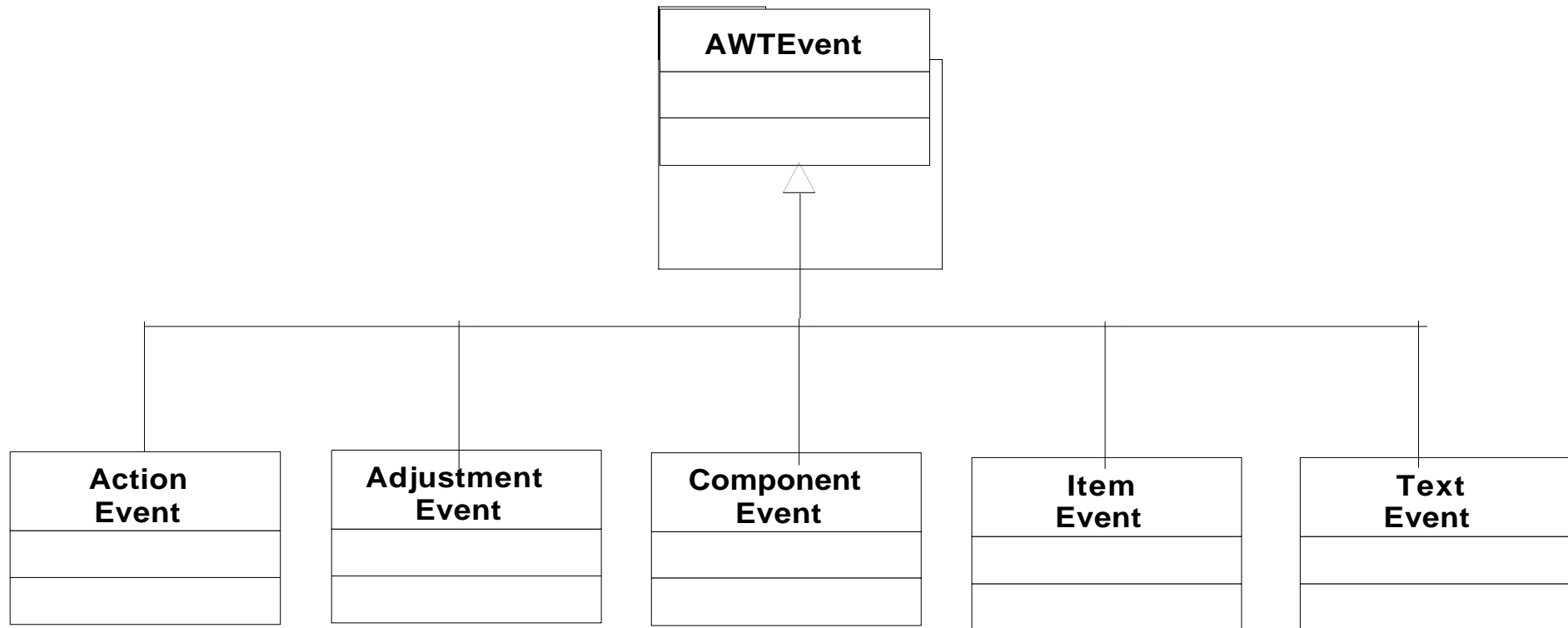
Die Klasse EventObject

- ❖ EventObject ist die Basisklasse für alle Standard-Ereignisse in Java.

```
public class EventObject {  
    public EventObject(Object source) ;  
    public Object getSource() ;  
    public String toString() ;  
}
```

- ❖ Für ein gegebenes Ereignis **e** bekommt man mit dem Aufruf **e.getSource()** eine Referenz auf die Ereignisquelle und mit **e.toString()** eine textuelle Beschreibung des Ereignisses.
- ❖ Uns interessiert die Unterklasse "*Ereignisse, die von graphischen Komponenten erzeugt werden*", d.h. Ereignisse vom Typ **AWTEvent**.

Basis-Ereignisse graphischer Komponenten in Java



ActionEvent: Benutzer hat irgendetwas geklickt (Komponente **Button**, **TextField**, ..)

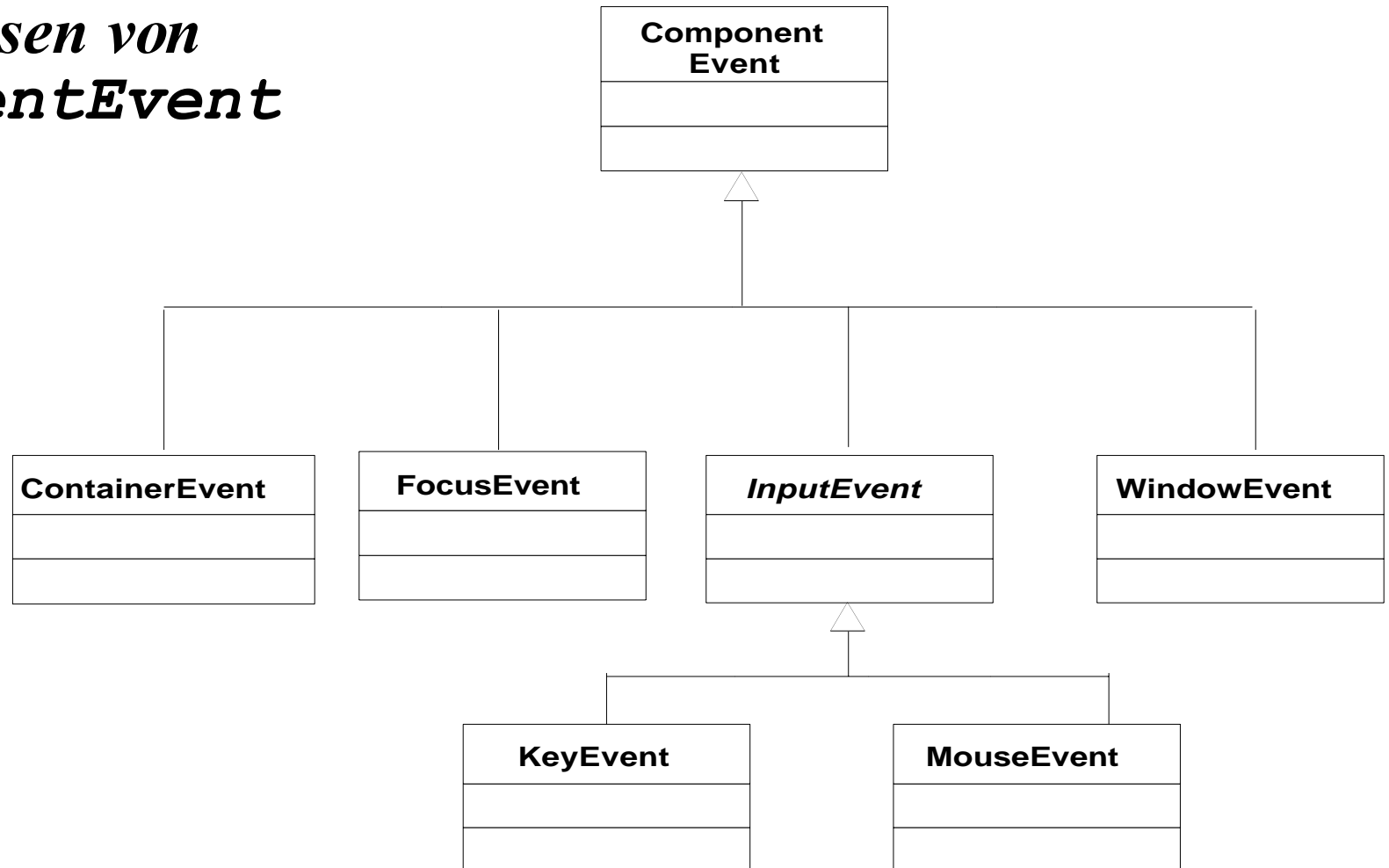
AdjustmentEvent: Benutzer hat in einem Fenster den Inhalt verschoben

ComponentEvent: Eine Komponente wurde bewegt, verkleinert oder vergrößert

ItemEvent: Ein Benutzer hat eine Wahl getroffen (Komponente **Choice**)

TextEvent: Benutzer hat Text editiert

Unterklassen von ComponentEvent



ContainerEvent: Benutzer hat eine Komponente zu einem Container addiert/entfernt

FocusEvent: Benutzer hat eine Komponente selektiert

KeyEvent: Benutzer hat eine Taste gedrückt

MouseEvent: Benutzer hat eine Maustaste gedrückt

WindowEvent: Benutzer hat ein Fenster manipuliert

Graphische Komponenten und dazugehörige Ereignisse

Graphische Komponente	Ereignis	Beschreibung
Button	ActionEvent	Benutzer hat Knopf gedrückt
Component	ComponentEvent	Eine Komponente wurde bewegt, verkleinert oder vergrößert
	FocusEvent	Komponente kam in Fokus
	KeyEvent	Benutzer drückte eine Taste
	MouseEvent	Benutzer benutzte die Maus
Container	ContainerEvent	Eine Komponente wurde bei einem Container hinzugefügt oder gelöscht
TextComponent	TextEvent	Benutzer hat Text editiert
Textfield	ActionEvent	Benutzer hat "Enter" Taste gedrückt
Window	WindowEvent	Benutzer hat Fenster manipuliert

Das graphische Ereignis "Maus ist geklickt" in Java's Vererbungshierarchie

```
public class Object { ... }
```

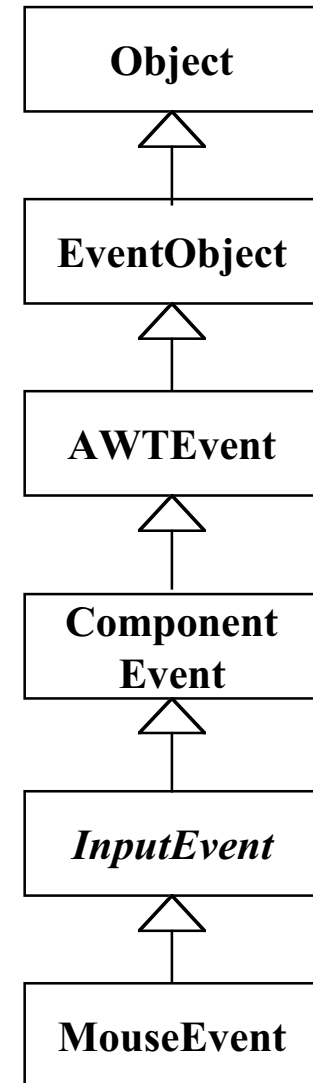
```
public class EventObject extends Object { ... }
```

```
public class AWTEvent extends EventObject { ... }
```

```
public class ComponentEvent extends AWTEvent { ... }
```

```
public class InputEvent extends ComponentEvent { ... }
```

```
public class MouseEvent extends InputEvent { ... }
```



Komponenten können auch verschiedene Ereignisse erzeugen

- ❖ Ein **Button** ist die Ereignisquelle für **ActionEvent** und **MouseEvent**
- ❖ Ein **TextField** generiert die Ereignisse **ActionEvent**, **KeyEvent** und **TextEvent**, und zwar immer, wenn ein Benutzer einen Buchstaben innerhalb eines Textfeldes tippt.

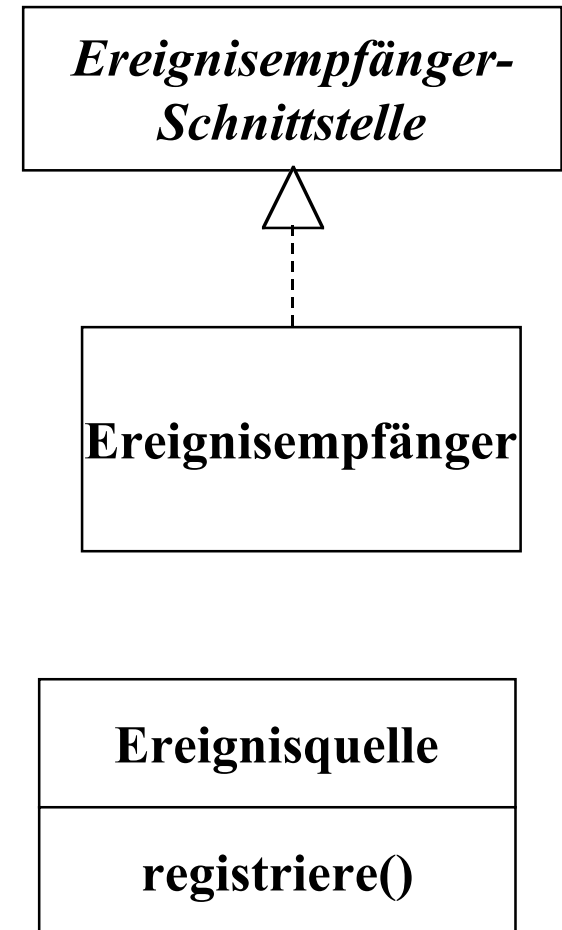
Ereignisempfänger in Java

- ❖ Wir haben jetzt die Ereignis-Hierarchie in Java kennengelernt
 - ◆ Wir kennen insbesondere einige Ereignisse, die von graphischen Komponenten erzeugt werden können.
- ❖ Wenn ein Ereignis erzeugt wird, dann möchten wir, dass das Ereignis von jemanden verarbeitet wird (*"Sonst hört keiner den fallenden Baum."*).
- ❖ Die Verbindung zwischen Ereignis und Ereignisverarbeitung hatten wir mit Hilfe des **Ereignisempfänger**-Konzeptes modelliert (siehe Folie 9).
 - ◆ Java unterstützt die Implementierung von Ereignisempfängern.

Zusammenspiel von Ereignisquelle und -empfänger

- ❖ Ein *Ereignisempfänger* (event listener) ist ein beliebiges Objekt, das eine *Ereignisempfänger-Schnittstelle* (listener interface) implementiert.
- ❖ Ein Ereignisempfänger kann nur Ereignisse von einem bestimmten Typ empfangen.

- ❖ Welche Ereignisse das sind, wird von der *Ereignisquelle* bestimmt, die die Ereignisempfänger für Ereignisse registriert.



Zusammenspiel von Ereignisquelle und -empfänger in Java

❖ In Java werden Ereignisempfänger nach folgenden Konventionen benannt:

- ◆ Der Ereignistyp, für den sich ein Empfänger registriert, wird nicht als Parameter übergeben, sondern ist Teil des Namens der Ereignisempfänger-Schnittstelle:

- ◆ Beispiel:

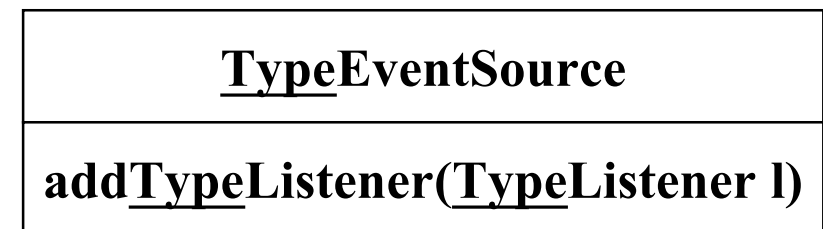
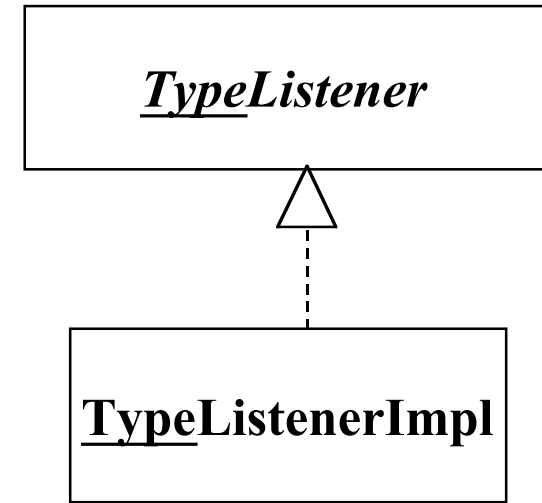
ActionListener für **ActionEvent**

❖ **Allgemeines Schema für die Registrierung:**
Der Ereignisempfänger (**TypeListener**) für **TypeEvent** wird durch **addTypeListener()** registriert.

- ◆ Beispiel:

addActionListener(this)

bindet **ActionEvent** an den Ereignisempfänger, der von **this** implementiert ist



Komponenten, Ereignisse, Ereignisempfänger-Schnittstellen

Graphische Komponente	Ereignis	Assoziierte Ereignisempfänger-Schnittstelle
Button	ActionEvent	ActionListener
Component	ComponentEvent	ComponentListener
	FocusEvent	FocusListener
	KeyEvent	KeyListener
	MouseEvent	MouseListener
Container	ContainerEvent	ContainerListener
TextComponent	TextEvent	TextListener
Textfield	ActionEvent	ActionListener
Window	WindowEvent	WindowListener

Ereignis, Ereignisempfänger-Schnittstelle und Ereignisempfänger

- ❖ Für jedes Ereignis **TypeEvent** in der Ereignishierarchie von Java gibt es eine *assoziierte Ereignisempfänger-Schnittstelle* mit dem Namen **TypeListener**.
- ❖ Beispiel:
 - ◆ Die graphische Komponente **Button** erzeugt Ereignisse vom Typ **ActionEvent**. Die assoziierte Ereignisempfänger-Schnittstelle heißt **ActionListener**.
- ❖ **Definition:** Ein **Ereignisempfänger** für ein Ereignis ist ein *beliebiges Java-Objekt*, welches die mit dem Ereignis assoziierte Ereignisempfänger-Schnittstelle implementiert.

Ereignisempfänger

- ❖ Eine Ereignisempfänger-Schnittstelle ist eine Java-Schnittstelle
- ❖ Ein Java-Objekt, das Ereignisempfänger sein will, muss diese Schnittstelle implementieren.
- ❖ **Beispiel:**
 - ◆ Die graphische Komponente **Button** erzeugt Ereignisse vom Typ **ActionEvent**.
 - ◆ Die assoziierte Ereignisempfänger-Schnittstelle heißt **ActionListener**.
 - ◆ Wir wollen ein Applet schreiben, das diese Ereignisse verarbeitet.
- ❖ (aus dem Java-Referenzmanual) **java.awt.event.ActionListener:**

```
public abstract interface ActionListener extends EventListener {  
    public abstract void actionPerformed(ActionEvent e);  
}
```
- ❖ Um **ActionListener** zu implementieren, müssen wir also die abstrakte Methode **actionPerformed** implementieren.

Beispiel-Implementierung von actionPerformed

Signatur:

```
public abstract void actionPerformed(ActionEvent e);
```

Implementierung:

```
public void actionPerformed(ActionEvent e) {  
    if (e.getSource() == cfButton) {  
        System.out.println(e.toString());  
    }  
} // actionPerformed
```

Ausgabe von **System.out.println(e.toString())**
(in der Textkonsole für die Ausgabe von Java-Texten):

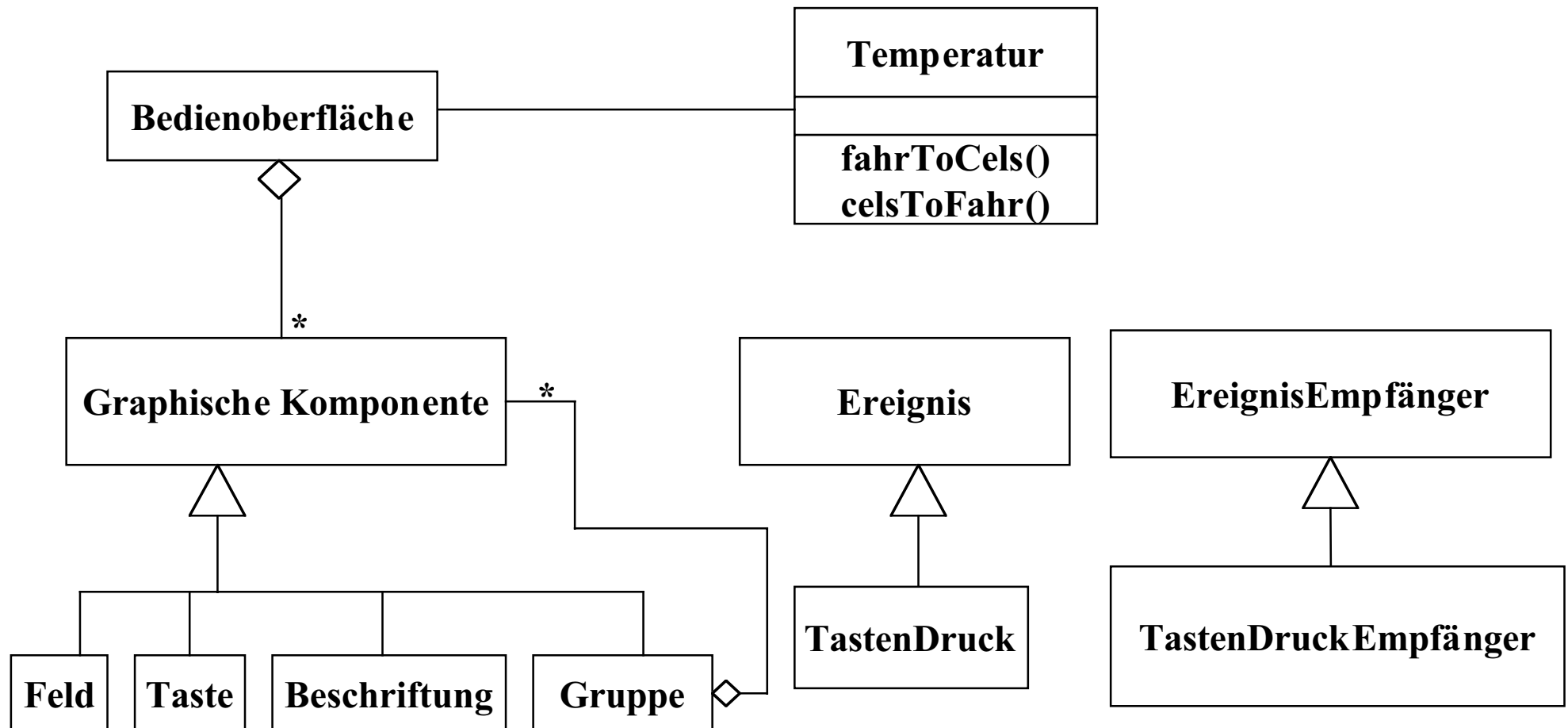
```
java.awt.event.ActionEvent[ACTION_PERFORMED,cmd=C nach F] on button0
```

Deklaration von Applets mit Ereignisempfängern

- ❖ Bei der Applet-Deklaration müssen alle Ereignisempfänger-Schnittstellen deklariert werden, die das Applet implementiert.
- ❖ Beispiel:
 - ◆ Ein Applet soll Ereignisempfänger für **ActionEvent** sein.
 - ◆ Dann muss die Deklaration des Applets sowohl von der Klasse **Applet** also auch von der assoziierten Ereignisempfänger-Schnittstelle **ActionListener** erben:

```
public class MyApplet extends Applet  
implements ActionListener
```

Wo stehen wir? Modell des Temperatur-Konvertierers



Was kommt jetzt?

- ❖ Bei der Modellierung von Ereignis-basierten interaktiven Systemen brauchen wir folgende vier Modelle:
 - ◆ Modell der Anwendungsdomäne
 - ◆ Modell der Interaktion mit dem Benutzer
 - ◆ Modell der Ereignisse
 - ◆ Modell der Ereignisempfänger

- ➡ Wir implementieren dann diese Modelle in Java:
 - ◆ Implementierung der Anwendungsdomäne
 - ◆ Implementierung der Bedienoberfläche
 - ◆ Implementierung des Ereignismodells
 - ◆ Implementierung der Ereignisempfänger

Implementierung des Temperatur-Konverters als Ereignis-basiertes interaktives System

- ❖ Konzeptionell besteht die Implementierung von Ereignis-basierten interaktiven Systemen aus vier Schritten:
 - 1. *Instantiierung:*** Hier werden alle Klassen aus der Anwendungsdomäne und der Bedienoberfläche deklariert und instantiiert.
 - 2. *Zusammenstellung des Containers:*** Hier werden alle graphischen Komponenten der Bedienoberfläche in den Container aufgenommen.
 - 3. *Assoziation von Ereignissen und Ereignisempfängern:*** Hier werden bestimmte Ereignisse der Ereignisquellen (graphischen Komponenten) dynamisch an Ereignisempfänger gebunden.
 - 4. *Implementierung aller Ereignisempfänger-Schnittstellen:*** Ereignisempfänger-Schnittstellen sind Java-Schnittstellen und müssen daher implementiert werden.

Beispiel für die Implementierung von Ereignis-basierten interaktiven Systemen mit Applets

```
public class TemperaturApplet extends Applet implements
    ActionListener {

    // 1. Instantiierung aller Komponenten, sowohl
    //    Anwendungsdomäne als auch der Bedienoberfläche
    public void init() {
        // 2. Zusammenstellung des Containers
        // 3. Assoziation von Ereignissen und Ereignisempfängern
    } // init()

    // 4. Implementierung aller
    //    Ereignisempfänger-Schnittstellen
} // TemperaturApplet
```

Implementierung des Temperatur-Konverters:

1. Instantiierung der Komponenten

```
public class TemperaturApplet extends Applet implements
    ActionListener {
    private TextField eingabeFeld = new TextField(15);
    private TextField ausgabeFeld = new TextField(15);
    private Label schrift1 = new Label("Temperatur:");
    private Label schrift2 = new Label("Resultat:");
    private Button cfButton = new Button("C nach F");
    private Button fcButton = new Button("F nach C");
    private Temperature temperatur = new Temperature();
    public void init () {
        // Zusammenstellung des Containers
        // Assoziation von Ereignissen und Ereignisempfängern
    } // init
    // Implementierung der Ereignisempfänger-Schnittstelle
} // TemperaturApplet
```

**Deklaration von
Objekten für die
Benutzerschnittstelle**

**Deklaration "von Objekten" für
die Anwendungsdomäne**

Implementierung des Temperatur-Konverters:

2. Zusammenstellung des Containers

```
public class TemperaturApplet extends Applet implements ActionListener {
```

```
// Instantiierung der Komponenten
```

```
public void init () {
```

```
    add(schrift1);      // Addiere Komponente vom Typ Label
```

```
    add(eingabeFeld);  // Addiere Komponente vom Typ TextField
```

```
    add(cfButton);     // Addiere Komponente vom Typ Button
```

```
    add(fcButton);    // Addiere Komponente vom Typ Button
```

```
    add(schrift2);     // Addiere Komponente vom Type Label
```

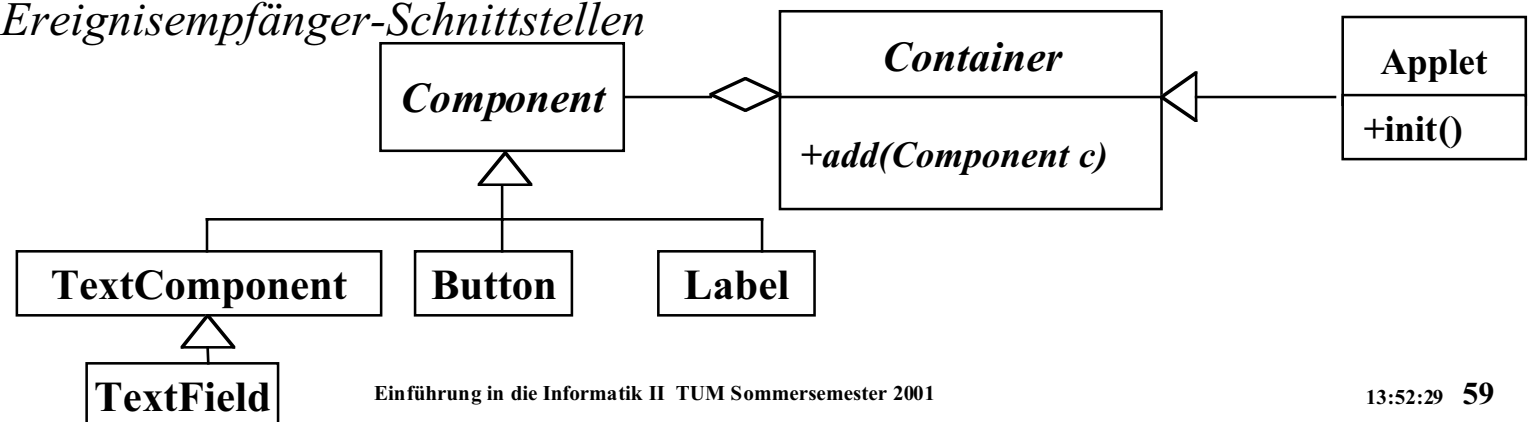
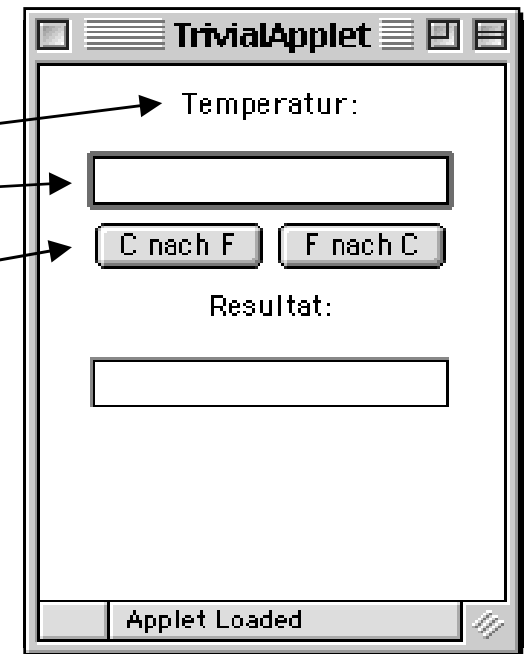
```
    add(ausgabeFeld); // Addiere Komponente vom Typ TextField
```

```
// Assoziation von Ereignissen und Ereignisempfängern
```

```
} // init ()
```

```
// Implementierung der Ereignisempfänger-Schnittstellen
```

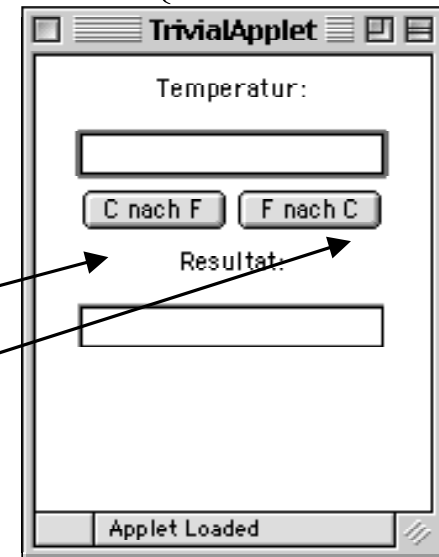
```
} // TemperaturApplet
```



Implementierung des Temperatur-Konverters:

3. Assoziation von Ereignisquellen und Ereignisempfängern

```
public class TemperaturApplet extends Applet implements ActionListener {  
    // Instantiierung der Komponenten  
    public void init () {  
        add(schrift1);        // Addiere Komponente vom Typ Label  
        add(eingabeFeld);     // Addiere Komponente vom Typ TextField  
        add(cfButton);        // Addiere Komponente vom Typ Button  
        add(fcButton);        // Addiere Komponente vom Typ Button  
        add(schrift2);        // Addiere Komponente vom Typ Label  
        add(ausgabeFeld);     // Addiere Komponente vom Typ TextField  
  
        cfButton.addActionListener(this); // Verbinde Ereignisquelle cfButton mit einem  
                                         // Ereignisempfänger, in diesem Fall dieses Applet  
        fcButton.addActionListener(this); // Verbinde Ereignisquelle fcButton mit einem  
                                         // Ereignisempfänger, in diesem Fall dieses Applet  
  
        setSize(160,200); // Setze die Größe des Applets auf 160x200 Pixel  
    } // init ()  
    // Implementierung der Ereignisempfänger-Schnittstellen  
} // TemperaturApplet
```



Implementierung des Temperatur-Konverters:

4. Implementierung der Ereignisempfänger-Schnittstellen

```
public class TemperaturApplet extends Applet implements ActionListener {
```

```
    // Instantiierung der Komponenten
```

```
    public void init () {
```

```
        // Zusammenstellung des Containers
```

```
        // Assoziation von Ereignissen und Ereignisschnittstellen
```

```
    } // init
```

```
    public void actionPerformed (ActionEvent e) {
```

```
        String eingabe = eingabeFeld.getText(); // Lies Eingabe vom Benutzer
```

```
        double eingabeZahl = Double.valueOf(eingabe).doubleValue();
```

```
        double resultat = 0.0;
```

```
        if (e.getSource() == cfButton) {
```

```
            resultat = temperature.celsToFahr(eingabeZahl);
```

```
            ausgabeFeld.setText(eingabeZahl + "°C = " + resultat + "°F");
```

```
        } else if (e.getSource() == fcButton) {
```

```
            resultat = temperature.fahrToCels(eingabeZahl);
```

```
            ausgabeFeld.setText(eingabeZahl + "°F = " + resultat + "°C ");
```

```
        }
```

```
    } // actionPerformed
```

```
} // TemperaturApplet
```

**Deklaration des
Ereignisempfänger-Schnittstelle
ActionListener**

**Implementierung von
actionPerformed
(öffentliche Methode
in ActionListener)**

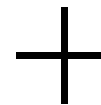
Exekution von Applets

- ❖ Mit Applet Viewer
- ❖ Mit Web-Browser

HTML-Code zum Einbetten des Applets

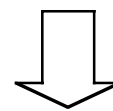
```
<! -- *****  
file:Temperatur.html  
-->  
<html>  
<body>  
<applet  
  code="TemperaturApplet.class"  
  width="300" height="200">  
</applet>  
</body>  
</html>
```

.html-Datei



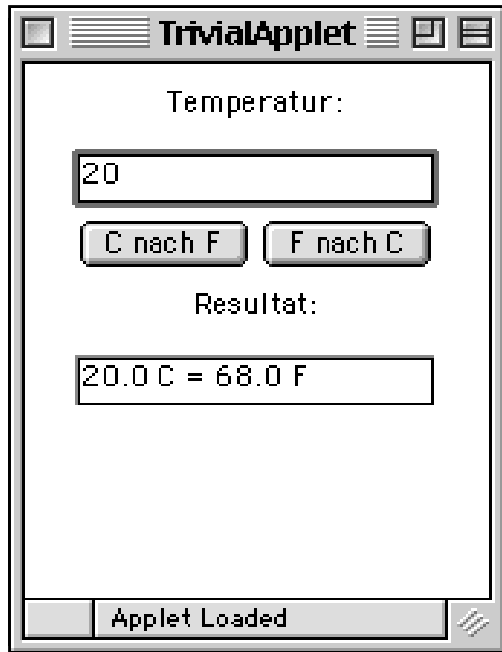
```
TemperaturApplet.class  
1 2  
.fæ-ç<clinit>() V  
eingabeFeld[]Ljava/awt/TextField;  
ausgabeFeld[]schrift1[]Ljava/awt/L  
celsToFahr[]Ljava/awt/Button;  
fahrToCels  
temperature  
ITemperature;[]init  
[]TemperatureApplet[] []add+  
[][]java/awt/Container[]  
[] [] []  
[] [] []![]addActionListener  
#$[]java/awt/Button&  
'%[]setSize[](II)V)*[]java/awt/Co  
-+[]Code[]actionPerformed(Ljava/  
V[]getText[]()Ljava/lang/String;  
64[]valueOf&(Ljava/lang/String);)  
89[]java/lang/Double;  
<:  
doubleValue[]()
```

.class-Datei

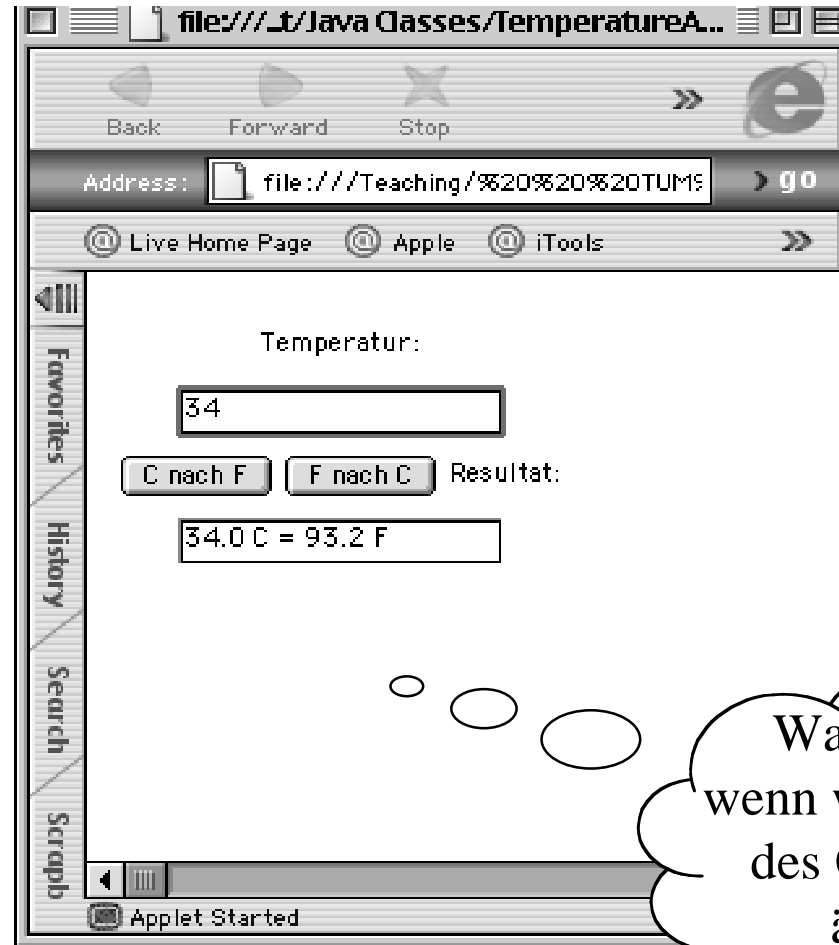


Web-Browser (Netscape Navigator, Internet Explorer, iCab, ...)

Exekution des Temperatur-Konvertiers



Mit Applet Viewer



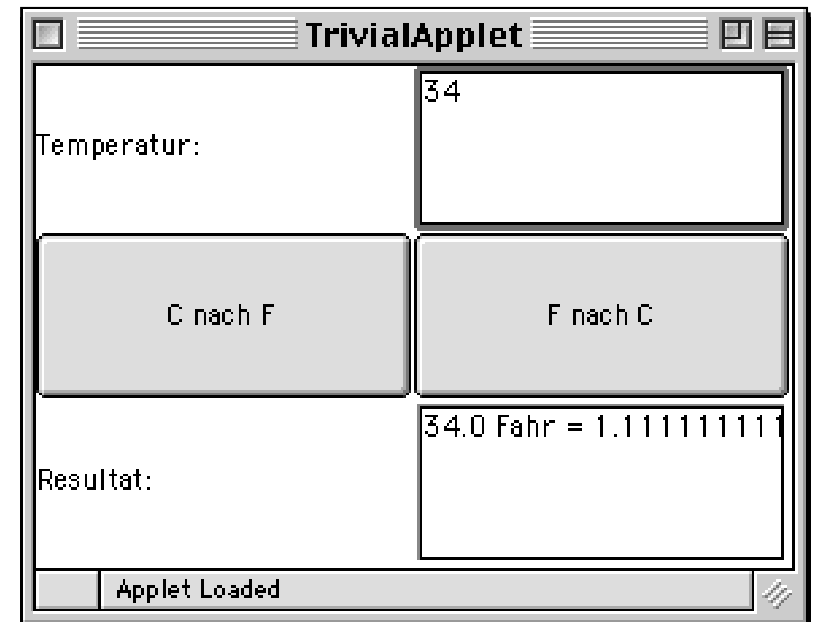
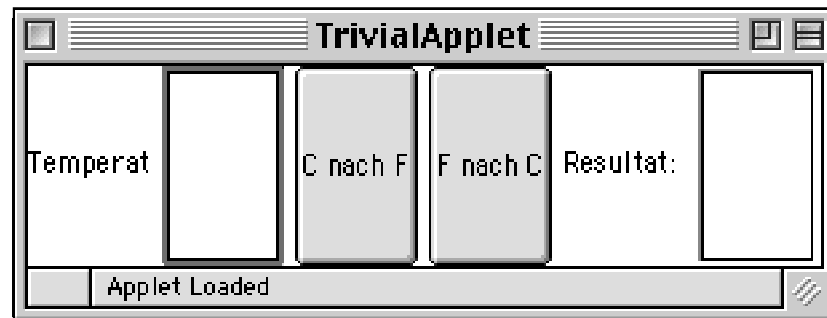
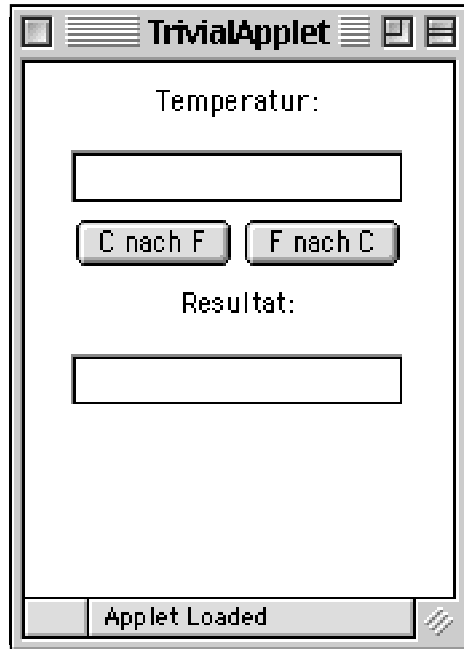
Mit Web-Browser

Was passiert,
wenn wir die Größe
des Containers
ändern?

Layout-Manager

- ❖ Ein Container ist eine Komponente, die andere Komponenten enthalten kann. Ein Container ist ein einfaches Objekt, dessen Hauptaufgabe es ist, die Komponenten in einer bestimmten Ordnung zu halten (**add ()**, **remove ()**).
- ❖ Ein wichtiges Problem ist es, diese Komponenten visuell anzuordnen.
- ❖ **Definition Layout-Manager:** Ein Layout-Manager ist ein Objekt, das die visuelle Anordnung eines Containers bestimmt.
Seine Aufgaben sind:
 - ◆ Bestimmung der Gesamtgröße der Container-Darstellung
 - ◆ Bestimmung der Darstellungsgrößen der Komponenten im Container
 - ◆ Bestimmung des Abstandes zwischen den Komponenten
 - ◆ Bestimmung der Darstellungsposition jeder Komponente

Verschiedene Layouts für das Temperatur-Applet

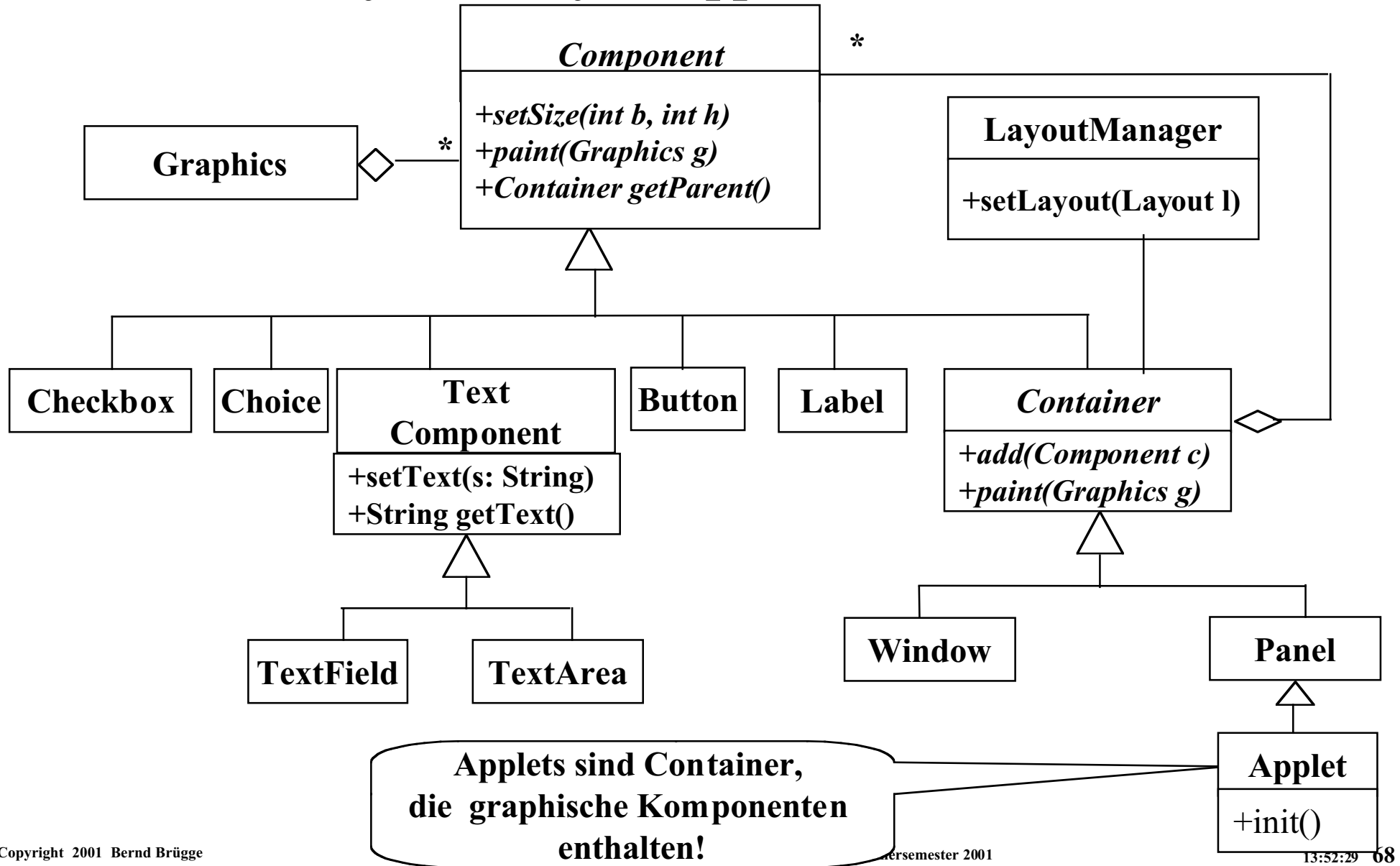


Layout-Manager in Java

- ❖ Sie könnten Ihren eigenen Layout-Manager schreiben.
- ❖ Nicht sehr empfehlenswert: Java enthält eingebaute Layout-Manager.

Layout Manager	Beschreibung
java.awt.FlowLayout	Arrangiert die Komponenten von links nach rechts <i>(Dies ist die Voreinstellung für AWT)</i>
java.awt.GridLayout	Arrangiert die Komponenten als 2-dimensionale Reihung von gleichgroßen Zellen
java.awt.BorderLayout	Arrangiert die Komponenten mit Hilfe von Kompassrichtungen (Nord, Süd, Ost, West)
java.awt.CardLayout	Arrangiert die Komponenten wie einen Stapel von Karten
java.swing.BoxLayout	Arrangiert die Komponenten in einer Reihe oder einer Spalte
java.swing.OverlayLayout	Arrangiert die Komponenten überlappend

Applets und Layout-Manager: Ausschnitt aus java.awt, java.applet

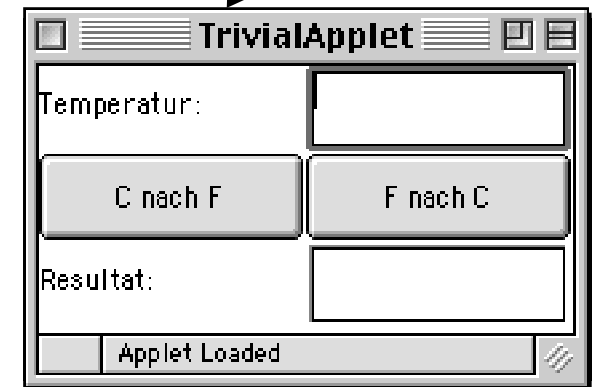
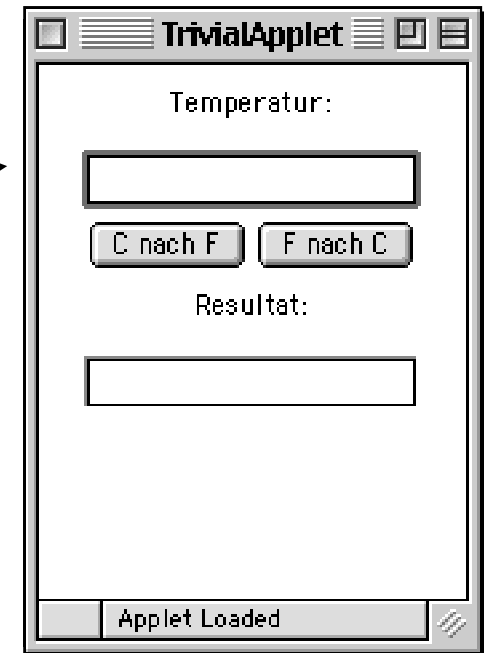


Beispiel von Layouts

- ❖ Das Layout wird in üblicherweise in der `init()`-Methode des Applets festgelegt:

```
public void init() {  
    setLayout(new FlowLayout());  
    //Voreinstellung bei AWT  
    setSize(160,200); // Applet-Größe  
}
```

```
public void init() {  
    setLayout(new GridLayout(3,2,1,1));  
    setSize(200,100);  
}
```



Parameter des `GridLayout()`-Konstruktors:
Anzahl der Reihen, Anzahl der Spalten,
Abstand zwischen den Reihen, Abstand zwischen den Spalten

Entwurf von Bedienoberflächen

- ❖ Eine Bedienoberfläche muss folgende Elemente haben:
 - ◆ Eingabe von Information
 - ◆ Ausgabe von Information
 - ◆ Führung/Hilfe für die Benutzer
 - ◆ Kontrolle der Interaktion zwischen den Benutzern und den Komponenten aus dem Anwendungsmodell
 - ◆ Intuitive Benutzbarkeit und Robustheit
- ❖ **Wichtige Fragen, die beim Entwurf zu stellen sind:**
 - ◆ Wie können Benutzer ihre Aufgaben am effektivsten erfüllen?
 - ◆ Welche Komponenten sind am besten für die Interaktion geeignet?
 - ◆ Wie kann man die Komponenten am besten visuell repräsentieren?
 - ◆ Was können Benutzer alles falsch machen?
- ❖ **Allgemeines Problem: Mensch-Maschine-Kommunikation**
⇒ Hauptstudium

Dynamische Eigenschaften eines Systems

- ❖ Zur Modellierung von Systemen haben wir bisher vorwiegend Klassendiagramme und Anwendungsfälle benutzt.
 - ◆ Sie sind sehr nützlich bei der Beschreibung der *statischen* und *funktionellen* Eigenschaften eines Informatik-Systems (Klasse "*Berechnung von Funktionen*", siehe Info I - Vorlesung 2).
- ❖ Klassendiagramme und Anwendungsfälle sind nicht geeignet, um die *dynamischen* Eigenschaften eines Systems, d.h. den Informationsfluss und Zustandsübergänge im System zu beschreiben.
 - ◆ Bei offenen Informatik-Systemen wie z.B. beim Typ "*Interaktives System*" sind die dynamischen Eigenschaften genauso wichtig wie die statischen oder funktionellen Eigenschaften - und oft interessanter als diese.

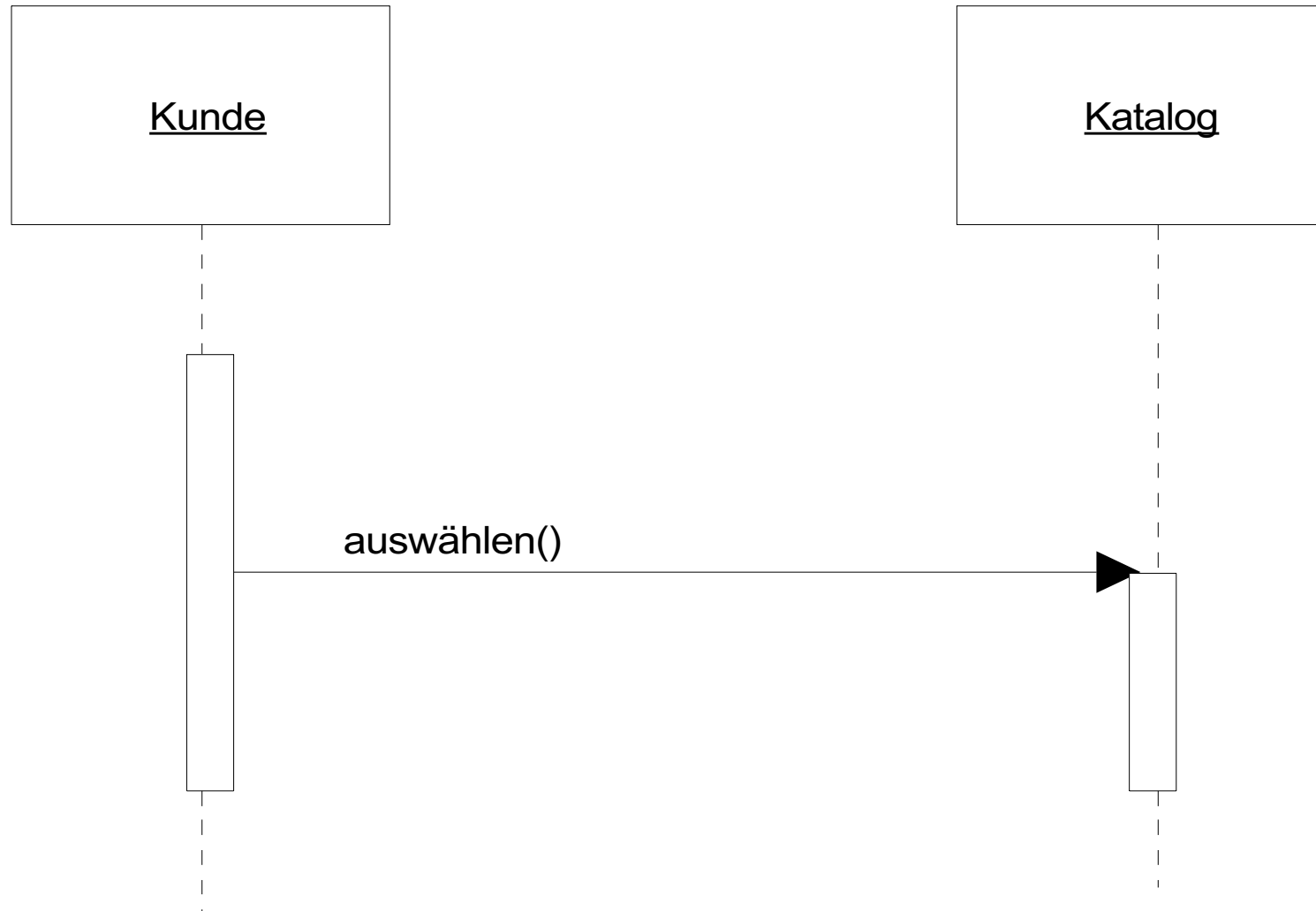
Modellierung der dynamischen Eigenschaften eines Systems

- ❖ Wir modellieren die dynamischen Eigenschaften eines Systems in UML mit 2 weiteren Notationen:
- ❖ **Sequenz-Diagramm:** Eine UML-Notation für die Modellierung des Informationsflusses zwischen *mehreren Objekten in einem Anwendungsfall*.
 - ⇒ Dieser Vorlesungsblock
- ❖ **Zustands-Diagramm:** Eine UML-Notation für die Modellierung der Zustandsübergänge eines *einzelnen Objektes in mehreren Anwendungsfällen*.
 - ⇒ Vorlesungsblock über Automaten

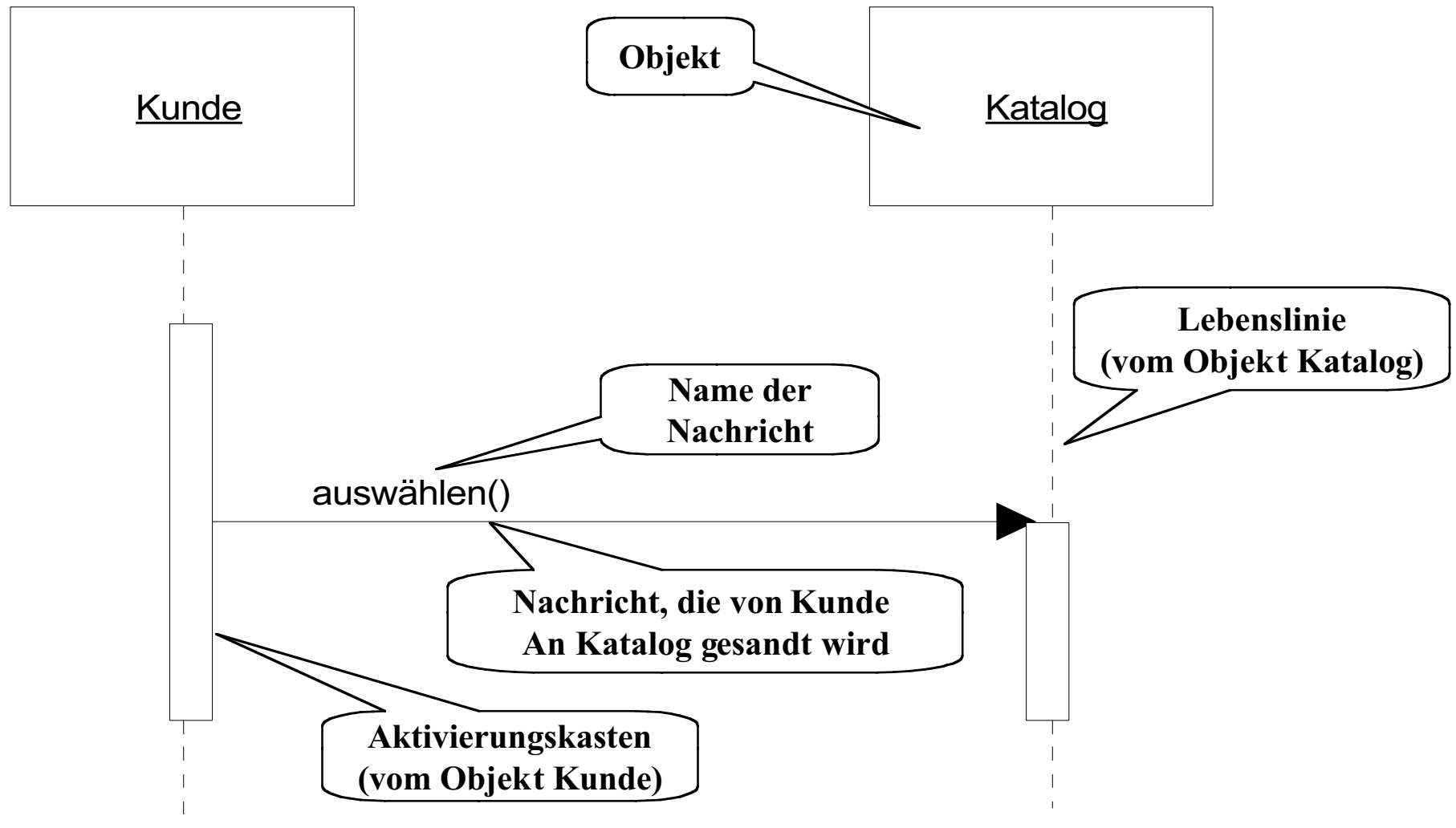
UML-Interaktionsdiagramm

- ❖ **Definition:** Ein **Interaktionsdiagramm** beschreibt die Zusammenarbeit einer *Gruppe* von *Objekten* (*nicht Klassen!*) in einem *Anwendungsfall*
- ❖ Es gibt 2 Arten von Interaktionsdiagrammen:
 - ◆ **Sequenzdiagramme** betonen den Ablauf von Ereignissen. Objekte werden als Kästchen mit einer gestrichelten Linie (auch *Lebenslinie* genannt) gezeichnet. Die Objekte kommunizieren über Nachrichten, die als Pfeile zwischen den Lebenslinien gezeichnet werden.
 - ◆ **Kollaborationsdiagramme** benutzen das selbe Layout wie Instanzdiagramme: Die Assoziationen werden mit nummerierten Pfeilen annotiert, welche den Informationsfluss zwischen den Objekten darstellen.
- ❖ In Info II behandeln wir nur Sequenzdiagramme

UML-Sequenzdiagramm: Beispiel



UML-Sequenzdiagramm: Beispiel



UML-Sequenzdiagramm: Konzepte

- ❖ Sequenzdiagramme modellieren die Interaktion von Objekten, die über Nachrichten kommunizieren.
 - ◆ Die **Lebenslinie** repräsentiert die Lebensdauer des Objektes während der Interaktion
 - ◆ Der optionale **Aktivierungskasten** markiert einen Bereich der Lebenslinie, in dem das Objekt aktiv ist
 - ◆ Jede **Nachricht** wird durch einen Pfeil zwischen den Lebenslinien zweier Objekte dargestellt
 - ◆ Ein Objekt kann sich auch selbst Nachrichten schicken. In diesem Fall zeigt das Ende des Nachrichtenpfeils auf die Lebenslinie desselben Objektes.
 - ◆ Als Resultat einer Nachricht kann ein Objekt einen Wert zurücksenden. Die **Wertrückgabe** als Antwort auf eine Nachricht wird als gestrichelter Pfeil dargestellt.

Modellierung von Interaktiven Systemen

1. Modellierung der *funktionalen* Aspekte:

Vom Szenario zum Anwendungsfall, insbesondere Ereignisfluss

2. Modellierung der *dynamischen* Aspekte:

Vom Ereignisfluss zum Sequenzdiagramm

3. Modellierung der *statischen* Aspekte:

Vom Sequenzdiagramm zum Klassendiagramm

❖ **Beispiel** (aus Info II - Vorlesung 1):

- ◆ Szenario: Der Kunde durchstöbert einen Katalog und legt die ausgewählten Artikel in seinen Einkaufskorb. Zum Zahlen gibt er seine Versand- und Kreditkarteninformation an und bestätigt seinen Kauf. Das System autorisiert den Verkauf über eine Kreditkarte und bestätigt den Verkauf mit einer Quittung und einer E-Mail an die E-Mail-Adresse aus der Versandinformation.

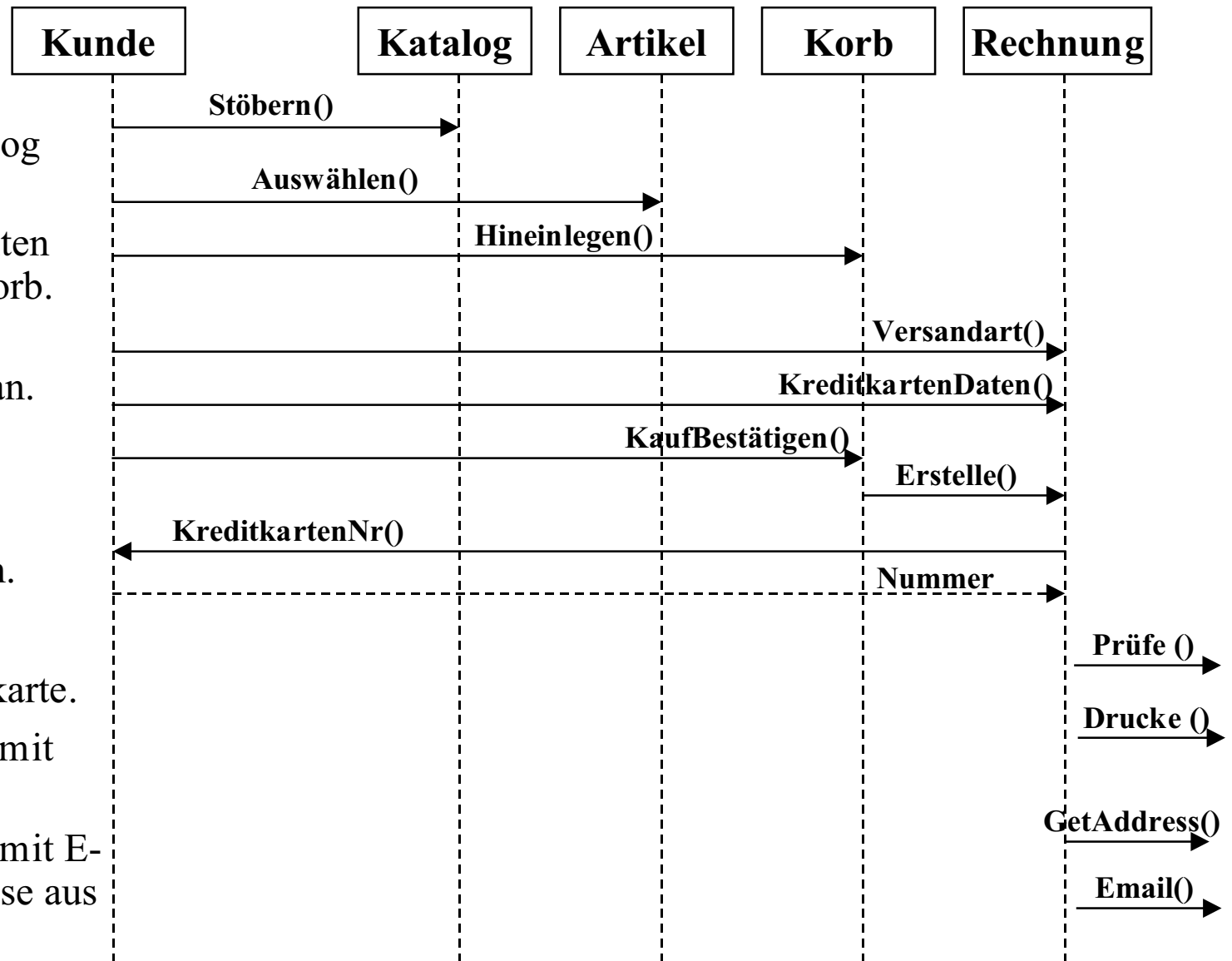
Modellierung als Anwendungsfall

- ❖ **Name** des Anwendungsfalls: Elektronischer Einkauf
- ❖ **Akteur**: Kunde
- ❖ **Ereignisfluss**:
 - ◆ Kunde durchstöbert den Katalog.
 - ◆ Kunde wählt Artikel aus.
 - ◆ Kunde legt die ausgewählten Artikel in den Einkaufskorb.
 - ◆ Kunde gibt Versand- und Kreditkarteninformation an.
 - ◆ Kunde bestätigt Kauf.
 - ◆ Kunde gibt Kreditkarte an.
 - ◆ System autorisiert die Kreditkarte.
 - ◆ System bestätigt den Verkauf mit Quittung.
 - ◆ System bestätigt Verkauf mit E-Mail an die E-Mail-Adresse aus der Versandinformation.

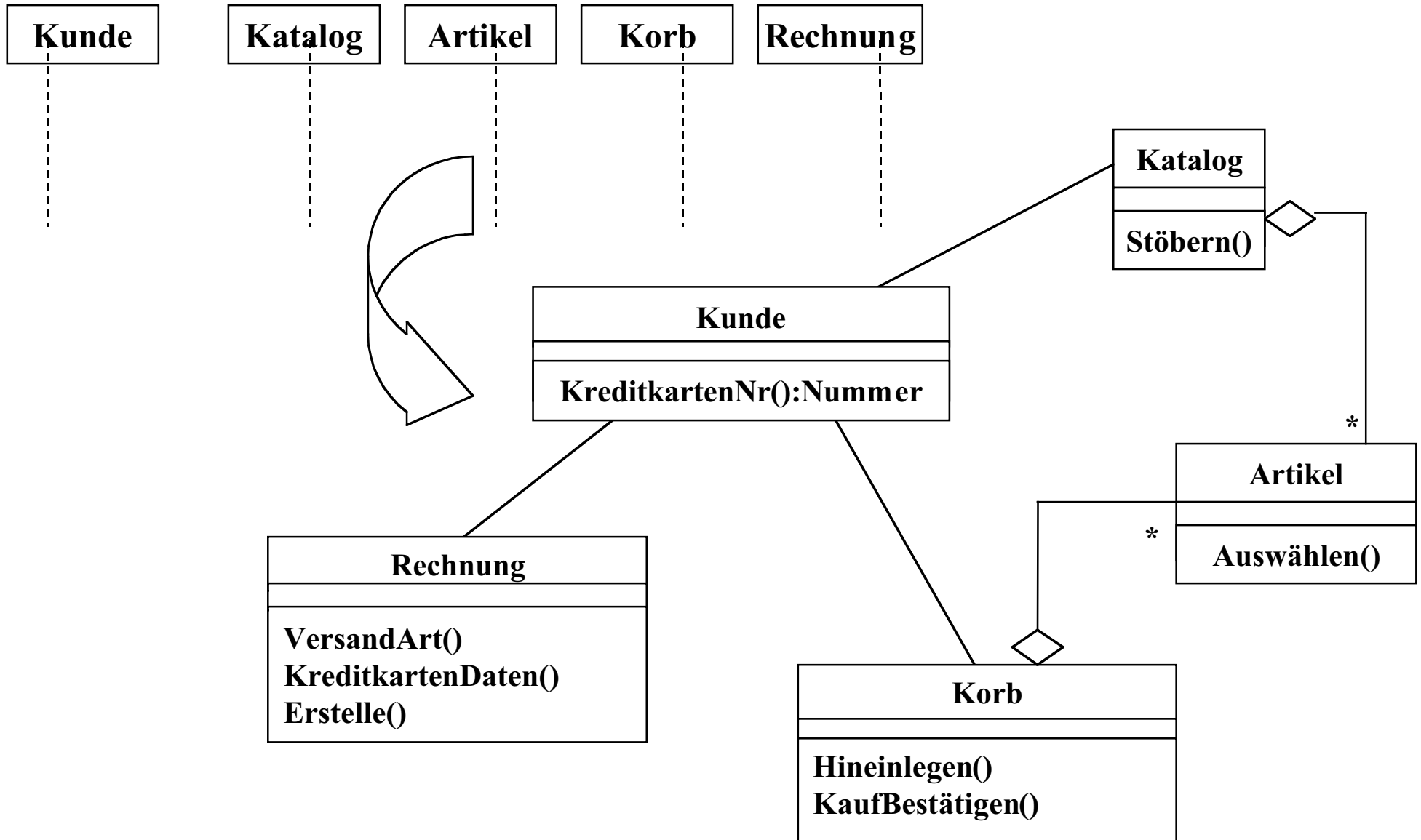
Vom Anwendungsfall zum Sequenzdiagramm

Ereignisfluss:

- ❖ Kunde durchstöbert Katalog
- ❖ Kunde wählt Artikel aus.
- ❖ Kunde legt die ausgewählten Artikel in den Einkaufskorb.
- ❖ Kunde gibt Versand- und Kreditkarteninformation an.
- ❖ Kunde bestätigt Kauf.
- ❖ Kunde gibt Kreditkarte an.
- ❖ System autorisiert Kreditkarte.
- ❖ System bestätigt Verkauf mit Quittung.
- ❖ System bestätigt Verkauf mit E-Mail an die E-Mail-Adresse aus der Versandinformation.



Vom Sequenzdiagramm zum Klassendiagramm



Heuristiken für Modellierung mit Sequenzdiagrammen und Klassendiagrammen

Sequenzdiagramme:

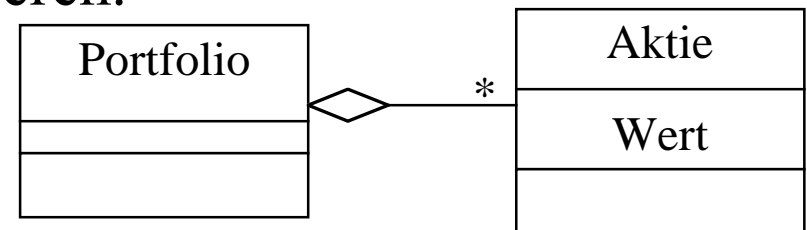
- ❖ Erstelle *ein* Sequenzdiagramm für *jeden* Anwendungsfall.
 - ◆ Jeder Schritt im *Ereignisfluss* des Anwendungsfalles erzeugt eine oder mehrere Nachrichten
- ❖ Sequenzdiagramme sind lesbarer, wenn man nicht für jede Nachricht die Rückkehr zeichnet.

Klassendiagramme:

- ❖ Objekte in Sequenzdiagrammen sind mögliche Repräsentanten für neue Klassen
- ❖ Nachrichten sind gute Kandidaten für (öffentliche) Operationen auf den Klassen, zu deren Instanzen sie geschickt wurden
 - ◆ Oft muss man die Namen allerdings verändern
- ❖ Rückgaben werden zu Rückgabewerten von Operationen

Ein immer wiederkehrendes Entwurfsproblem

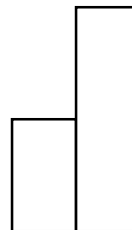
- ❖ **Ausgangspunkt:** Ein Objekt repräsentiert einen wichtigen Zustand. Es gibt viele Interessenten für diesen Zustand. Immer wenn sich der Zustand ändert, sollen alle Interessenten davon benachrichtigt werden.
 - ◆ **Beispiel:** Eine Portfolio hat einen Wert. Wir wollen diesen Wert durch verschiedene Sichten repräsentieren.



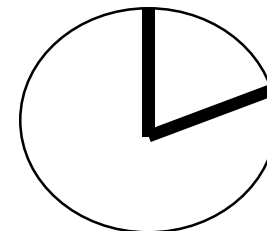
Tabellensicht

AAPL	20
MSFT	80

Histogrammsicht



Tortensicht

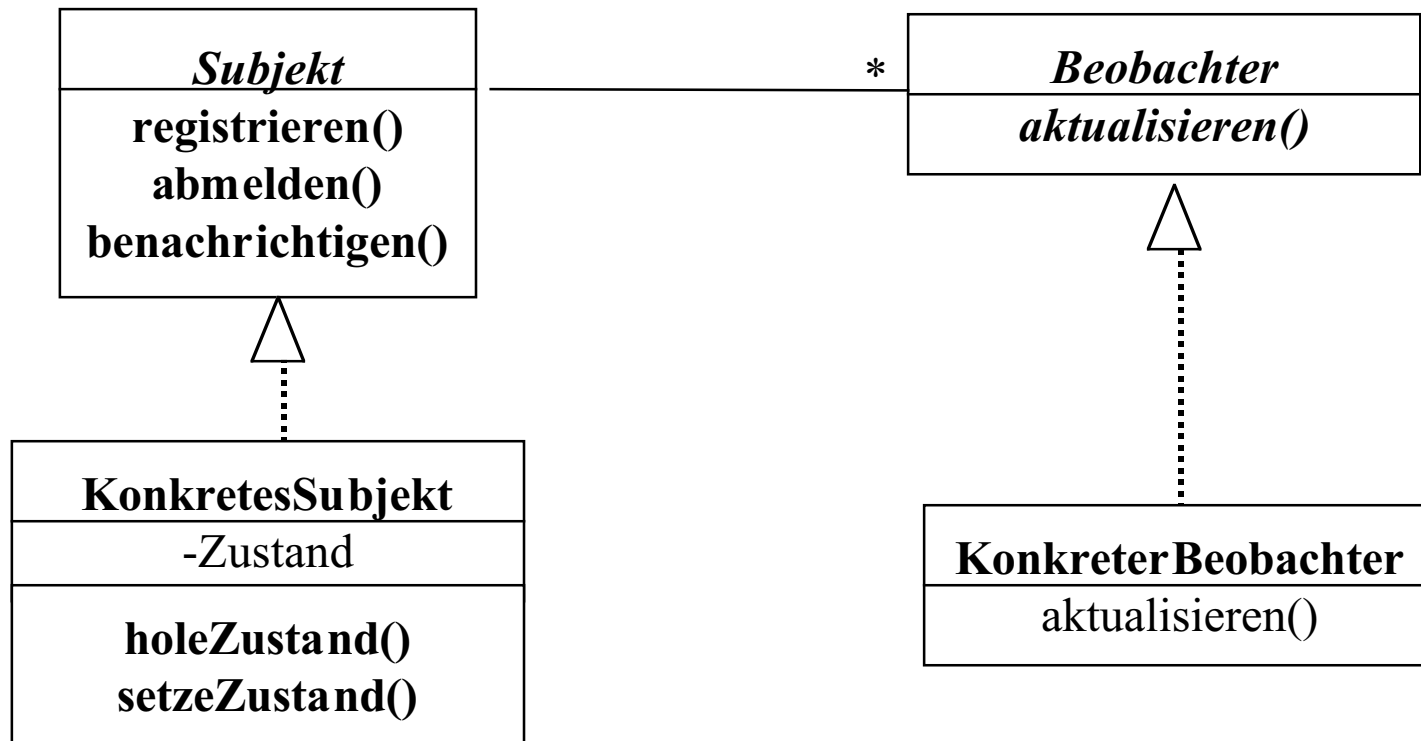


- ❖ **Problem:** Wie können wir Konsistenz zwischen den Sichten garantieren?
 - ◆ **Beispiel:** Wie können wir garantieren, dass alle Sichten immer den gleichen Zustand des Portfolios repräsentieren?

Lösung: Das Beobachter-Muster

- ◆ Eine 1-* Assoziation zwischen dem *Objekt* und den *Interessenten*.
 - ◆ Wenn das Objekt seinen Zustand ändert, werden alle Interessenten benachrichtigt.
 - ◆ Die Interessenten können dann ihre Repräsentation des Objektzustands aktualisieren.
- ❖ Andere Bezeichnungen:
- ◆ *Objekt*: **Subjekt**, Modell, Herausgeber (publisher)
 - ◆ *Interessent*: **Beobachter**, Sicht, Abonnent (subscriber)

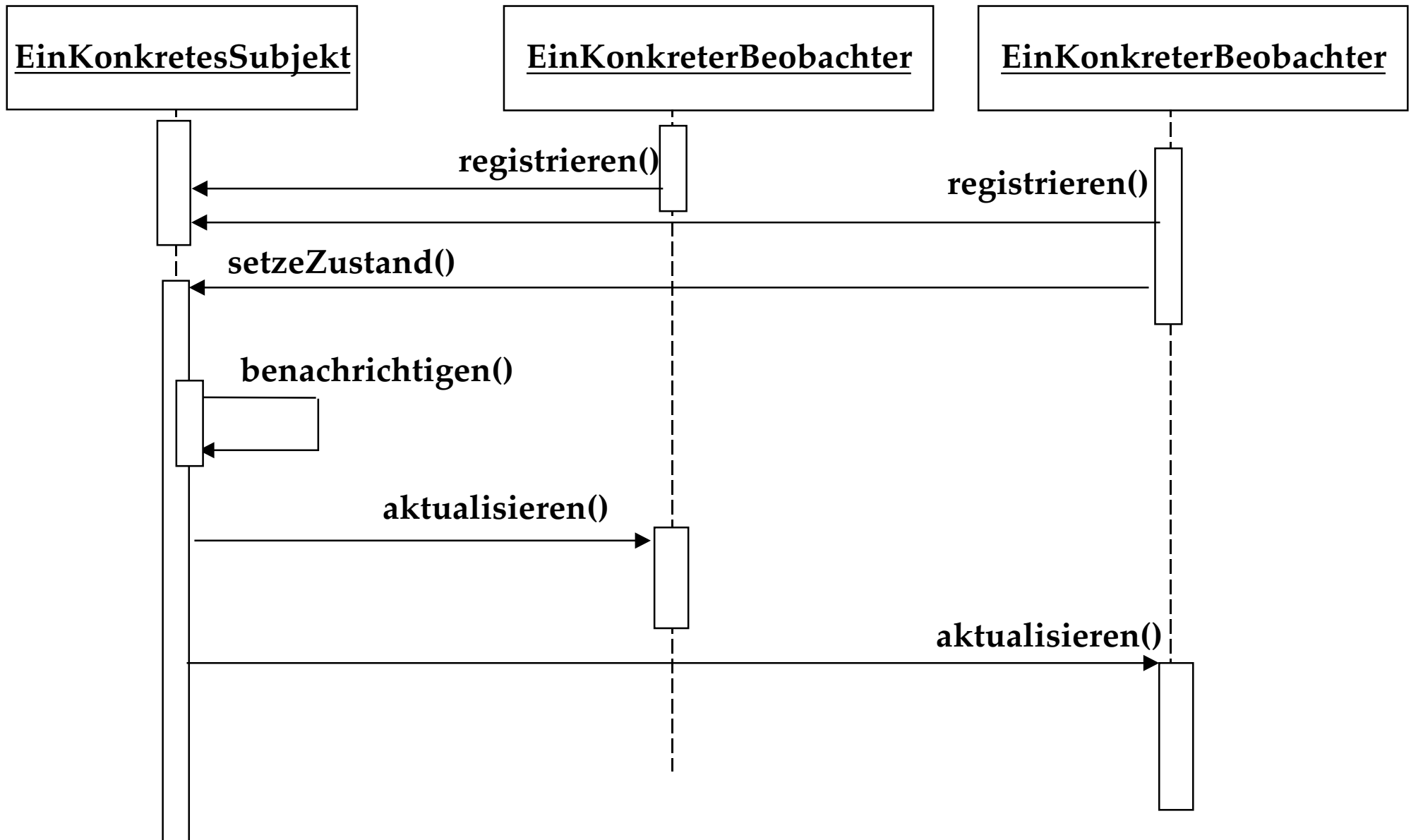
Beobachter-Muster (Observer Pattern)



Das Beobachter-Muster gibt es in 2 Varianten

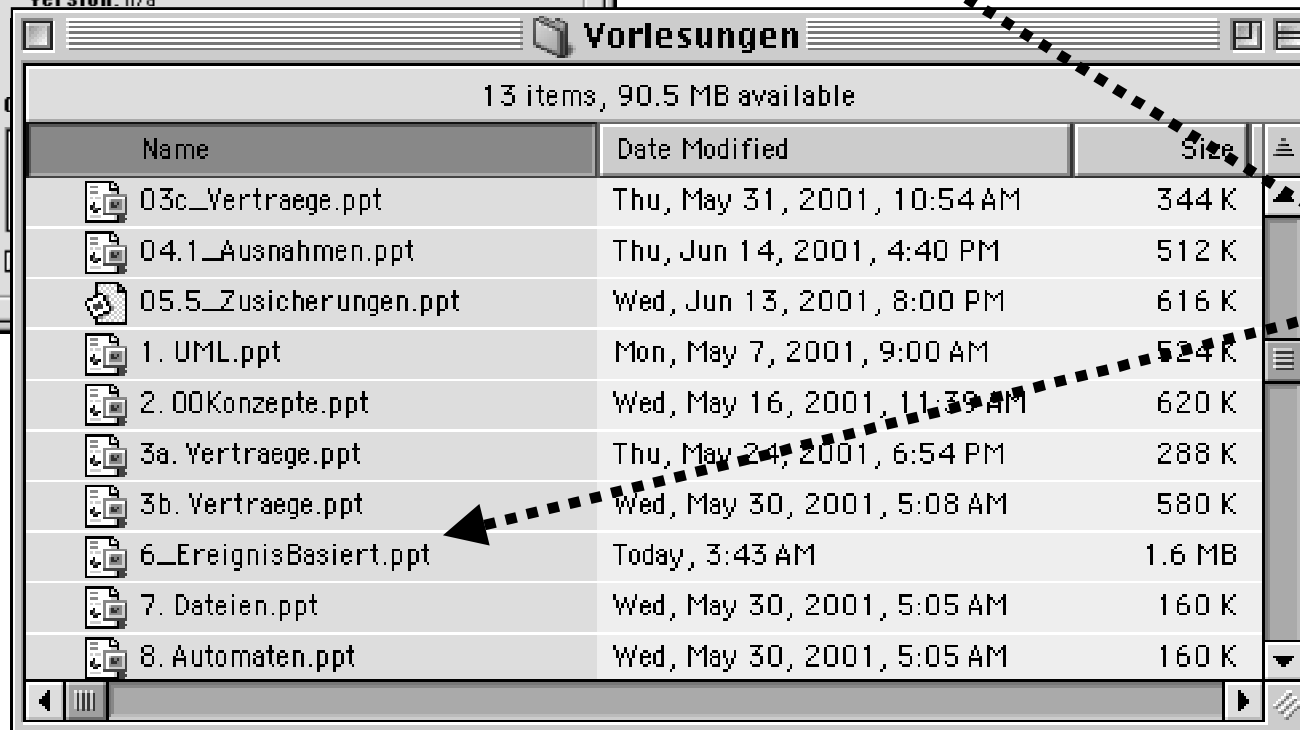
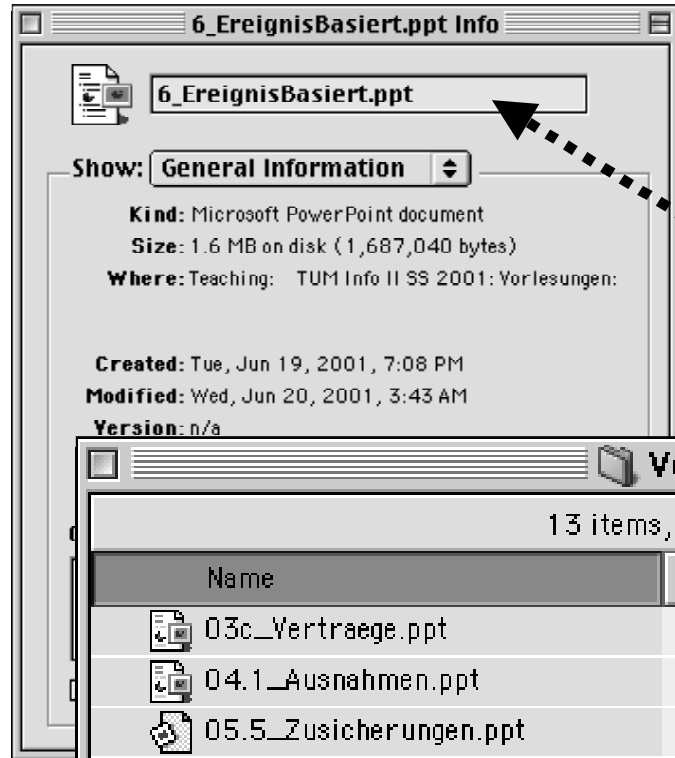
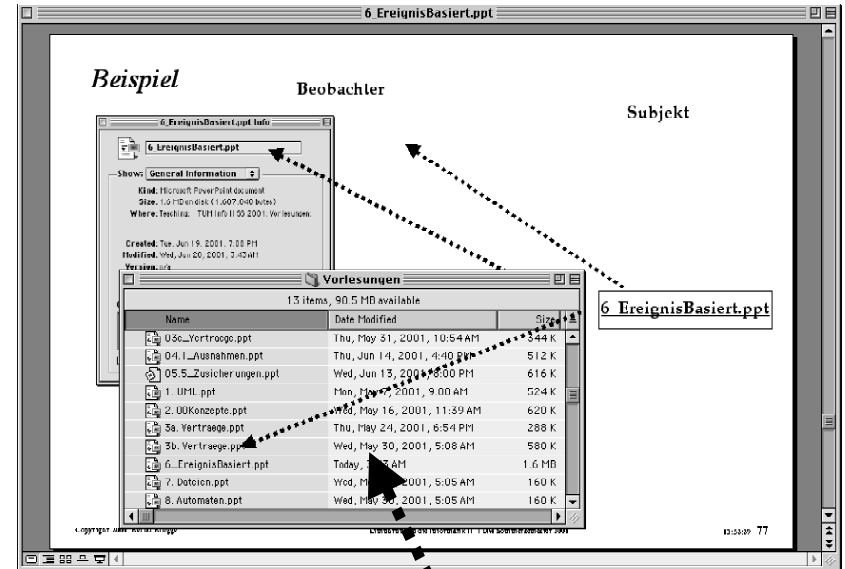
- ❖ **"Push-Variante"**: Wenn das Subjekt die Beobachter zur Aktualisierung auffordert, schickt es gleich den neuen Zustand mit.
 - ◆ sinnvoll, wenn der Zustand durch wenige Attribute beschrieben wird (*"Unfall in Arcisstrasse 21"*)
- ❖ **"Pull-Variante"**: Das Subjekt sagt den Beobachtern nur, dass sich der Zustand geändert hat; die Beobachter fragen dann beim Subjekt nach, wie der neue Zustand ist.
 - ◆ wird oft benutzt, wenn der Zustand durch viele Attribute beschrieben wird, oder wenn eines der Attribute sehr groß ist (*"Update erhältlich"*, *"Neuer Film gerade herausgekommen"*, *"Zeitung von heute verfügbar"*).
 - ◆ Beispiel: Sie schicken einem Bekannten eine URL, die auf eine interessante Nachricht zeigt.

Sequenzdiagramm für Beobachtermuster ("Push-Variante")

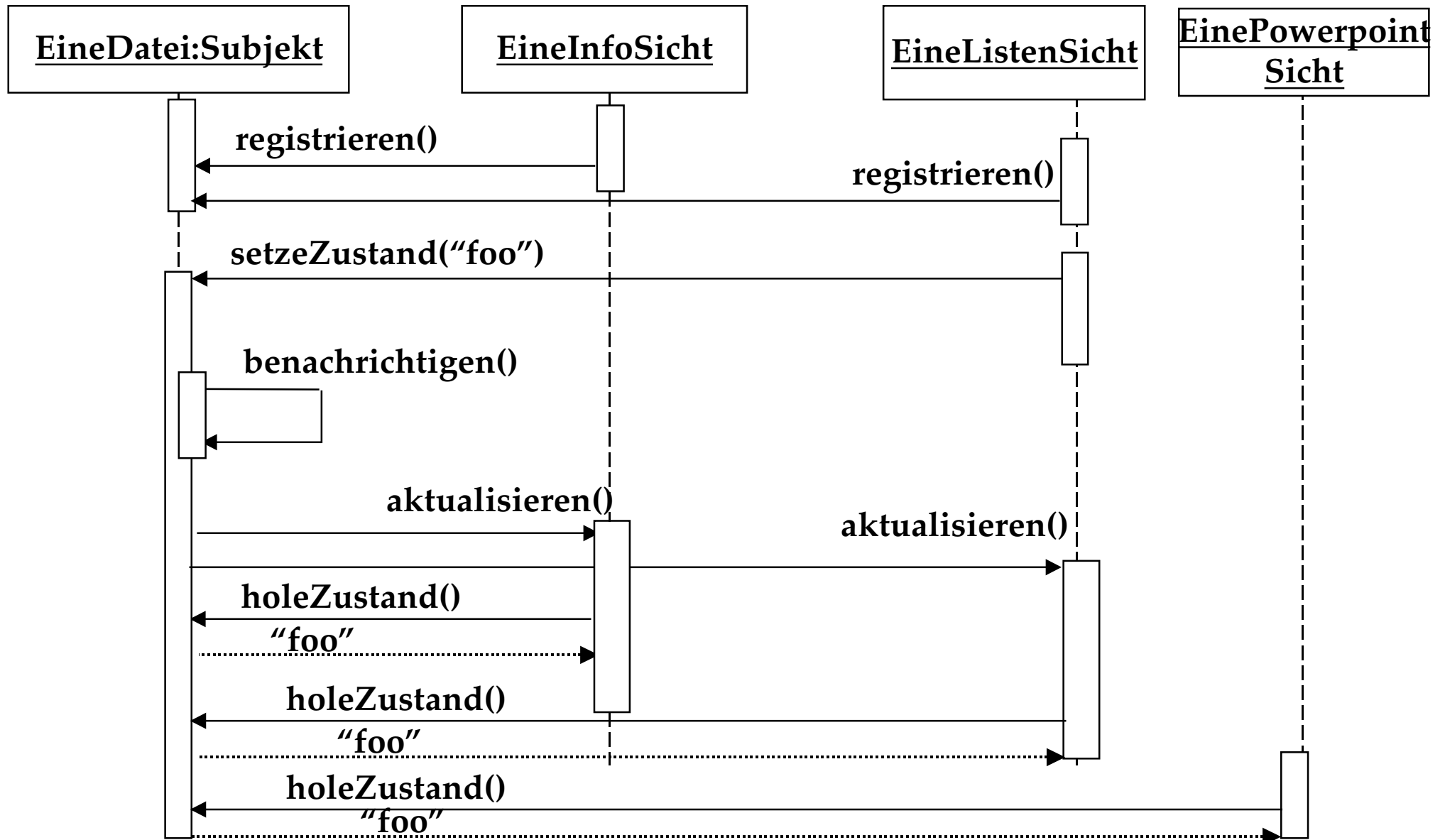


Beispiel

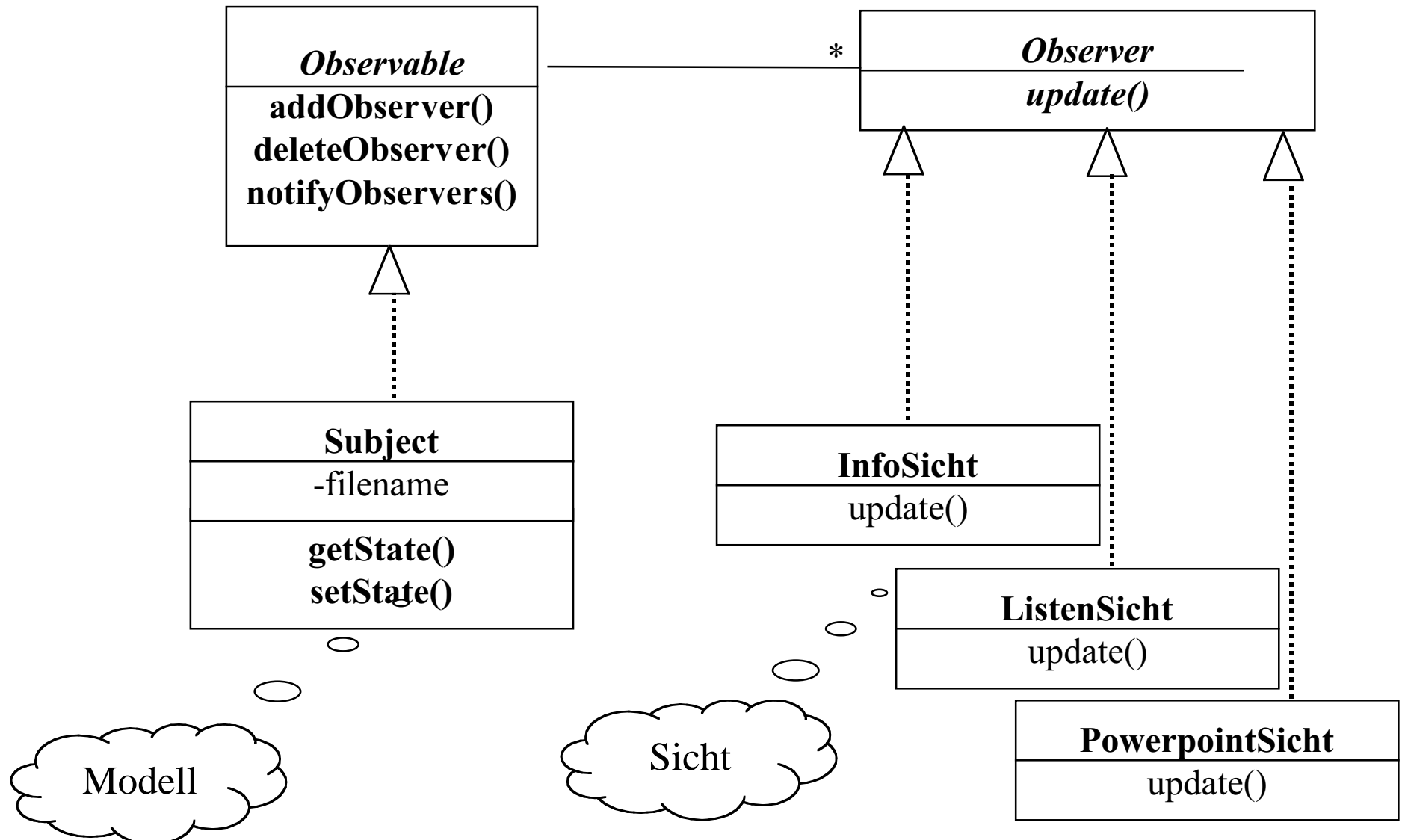
Beobachter



Sequenzdiagramm "Ändere Dateiname" (Pull-Variante)



Detaillierter Entwurf:

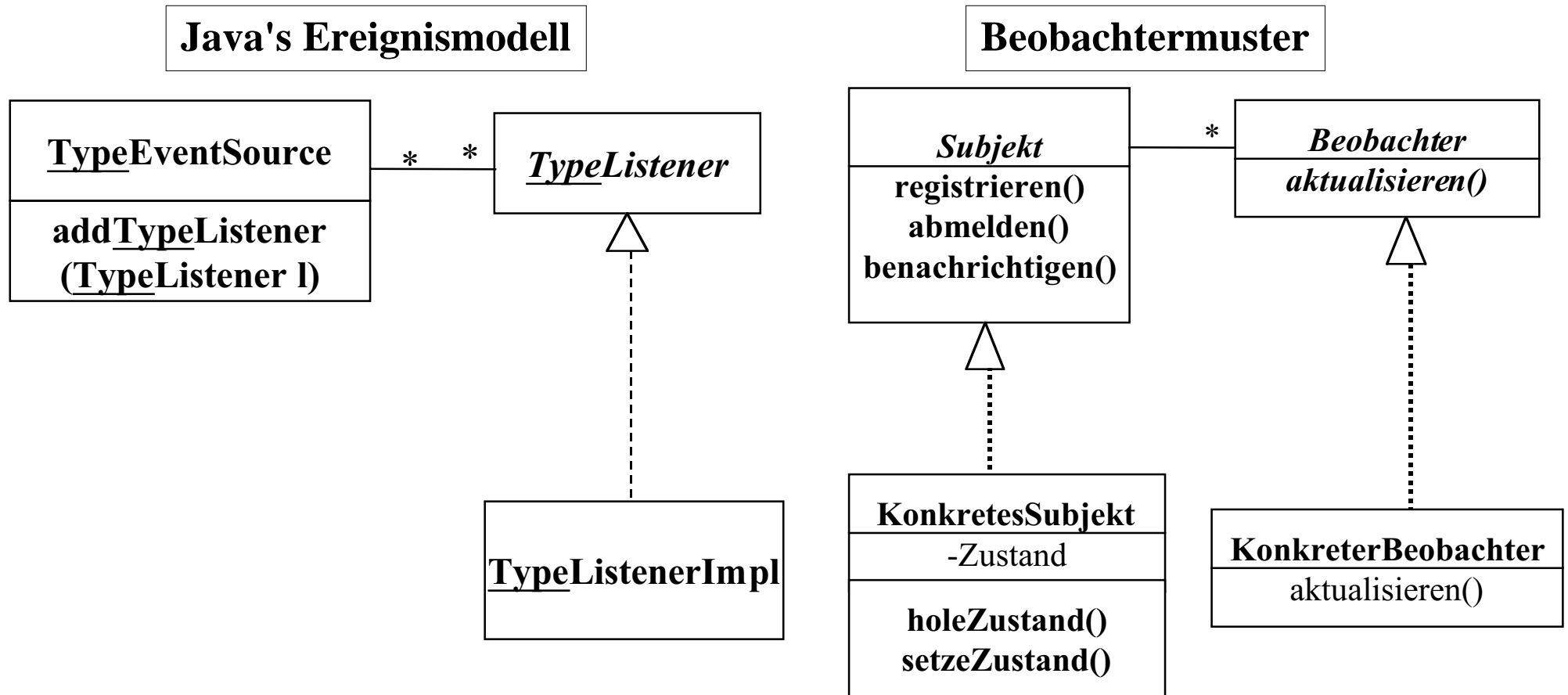


Implementierung von Subject mit Java's Klassenbibliothek

```
// import java.util;
public class Observable extends Object {
    public void addObserver(Observer o);
    public void deleteObserver(Observer o);
    public boolean hasChanged();
    public void notifyObservers();
    public void notifyObservers(Object arg);
}
public abstract interface Observer {
    public abstract void update(Observable o, Object arg);
}
public class Subject extends Observable {
    public void setState(String filename);
    public string getState();
}
```

Java's Ereignismodell und Beobachtermuster

- ❖ Beobachtung: Java's Ereignismodell für graphische Komponenten basiert auf dem Beobachtermuster.



Zusammenfassung I

- ❖ Bei der Entwicklung von Ereignis-basierten Systemen benötigen wir:
 - ◆ Modellierung von *Ereignissen*
 - ◆ Modellierung von *Ereignisquellen und -empfängern*
 - ◆ Spezifikation von Ereignisquellen und -empfängern
 - ◆ Implementierung von Ereignisquellen und -empfängern
 - ◆ Modellierung von *Komponenten*
 - ◆ Modellierung der Anwendungsdomäne
- ❖ Bei interaktiven Systemen benötigen wir noch zusätzlich:
 - ◆ Modellierung der Benutzerschnittstelle:
 - ◆ Modellierung der Bedienoberfläche mit *graphischen Komponenten*
 - ◆ Modellierung der Dialoge mit dem Benutzer

Zusammenfassung II

- ❖ Ereignis-basierte interaktive Systeme lassen sich in Java sehr gut mit Applets implementieren.
- ❖ Beispiel für Ereignis-basierte interaktive Systeme
 - ◆ Graphische Benutzeroberflächen(Graphical User Interfaces (GUI))
- ❖ Applets benutzen Konzepte aus 2 Programmierparadigmen:
 - ◆ Ereignis-basierte Programmierung:
 - ◆ Der Kontrollfluss des Programms wird durch Ereignisse (insbesondere durch von graphischen Komponenten ausgelöste Ereignisse) bestimmt.
 - ◆ Ereignisse werden dynamisch an Ereignisempfänger gebunden, die beim Eintreten dieser Ereignisse aufgerufen werden.
 - ◆ Objekt-Orientierung:
 - ◆ Vererbung, Methodenüberschreibung, dynamischer Polymorphismus
- ❖ Sequenzdiagramme erlauben die Modellierung des Informationsflusses zwischen *mehreren* Objekten in *einem Anwendungsfall*.