

Medientechnik

Heinrich Hußmann
Ludwig-Maximilians-Universität München
Sommersemester 2010

Lehr- und Forschungseinheit Medieninformatik

Prof. Dr. Heinrich Hußmann

Amalienstr. 17, 5. OG, Raum 508

Email hussmann@ifi.lmu.de

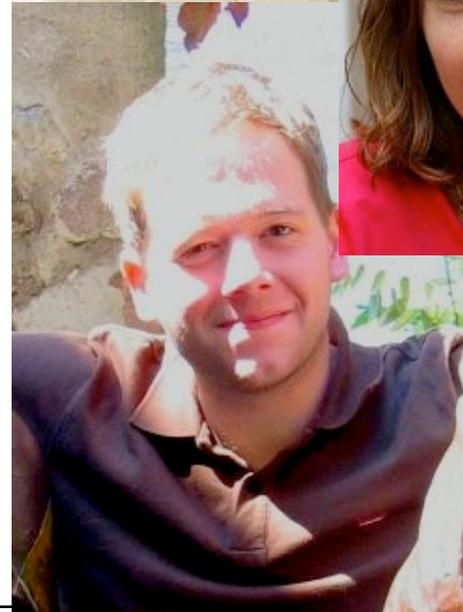
Übungsleitung:

Sara Streng, 5. OG, Raum 505

Fabian Hennecke, 2. OG, Raum 203

Wichtigste Informationsquelle: WWW

- Folien (vor der Vorlesung)
- Multimedia-Aufzeichnung
- Übungsaufgaben
- Aktuelle Informationen



Inhalt der Vorlesung

- Diese Vorlesung: Ergänzendes Wissen zu digitalen Medien
 - Teilweise aufbauend auf „Digitale Medien“
 - Technische Grundlagen der Multimediatechnik
 - Basisinformationen zu Algorithmen der Medienbearbeitung
- ***Achtung: Im Vergleich zu früheren Vorlesungen gleichen Titels (vor 2009) deutlich reduzierter Programmieranteil!***
- Themen:
 - Programmierung grafischer Benutzungsoberflächen (Bsp. Java Swing)
 - Grundlagen der Fototechnik, digitale Fotografie, Bildbearbeitung
 - Film- und Videotechnik analog und digital, digitaler Filmschnitt
 - Audio-Aufnahme- und Wiedergabetechnik, Tonbearbeitung
 - Ein- und Ausgabegeräte, Speichermedien, digitale Hardware-Schnittstellen

Vorlesung und Übungen

- Vorlesung "Medientechnik":
 - Konzepte, Überblickswissen,
keine vollständigen technischen Detailinformationen
- Übungen "Medientechnik":
 - Praktische Anwendung und Ergänzung des Vorlesungsstoffs
 - Zum Themenschwerpunkt Hardware/Technik:
 - » Drei **Praktika** (in kleinen Gruppen):
Praktische Arbeit zu Foto, Video, Audio
 - Zum Thema Programmierung:
 - » **Übungsgruppen**, (wenige) Hausaufgaben

Übungen und Praktika (1)

- Programmierübungen:
 - Einzel zu bearbeiten
 - Übungsblätter jeweils Montag vor der Übung online
 - Abgabe in der nächsten Woche nach der Übung am Mittwoch (bis 18 Uhr) über UniWorX
 - Anmeldung zu einer Übung verpflichtend, d.h. nur Besuch dieser Übung möglich
- Praktika (Foto, Video, Audio):
 - In Vierergruppen
 - Jeweils 3 Stunden Praktikum, Nachbearbeitung als Hausaufgabe
 - Den Gruppen sind schon Termine für die Foto-Praktika zugeordnet (siehe Wiki). In den anderen beiden Praktika bleiben diese Gruppen bestehen, die Praktikumstermine können Sie wählen (wird dann angekündigt).
 - Leider müssen auch „ungünstige“ Zeitfenster wegen der hohen Teilnehmerzahl sowie begrenzter Ressourcen genutzt werden.
 - Falls noch bei keiner Praktikumsgruppe angemeldet: Umgehend über UniWorx anmelden!
 - **Anmeldung** zur Vorlesung, Übung oder zum Praktikum **spätestens heute in der Pause**, da danach die Anmeldung geschlossen wird!

Übungen und Praktika (2)

Programmierberatung:

- Bei Problemen mit aktuellen Programmieraufgaben kann die Programmierberatung im CIP-Pool (Amalienstr. 17) aufgesucht werden:
 - Di., 18:00 – 20:00 Uhr
 - Mi., 18:00 – 20:00 Uhr (KuM-Tutorin)
 - Bei Bedarf weitere regelmäßige Termine

Erwerb der ECTS-Punkte:

- Teilnahme an allen drei Praktika
- Sinnvolle (>50% der Punkte) Abgabe aller Hausaufgaben
- Ein „Freischuss“ erlaubt (weniger als 50% der Punkte bei einem ÜB)
- Erstellung eines abschließenden Berichts zu den Übungen und Praktika
- Beteiligung an der Abschlusspräsentation der Praktikumsergebnisse
- Keine Klausur, keine Benotung!

Zeitplan Übungen und Praktika

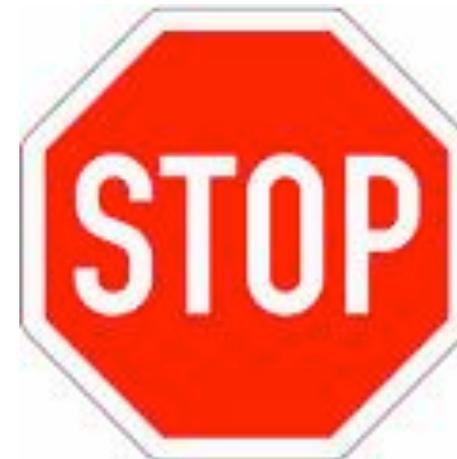
Datum	Praktika	Übungen
19.4. - 23.4.	Praktikumsanmeldung	
26.4. - 30.4.	Foto 1	Einführung Foto / Bildbearbeitung (P)
3.5. - 7.5.	Foto 2	Swing
10.5. - 14.5.	Foto 3	MVC
17.5. - 21.5.	Foto 4	Java Bildbearbeitung
24.5. - 29.5.	Video 1	
31.5. - 4.6.	Video 2	Java Video-Effekte
7.6. - 11.6.	Video 3	
14.6. - 18.6.	Video 4	Einführung Audio (P)
21.6. - 25.6.	Audio 1	Java Sound
28.6. - 2.7.	Audio 2	
5.7. - 9.7.	Audio 3	
12.7. - 16.7.	Abschlußpräsentation	
19.7. - 23.7.	-	

<http://www.die-informatiker.net>

- Im Forum können Probleme mit Aufgaben aktueller Übungsblätter *diskutiert* werden (keine Lösungen posten!)
 - Tutoren/Übungsleiter schauen regelmäßig ins Forum und können dort ggf. fachliche Fragen beantworten
- Bei Fragen zu Korrekturen oder Bewertungen bitte *persönlich* an die Tutoren bzw. die Übungsleitung wenden!

Plagiarismus

- Abschreiben
 - von Kommilitonen
 - von anderen Quellen (z.B. Wikipedia)ist **in keiner Weise** erlaubt!
- Ok:
 - zusammen lernen
 - Links zu guten Quellen austauschen
 - gemeinsames Code Review
 - Tricks verraten
- Nicht ok:
 - Code/Text von anderen kopieren
 - oder zusammen schreiben



Plagiarismus

- Im Zweifelsfall immer vor der Abgabe nachfragen.
- Wenn Sie erwischt werden, wird das Übungsblatt oder auch die gesamte Übung als nicht bestanden gewertet.
- Wir prüfen nach – eventuell auch erst nach Ende der Übungen.

Gliederung

1. Programmierung von Benutzungsschnittstellen (Swing)
 - Grafische Oberflächen, Ereignisgesteuerte Programmierung, MVC-Muster
2. Fototechnik und digitale Bildbearbeitung
 - Grundlagen der digitalen Fototechnik, Bildgestaltung, Bildbearbeitung
3. Film- und Videotechnik und digitale Videobearbeitung
 - Grundlagen der Film- und Videotechnik, Spezialeffekte, Filmschnitt
4. Tontechnik und digitale Tonbearbeitung
 - Grundlagen der Tontechnik, Audiogestaltung, Tonbearbeitung
5. Eingabe- und Ausgabetechnik
 - Technik von Monitoren, Druckern, Eingabegeräten
6. Schnittstellen und Speichermedien
 - Anschlusstechnologien, optische Speicherung (z.B. CD/DVD/Bluray)
7. Digitale Rundfunktechnik
 - DAB, DVB

Begleitende Literatur

Zu dieser Vorlesung gibt es kein passendes Lehrbuch.

(Spezialthemen können meist nur einführend behandelt werden.)

Zu Hintergrundinformationen und einzelnen Themen:

- Rainer Malaka, Andreas Butz, Heinrich Hußmann:
Medieninformatik - eine Einführung
Pearson Studium 2009
- Peter A. Henning: Taschenbuch Multimedia,
2. Auflage, Fachbuchverlag Leipzig/Carl Hanser 2001
- Andreas Holzinger: Basiswissen Multimedia, Band 1: Technik,
Vogel Verlag, 2000

Kapitelspezifische und weiterführende Literatur:

- Bei den einzelnen Kapiteln angegeben

1. Programmierung von Benutzungsschnittstellen

1.1 Modell-Sicht-Paradigma

1.2 Bausteine für grafische Oberflächen

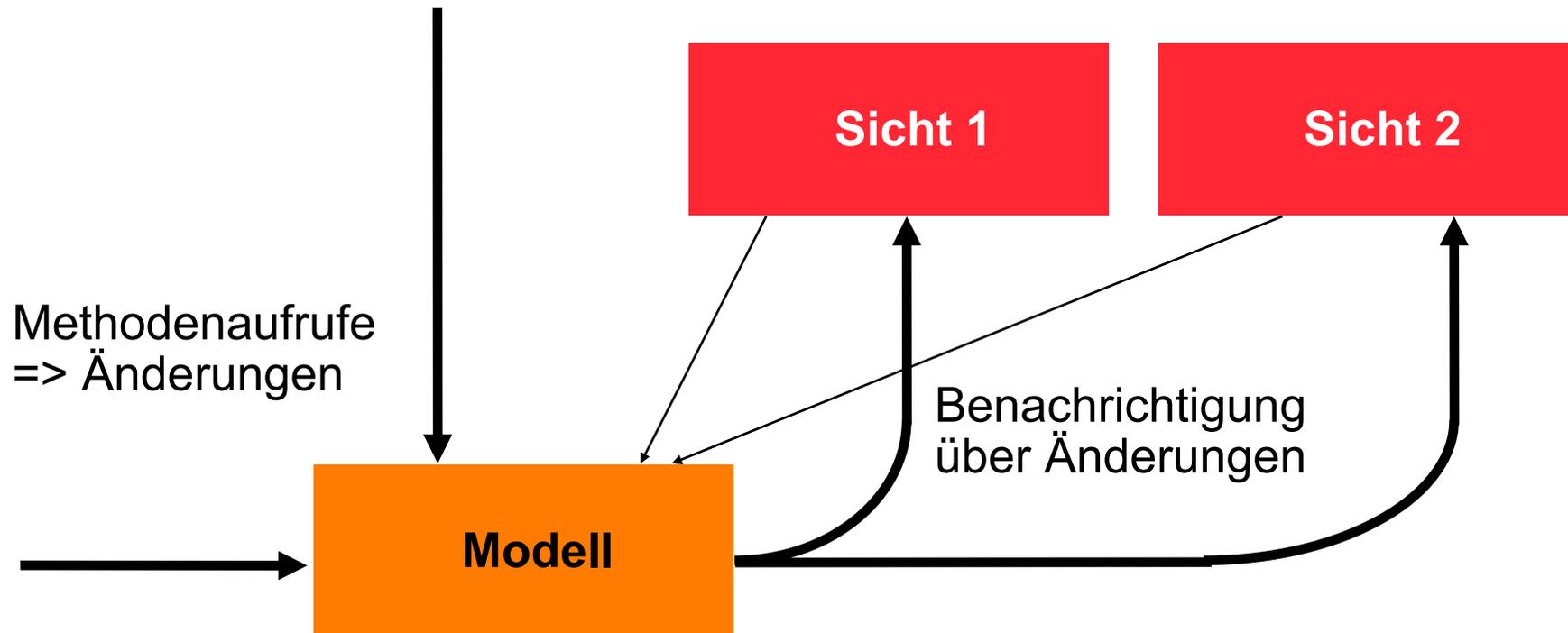
1.3 Ereignisgesteuerte Programme



Benutzungsoberflächen

- Technische Realisierungen:
 - Stapelverarbeitungssprache (*batch control, job control*)
 - Zeilenorientierte interaktive Kommandosprache
 - » Beispiele: Kommandosprachen von MS-DOS, UNIX
 - Skriptsprache
 - Bildschirm- und maskenorientierter Dialog
 - » Beispiele: Dialogoberfläche von MVS, VM/CMS
 - **Graphische Benutzungsoberfläche (*graphical user interface, GUI*)**
 - Multimedia-Benutzungsoberfläche
 - Virtuelle Welt
- Tendenz:
 - Bessere Anpassung an menschliche Kommunikation
 - Weg von sequentieller Organisation hin zu freier Interaktionsgestaltung

Modell und Sicht

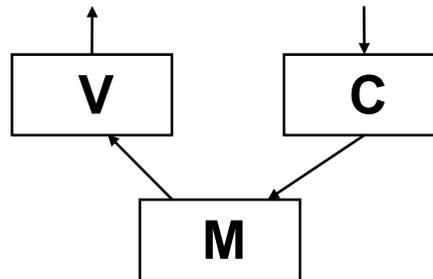


Beispiele: Verschiedene Dokumentenansichten, Statusanzeigen, Verfügbarkeit von Menüpunkten

Frage: *Wie hält man das Modell unabhängig von den einzelnen Sichten darauf ?*

Muster "Observer"

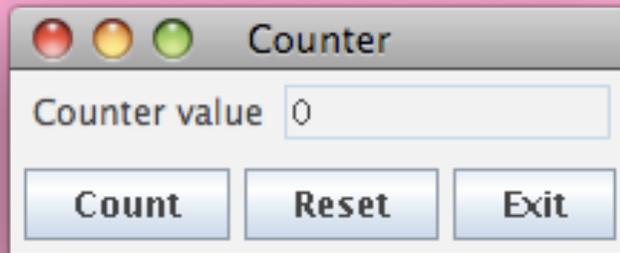
Model-View-Controller-Architektur (MVC)



- Model:
 - Fachliches Modell, weitestgehend unabhängig von Oberfläche
 - Beobachtbar (*observable*)
- View:
 - Repräsentation auf Benutzungsoberfläche
 - Beobachter des Modells
 - Erfragt beim "update" ggf. notwendige Daten beim Modell
- Controller:
 - Modifiziert Werte im Modell
 - Ist an bestimmte Elemente der "View" (z.B. Buttons) gekoppelt
 - Reagiert auf Ereignisse und setzt sie um in Methodenaufrufe

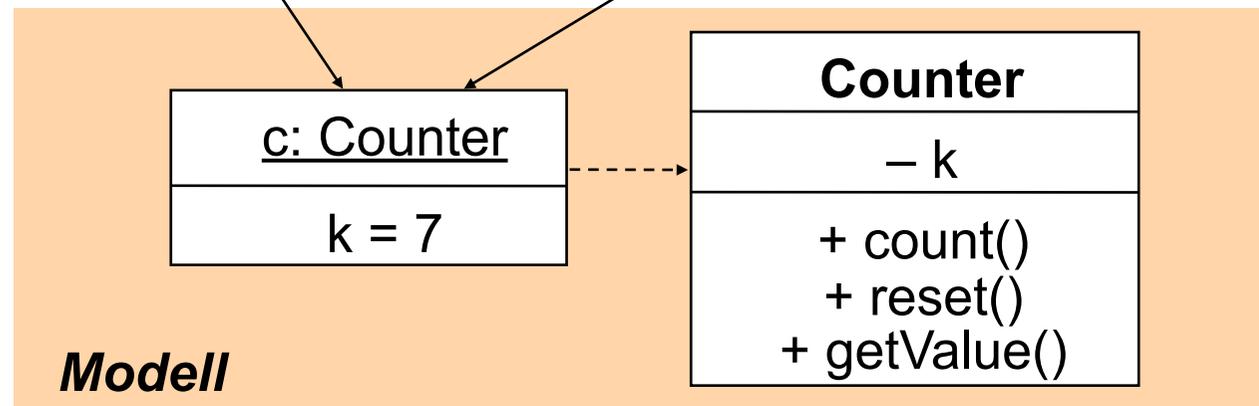
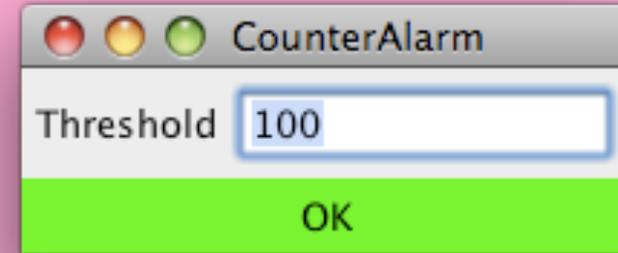
Sichten: Motivierendes Beispiel

Sicht 1



cf: CounterFrame

Sicht 2



Ein Zähler (Beispiel fachliches Modell)

```
class Counter {  
    private int k = 0;  
  
    public void count () {  
        k++;  
    }  
  
    public void reset () {  
        k = 0;  
    }  
  
    public int getValue () {  
        return k;  
    }  
}
```

Beobachtbares Modell (*Model*)

```
class Counter extends Observable {
    private int k = 0;
    public void count () {
        k++;
        setChanged();
        notifyObservers();
    }
    public void reset () {
        k = 0;
        setChanged();
        notifyObservers();
    }
    public int getValue () {
        return k;
    }
}
```

- Das fachliche Modell enthält keinerlei Bezug auf die Benutzungsoberfläche !

java.util.Observable, java.util.Observer

```
public class Observable {  
    public void addObserver (Observer o);  
    public void deleteObserver (Observer o);  
  
    protected void setChanged();  
    public void notifyObservers ();  
    public void notifyObservers (Object arg);  
}  
  
public interface Observer {  
    public void update (Observable o, Object arg);  
}
```

Argumente für notifyObservers():

meist nur Art der Änderung, nicht gesamte Zustandsinformation

Beobachter können normale Methodenaufrufe nutzen, um sich näher zu informieren.

Grundidee der Implementierung von Observable

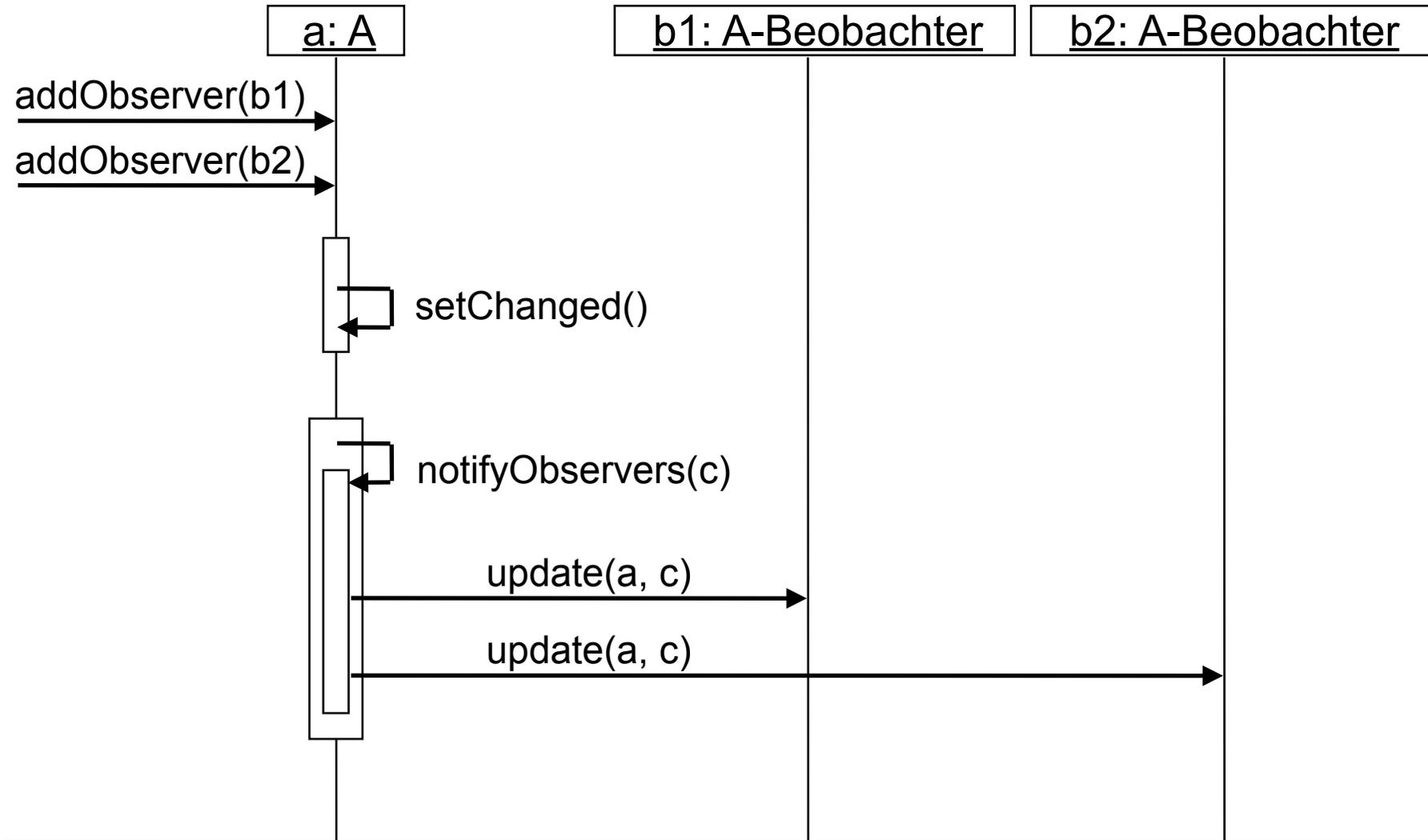
Der Programmierer muß den hier skizzierten Code nicht kennen, sondern nur indirekt anwenden!

```
public class Observable {  
  
    private Collection observed;  
    private boolean changed = false;  
    ...  
  
    public void addObserver (Observer o) { observed.add(o); }  
  
    public void setChanged() { changed = true; }  
  
    public void notifyObservers (Object arg) {  
        Iterator it = observed.iterator();  
        if (!changed) return;  
        while (it.hasNext()) {  
            (it.next()).update(this, arg);  
        }  
    }  
}
```

Beispielablauf

a extends Observable;

b1, b2 implements Observer;



1. Programmierung von Benutzungsschnittstellen

1.1 Modell-Sicht-Paradigma

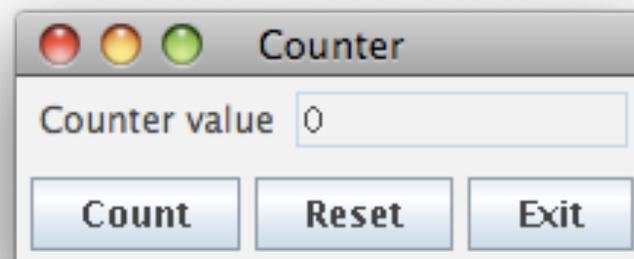
1.2 Bausteine für grafische Oberflächen

1.3 Ereignisgesteuerte Programme



Graphische Benutzungsoberflächen

- 1980: Smalltalk-80-Oberfläche (Xerox)
- 1983/84: Lisa/Macintosh-Oberfläche (Apple)
- 1988: NextStep (Next)
- 1989: OpenLook (Sun)
- 1989: Motif (Open Software Foundation)
- 1987/91: OS/2 Presentation Manager (IBM)
- 1990: Windows 3.0 (Microsoft)
- 1995-2010: Windows 95/NT/98/2000/ME/XP/Vista/7 (Microsoft)
- 1995: Java **Abstract Window Toolkit AWT** (SunSoft)
- 1997: **Swing** Components for Java (SunSoft)

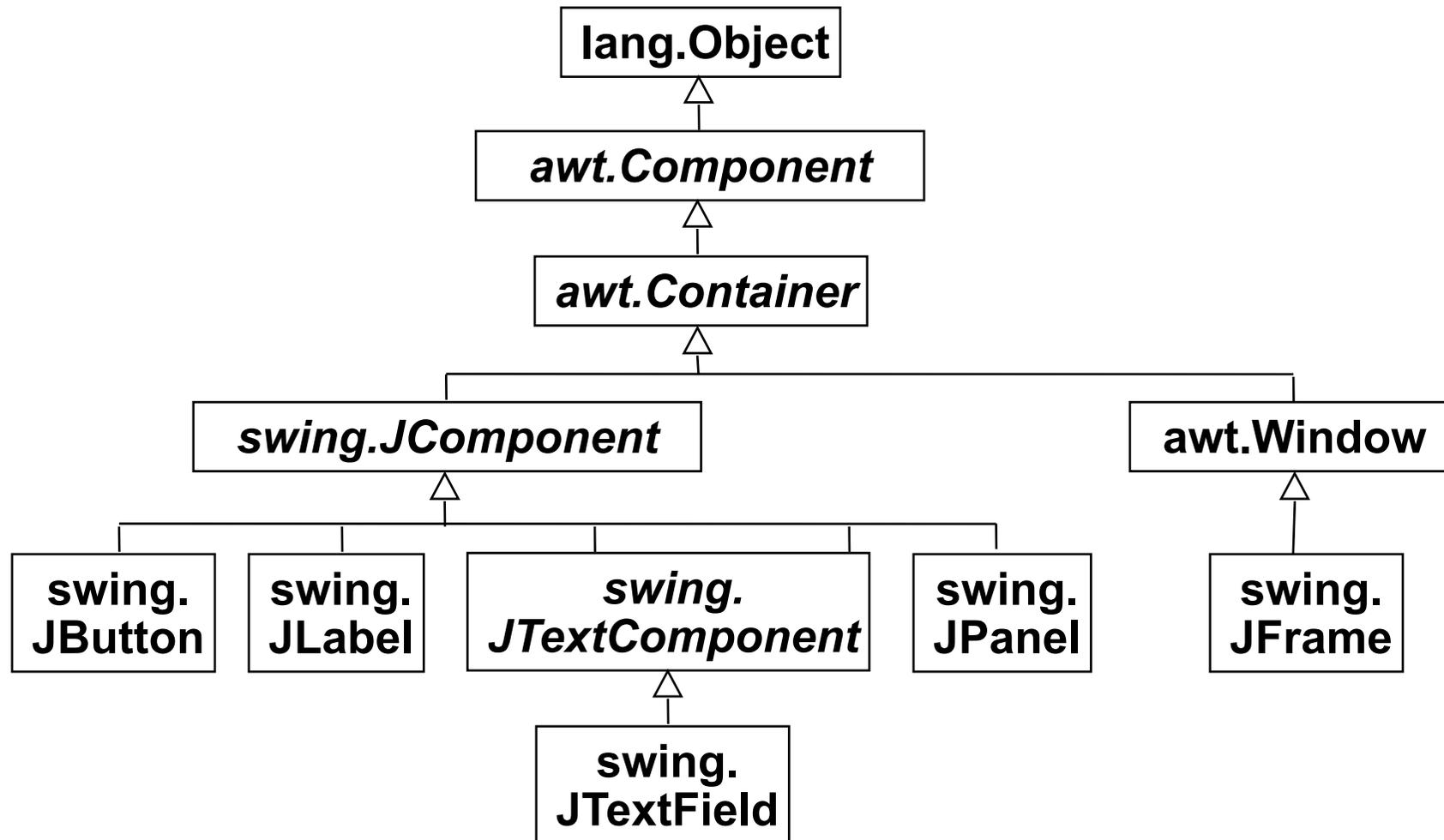


Bibliotheken von AWT und Swing

- Wichtigste AWT-Pakete:
 - **java.awt**: u.a. Grafik, Oberflächenkomponenten, Layout-Manager
 - **java.awt.event**: Ereignisbehandlung
 - Andere Pakete für weitere Spezialzwecke
- Wichtigstes Swing-Paket:
 - **javax.swing**: Oberflächenkomponenten
 - Andere Pakete für Spezialzwecke
- Viele AWT-Klassen werden auch in Swing verwendet!
- Standard-Vorspann:

```
import java.awt.*;  
import java.awt.event.*;  
import javax.swing.*;
```
- (Naiver) Unterschied zwischen AWT- und Swing-Komponenten:
 - AWT: Button, Frame, Menu, ...
 - Swing: JButton, JFrame, JMenu, ...

AWT/Swing-Klassenhierarchie (Ausschnitt)



- Dies ist nur ein sehr kleiner Ausschnitt!
- Präfixe "java." und "javax." hier weggelassen.

Component, Container, Window, JFrame, JPanel

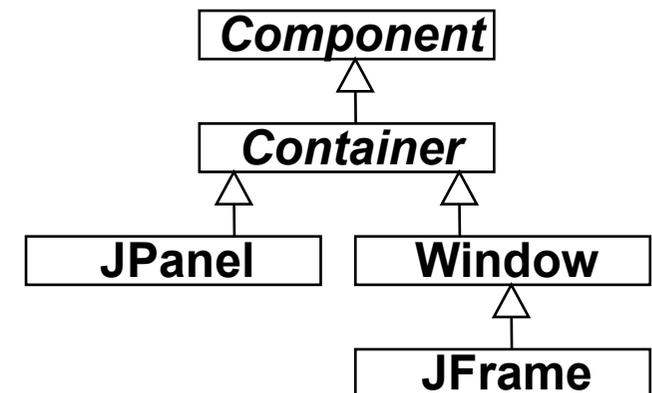
- **awt.Component** (abstrakt):
 - Oberklasse aller Bestandteile der Oberfläche

```
public void setSize (int width, int height);  
public void setVisible (boolean b);
```
- **awt.Container** (abstrakt):
 - Oberklasse aller Komponenten, die andere Komponenten enthalten

```
public void add (Component comp);  
public void setLayout (LayoutManager mgr);
```
- **awt.Window**
 - Fenster ohne Rahmen oder Menüs

```
public void pack (); //Größe anpassen
```
- **swing.JFrame**
 - Größenveränderbares Fenster mit Titel

```
public void setTitle (String title);
```
- **swing.JPanel**
 - Zusammenfassung von Swing-Komponenten



JComponent

- Oberklasse aller in der Swing-Bibliothek neu implementierten, verbesserten Oberflächenkomponenten. Eigenschaften u.a.:
 - Einstellbares "Look-and-Feel" (sh. später)
 - Komponenten kombinierbar und erweiterbar
 - Rahmen für Komponenten

```
void setBorder (Border border) ;
```

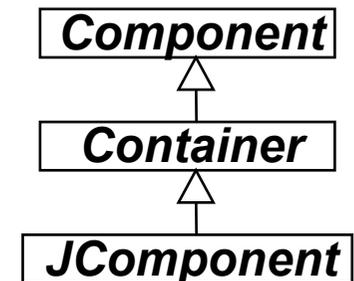
(Border-Objekte mit **BorderFactory** erzeugbar)

- ToolTips -- Kurzbeschreibungen, die auftauchen, wenn der Cursor über der Komponente liegt

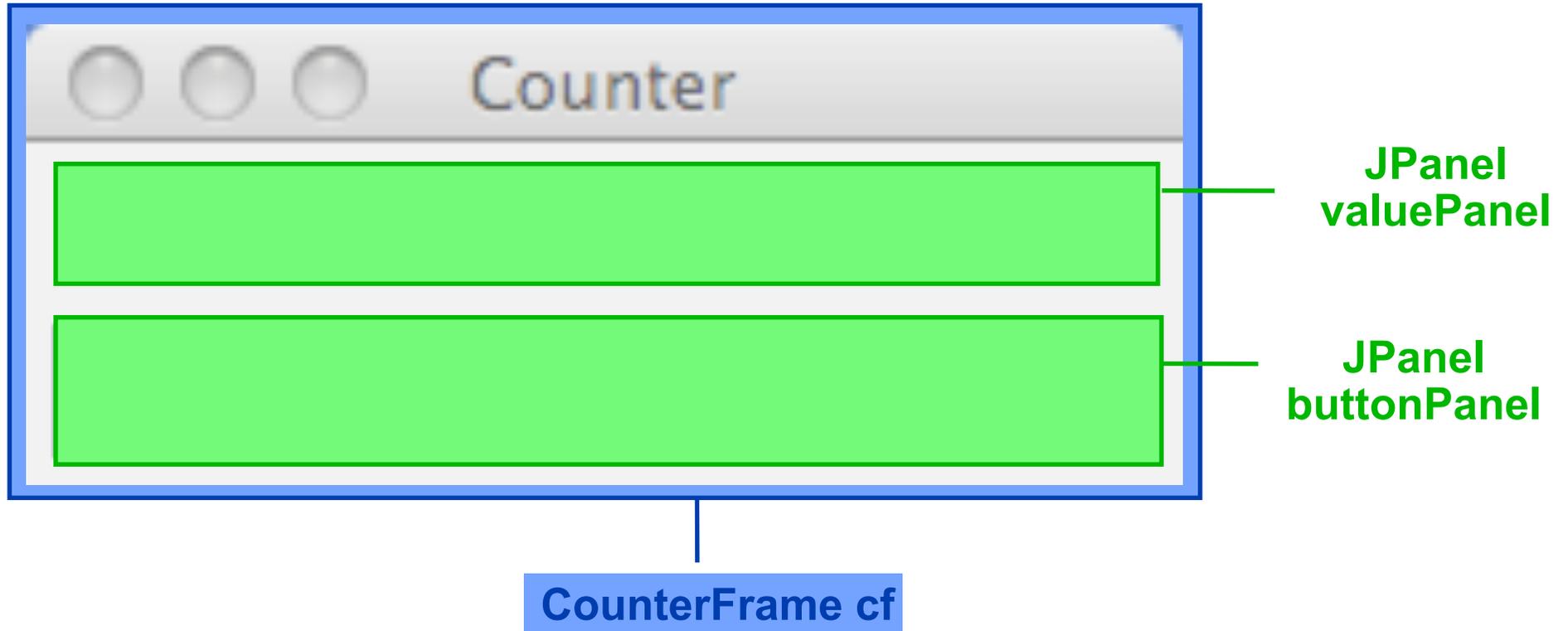
```
void setToolTipText (String text) ;
```

- Automatisches Scrolling

- Beispiele für weitere Unterklassen von JComponent:
 - JList: Auswahlliste
 - JComboBox: "Drop-Down"-Auswahlliste mit Texteingabemöglichkeit
 - JPopupMenu: "Pop-Up"-Menü
 - JFileChooser: Dateiauswahl



Zähler-Beispiel: Grobentwurf der Oberfläche



Die Sicht (View): Gliederung

```
class CounterFrame extends JFrame {
    JPanel valuePanel = new JPanel();

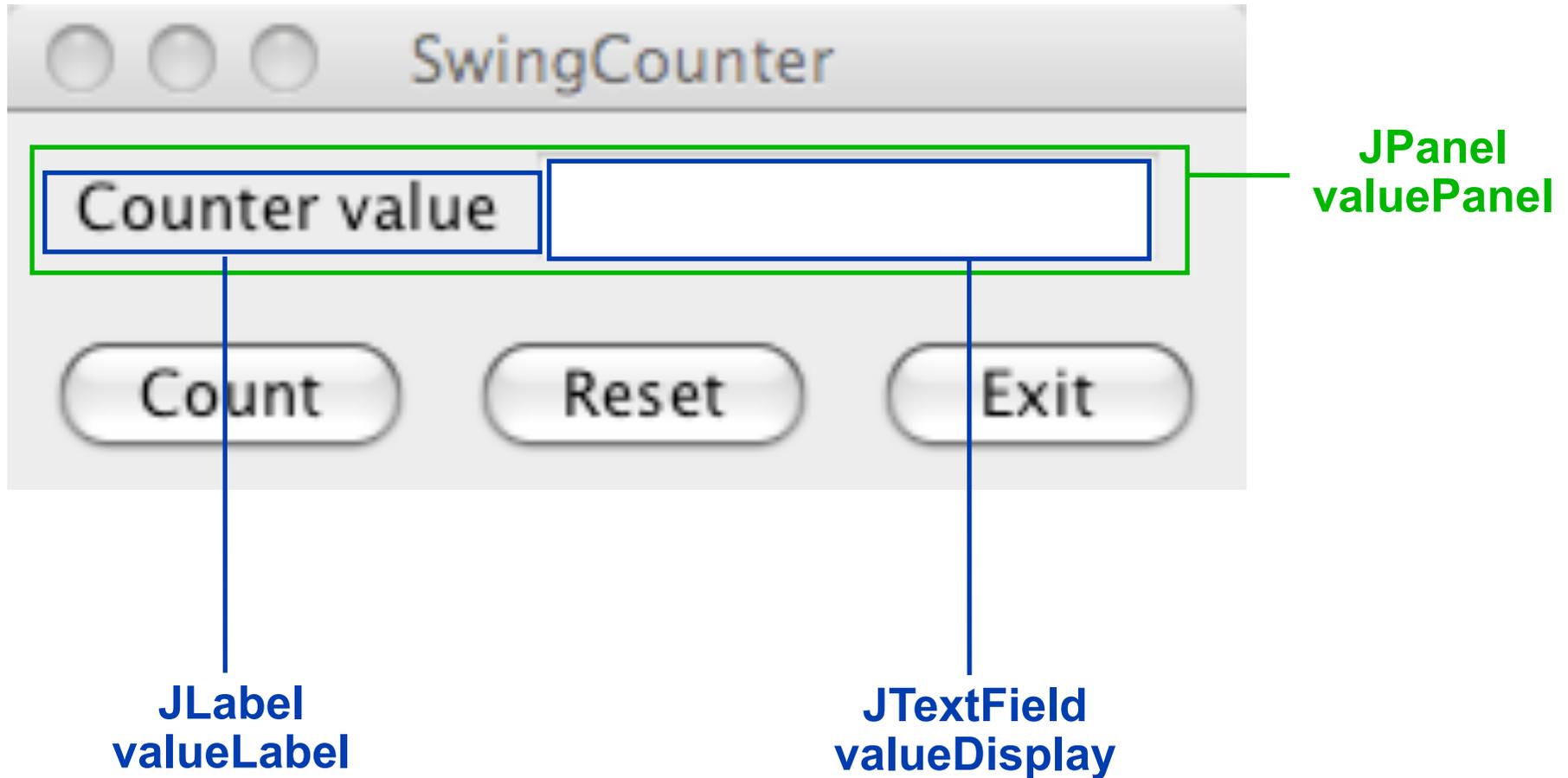
    JPanel buttonPanel = new JPanel();

    public CounterFrame (Counter c) {
        setTitle("SwingCounter");

        add(valuePanel);

        add(buttonPanel);
        pack();
        setVisible(true);
    }
}
```

Zähler-Beispiel: Entwurf der Wertanzeige



JTextComponent, JTextField, JLabel, JButton

- **JTextComponent:**

- Oberklasse von JTextField und JTextArea

- ```
public void setText (String t);
```

- ```
public String getText ();
```

- ```
public void setEditable (boolean b);
```

- **JTextField:**

- Textfeld mit einer Zeile

- ```
public JTextField (int length);
```

- **JLabel:**

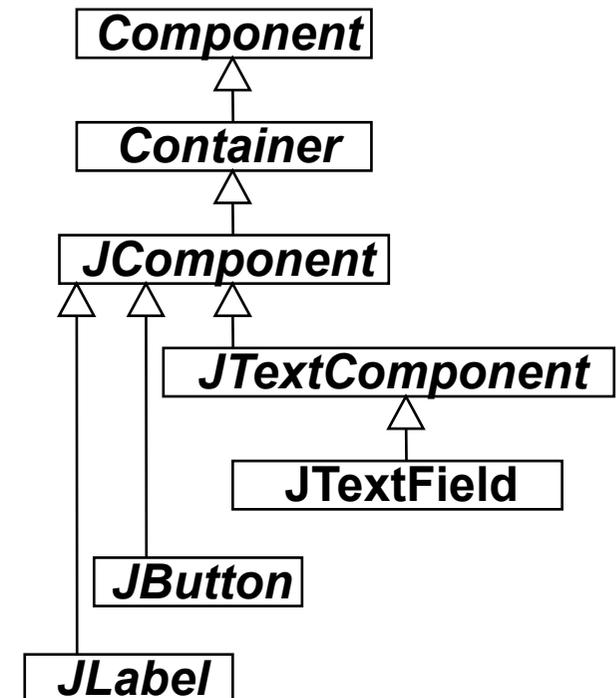
- Einzeiliger unveränderbarer Text

- ```
public JLabel (String text);
```

- **JButton:**

- Druckknopf mit Textbeschriftung

- ```
public JButton (String label);
```



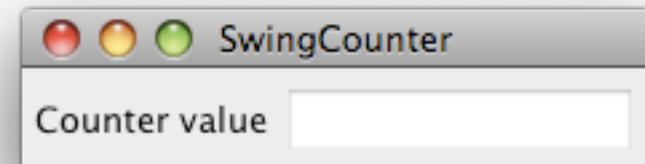
Die Sicht (View): Elemente der Wertanzeige

```
class CounterFrame extends JFrame {
    JPanel valuePanel = new JPanel();
    JTextField valueDisplay = new JTextField(10);
    JPanel buttonPanel = new JPanel();

    public CounterFrame (Counter c) {
        setTitle("SwingCounter");
        valuePanel.add(new JLabel("Counter value"));
        valuePanel.add(valueDisplay);
        valueDisplay.setEditable(false);
        add(valuePanel);

        add(buttonPanel);
        pack();
        setVisible(true);
    }
}
```

Swing2
Swing3

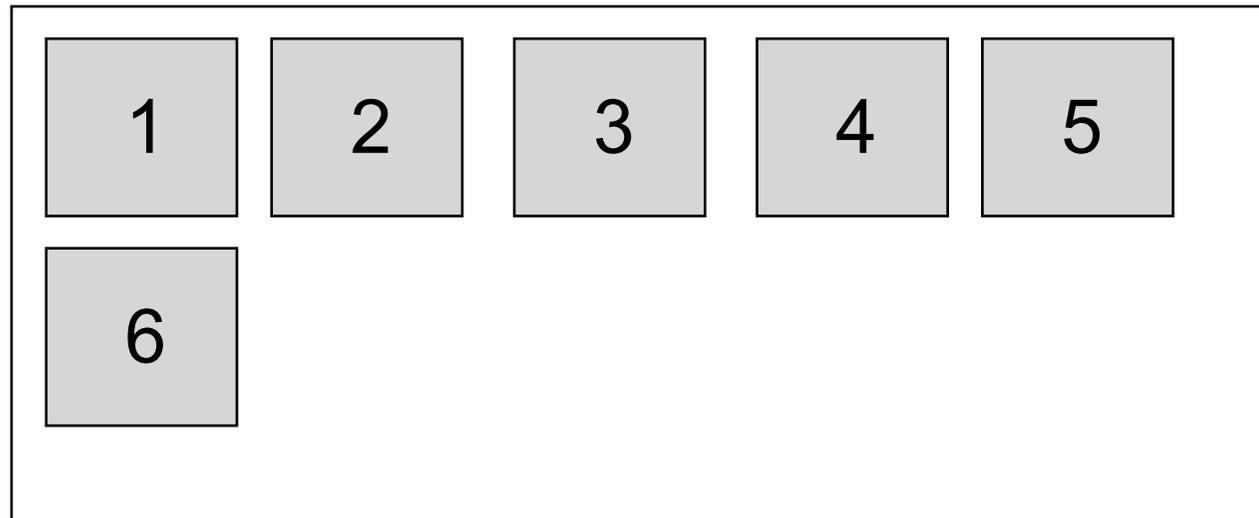


Layout-Manager

- **Definition** Ein *Layout-Manager* ist ein Objekt, das Methoden bereitstellt, um die graphische Repräsentation verschiedener Objekte innerhalb eines Container-Objektes anzuordnen.
- Formal ist `LayoutManager` ein *Interface*, für das viele Implementierungen möglich sind.
- In Java definierte Layout-Manager (Auswahl):
 - `FlowLayout` (`java.awt.FlowLayout`)
 - `BorderLayout` (`java.awt.BorderLayout`)
 - `GridLayout` (`java.awt.GridLayout`)
- In `awt.Component`:
`public void add (Component comp, Object constraints) ;`
erlaubt es, zusätzliche Information (z.B. Orientierung, Zeile/Spalte) an den Layout-Manager zu übergeben

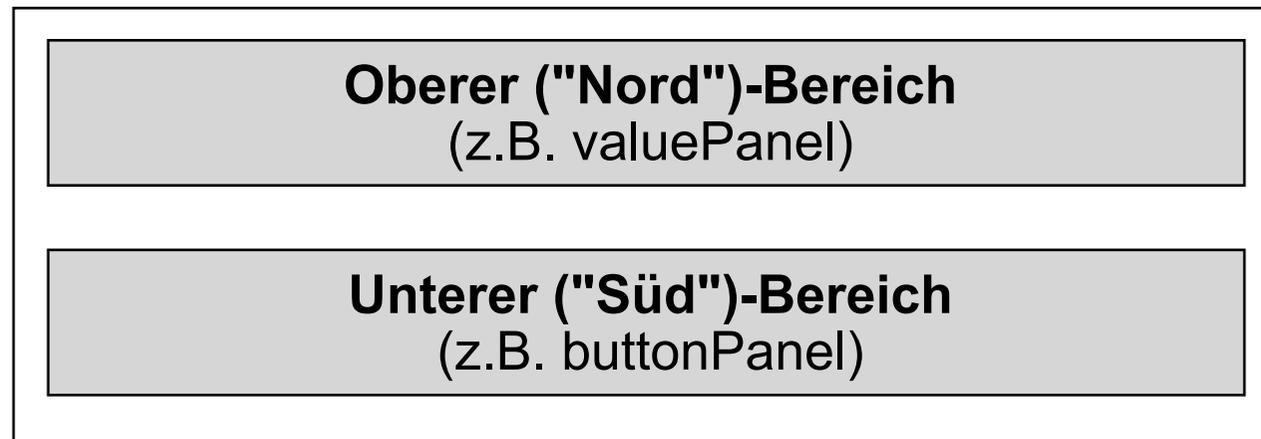
Flow-Layout

- Grundprinzip:
 - Anordnung analog Textfluß:
von links nach rechts und von oben nach unten
- Default für JPanels
 - z.B. in valuePanel und buttonPanel
für Hinzufügen von Labels, Buttons etc.
- Parameter bei Konstruktor: Orientierung auf Zeile, Abstände
- Constraints bei `add`: keine



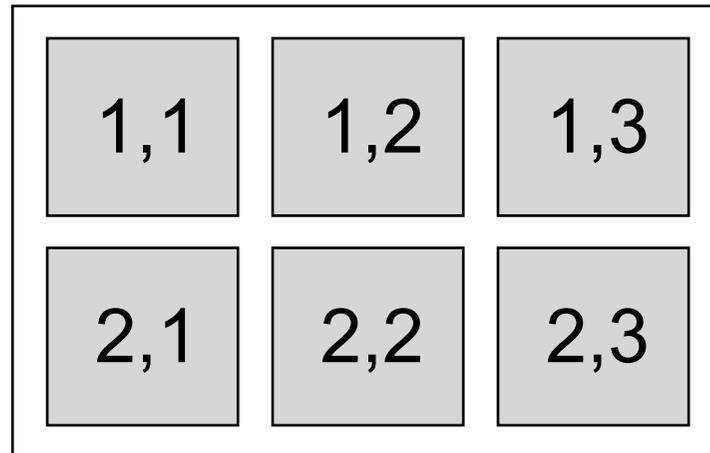
Border-Layout

- Grundprinzip:
 - Orientierung nach den Seiten (N, S, W, O) bzw. Mitte (center)
- Default für Window, JFrame
 - z.B. in CounterFrame für Hinzufügen von valuePanel, buttonPanel
- Parameter bei Konstruktor: Keine
- Constraints bei **add**:
 - **BorderLayout.NORTH, SOUTH, WEST, EAST, CENTER**

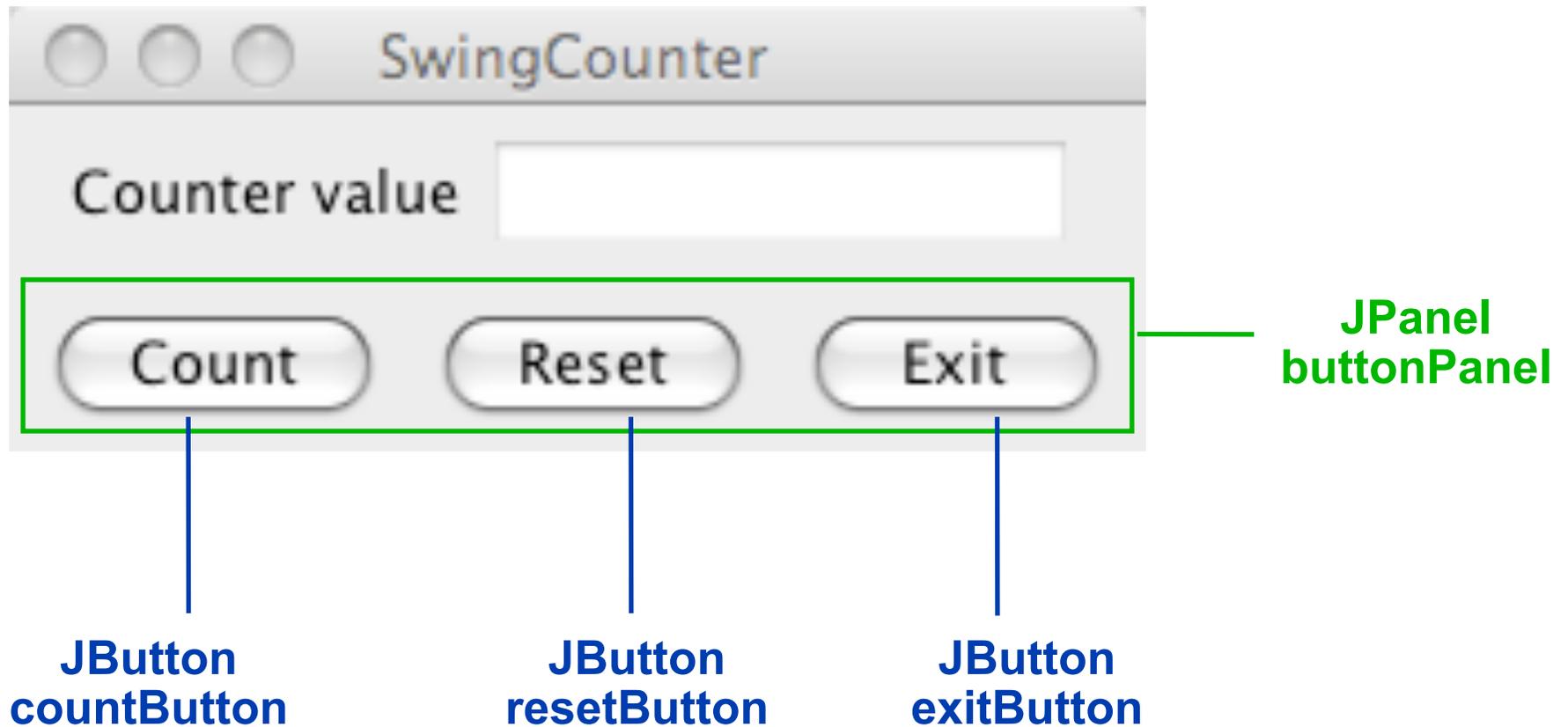


Grid-Layout

- Grundprinzip:
 - Anordnung nach Zeilen und Spalten
- Parameter bei Konstruktor:
 - Abstände, Anzahl Zeilen, Anzahl Spalten
- Constraints bei **add**:
 - Zeilen- und Spaltenindex als int-Zahlen



Zähler-Beispiel: Entwurf der Bedienelemente



Die Sicht (*View*): Bedienelemente

```
class CounterFrame extends JFrame {
    JPanel valuePanel = new JPanel();
    JTextField valueDisplay = new JTextField(10);
    JPanel buttonPanel = new JPanel();
    JButton countButton = new JButton("Count");
    JButton resetButton = new JButton("Reset");
    JButton exitButton = new JButton("Exit");

    public CounterFrame (Counter c) {
        setTitle("SwingCounter");
        valuePanel.add(new JLabel("Counter value"));
        valuePanel.add(valueDisplay);
        valueDisplay.setEditable(false);
        add(valuePanel);
        buttonPanel.add(countButton);
        buttonPanel.add(resetButton);
        buttonPanel.add(exitButton);
        add(buttonPanel);
        pack();
        setVisible(true);
    }
}
```

Die Sicht (*View*): Alle sichtbaren Elemente

```
class CounterFrame extends JFrame {
    JPanel valuePanel = new JPanel();
    JTextField valueDisplay = new JTextField(10);
    JPanel buttonPanel = new JPanel();
    JButton countButton = new JButton("Count");
    JButton resetButton = new JButton("Reset");
    JButton exitButton = new JButton("Exit");

    public CounterFrame (Counter c) {
        setTitle("SwingCounter");
        valuePanel.add(new JLabel("Counter value"));
        valuePanel.add(valueDisplay);
        valueDisplay.setEditable(false);
        add(valuePanel, BorderLayout.NORTH);
        buttonPanel.add(countButton);
        buttonPanel.add(resetButton);
        buttonPanel.add(exitButton);
        add(buttonPanel, BorderLayout.SOUTH);
        pack();
        setVisible(true);
    }
}
```

Zähler-Beispiel: Anbindung Model/View

```
class CounterFrame extends JFrame
    implements Observer {
    ...
    JTextField valueDisplay = new JTextField(10);
    ...

    public CounterFrame (Counter c) {
        ...
        valuePanel.add(valueDisplay);
        valueDisplay.setEditable(false);
        valueDisplay.setText(String.valueOf(c.getValue()));
        ...
        c.addObserver(this);
        pack();
        setVisible(true);
    }

    public void update (Observable o, Object arg) {
        Counter c = (Counter) o;
        valueDisplay.setText(String.valueOf(c.getValue()));
    }
}
```

1. Programmierung von Benutzungsschnittstellen

1.1 Modell-Sicht-Paradigma

1.2 Bausteine für grafische Oberflächen

1.3 Ereignisgesteuerte Programme



Ereignisgesteuerter Programmablauf

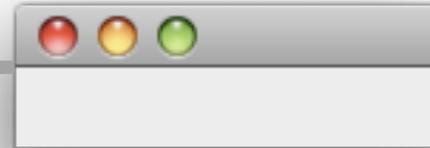
- **Definition** Ein *Ereignis* ist ein Vorgang in der Umwelt des Softwaresystems von vernachlässigbarer Dauer, der für das System von Bedeutung ist.

Eine wichtige Gruppe von Ereignissen sind Benutzerinteraktionen.

- **Beispiele** für Benutzerinteraktions-Ereignisse:
 - Drücken eines Knopfs
 - Auswahl eines Menüpunkts
 - Verändern von Text
 - Zeigen auf ein Gebiet
 - Schließen eines Fensters
 - Verbergen eines Fensters
 - Drücken einer Taste
 - Mausklick

Ereignis-Klassen

- Klassen von Ereignissen in (Java-)Benutzungsoberflächen:
 - WindowEvent
 - ActionEvent
 - MouseEvent
 - KeyEvent, ...
- Bezogen auf Klassen für Oberflächenelemente:
 - Window
 - JFrame
 - JButton
 - JTextField, ...
- Zuordnung (Beispiele):
 - JFrame erzeugt WindowEvent
 - » z.B. bei Betätigung des Schließsymbols (X)
 - JButton erzeugt ActionEvent
 - » bei Betätigung der Schaltfläche



Einfaches Fenster (leer)

```
import java.awt.*;
Import javax.swing.*;

class EventDemoFrame extends JFrame {

    public EventDemoFrame () {
        setTitle("EventDemo");
        setSize(150, 50);
        setVisible(true);
    }
}

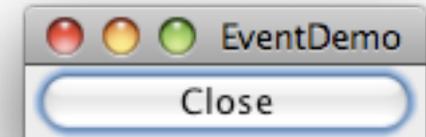
class Event1 {
    public static void main (String[] argv) {
        EventDemoFrame f = new EventDemoFrame();
    }
}
```

Einfaches Fenster mit Schaltfläche (Button)

```
import java.awt.*;
import javax.swing.*;

class EventDemoFrame extends JFrame {

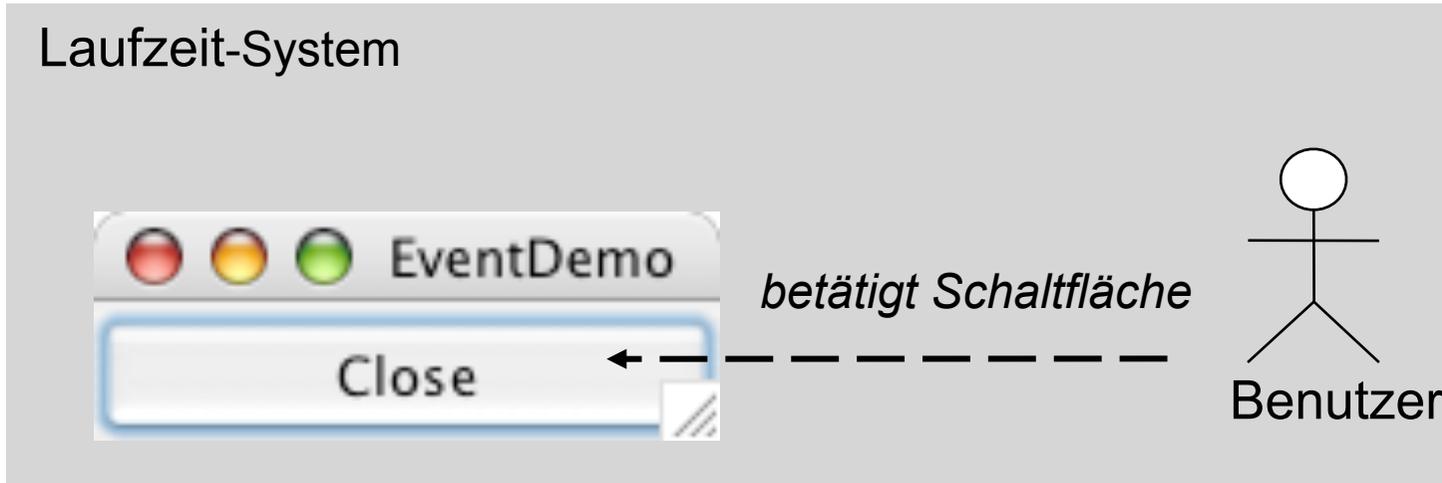
    public EventDemoFrame () {
        setTitle("EventDemo");
        JButton closeButton = new JButton("Close");
        add(closeButton);
        setSize(150, 50);
        setVisible(true);
    }
}
```



```
class Event2 {
    public static void main (String[] argv) {
        EventDemoFrame f = new EventDemoFrame();
    }
}
```

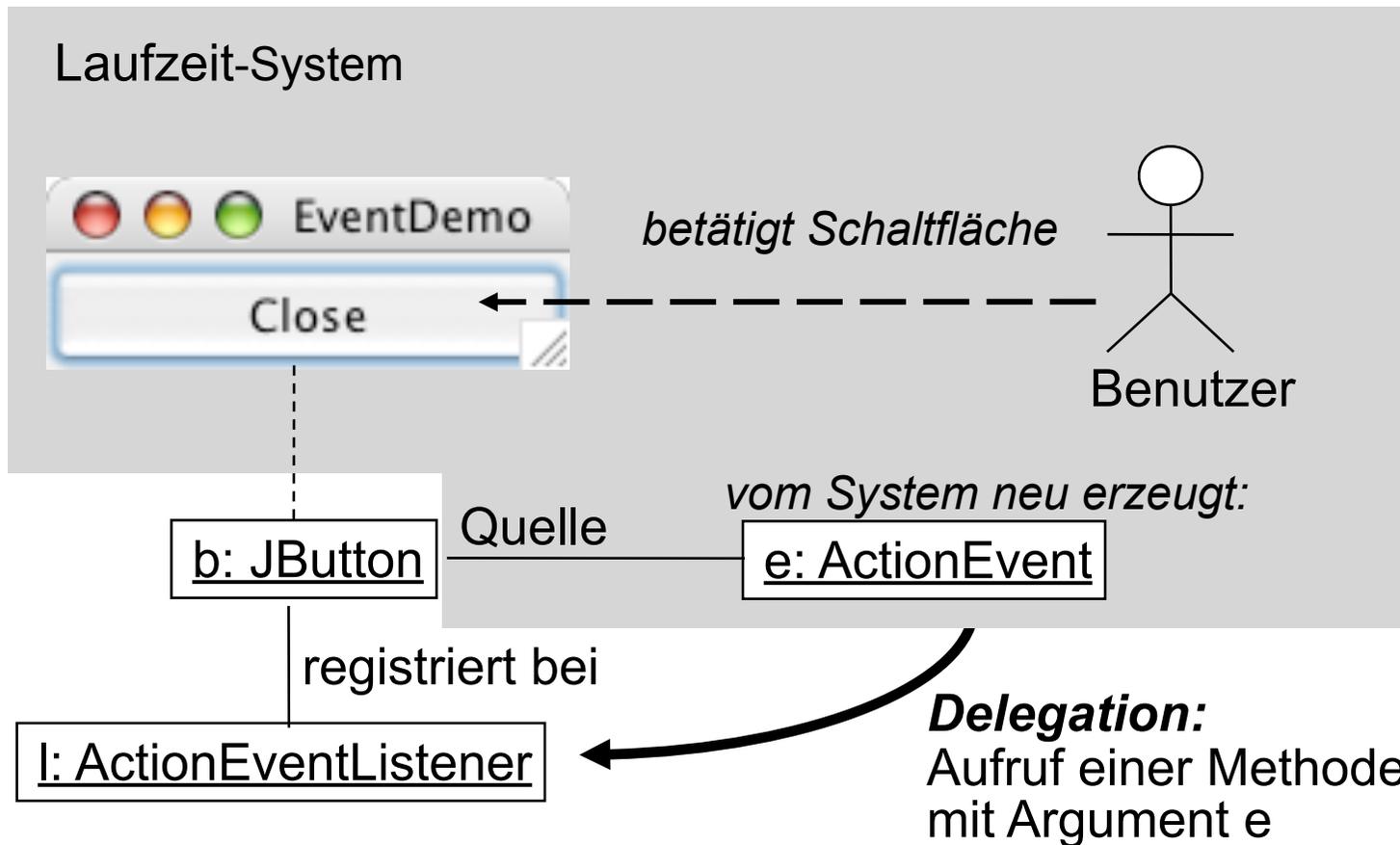
Event2

Ereignis-Delegation (1)



- Reaktion auf ein Ereignis durch Programm:
 - Ereignis wird vom Laufzeitsystem erkannt
- Programm soll von technischen Details entkoppelt werden
 - Beobachter-Prinzip:
 - » Programmteile registrieren sich für bestimmte Ereignisse
 - » Laufzeitsystem sorgt für Aufruf im passenden Moment
- Objekte, die Ereignisse beobachten, heißen bei Java *Listener*.

Ereignis-Delegation (2)



Registrierung für Listener

- In javax.swing.JButton (erbt von javax.swing.AbstractButton):

```
public class JButton ... {  
    public void addActionListener(ActionListener l)  
}
```

- java.awt.event.ActionListener ist eine Schnittstelle:

```
public interface ActionListener  
    extends EventListener{  
    public void actionPerformed(ActionEvent e)  
}
```

- Vergleich mit Observer-Muster:
 - Button bietet einen "Observable"-Mechanismus
 - Listener ist eine "Observer"-Schnittstelle

java.awt.event.ActionEvent

```
public class ActionEvent extends AWTEvent {  
    ...  
    // Konstruktor wird vom System aufgerufen  
    public ActionEvent (...);  
  
    // Abfragemöglichkeiten  
    public Object getSource ();  
    public String getActionCommand();  
    public long getWhen();  
    ...  
}
```

Listener für Ereignis "Schaltfläche gedrückt"

```
import java.awt.*;
import java.awt.event.*;

class CloseEventHandler implements ActionListener {

    public void actionPerformed(ActionEvent event) {
        System.exit(0);
    }

}

// System.exit(0) beendet das laufende Programm
```

Programm mit Schaltfläche "Schließen"

```
class Event3 {  
    public static void main (String[] argv) {  
        EventDemoFrame f = new EventDemoFrame();  
    }  
}
```

```
class EventDemoFrame extends JFrame {  
  
    public EventDemoFrame () {  
        setTitle("EventDemo");  
        JButton closeButton = new JButton("Close");  
        getContentPane().add(closeButton);  
        closeButton.addActionListener(new CloseEventHandler());  
        setSize(150, 50);  
        setVisible(true);    }  
}
```

```
class CloseEventHandler implements ActionListener {  
    public void actionPerformed(ActionEvent event) {  
        System.exit(0);  
    }  
}
```

Event3

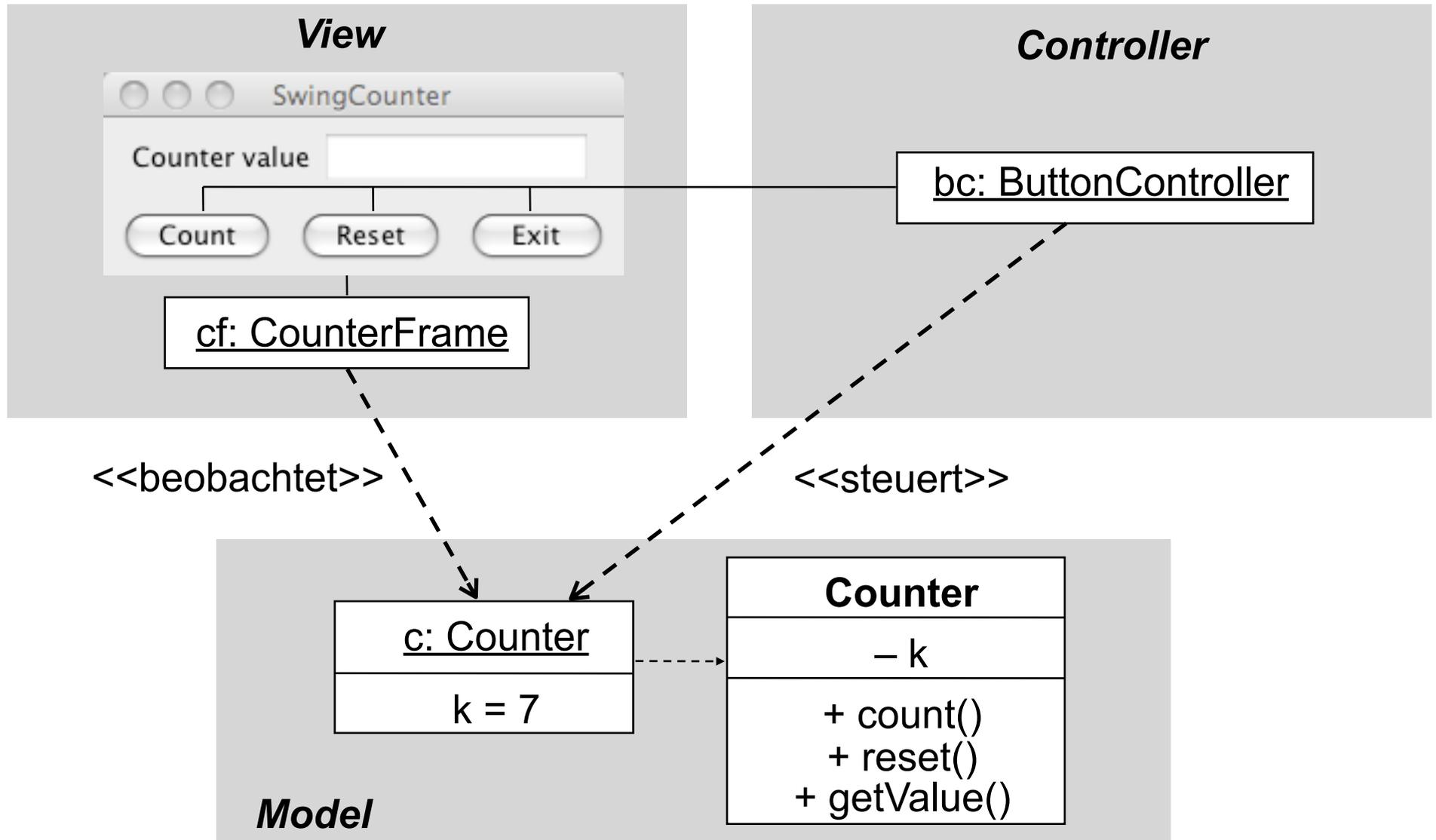
Vereinfachung 1: Innere Klasse

```
class EventDemoFrame extends JFrame {  
  
    class CloseEventHandler implements ActionListener {  
        public void actionPerformed(ActionEvent event) {  
            System.exit(0);  
        }  
    }  
  
    public EventDemoFrame () {  
        setTitle("EventDemo");  
        JButton closeButton = new JButton("Close");  
        getContentPane().add(closeButton);  
        closeButton.addActionListener(new CloseEventHandler());  
        setSize(150, 50);  
        setVisible(true);    }  
  
}
```

Vereinfachung 2: *Anonyme* innere Klasse

```
class EventDemoFrame extends JFrame {  
  
    public EventDemoFrame () {  
        setTitle("EventDemo");  
        JButton closeButton = new JButton("Close");  
        getContentPane().add(closeButton);  
        closeButton.addActionListener(new ActionListener() {  
            public void actionPerformed(ActionEvent event) {  
                System.exit(0);  
            }  
        });  
    };  
    setSize(150, 50);  
    setVisible(true);    }  
  
}
```

Model-View-Controller-Architektur



java.awt.event.ActionEvent, ActionListener

```
public class ActionEvent extends AWTEvent {
    ...
    // Konstruktor wird vom System aufgerufen

    public Object getSource ()
    public String getActionCommand()
    ...
}

public interface ActionListener
    extends EventListener {
    public void actionPerformed (ActionEvent ev);
}
```

Wieviele Controller?

- Möglichkeit 1: Ein Controller für mehrere Buttons (sh.nächste Folie)
 - Speicherplatzersparnis
 - Aber: Wie unterscheiden wir, woher die Ereignisse kommen?
 - Z.B. über `getSource()` und Abfrage auf Identität mit Button-Objekt
 - Z.B. über `getActionCommand()` und Abfrage auf Kommando-String
 - » Default: Kommando-String aus Button-Beschriftung
 - » Kann gesetzt werden mit `setActionCommand()`
 - » Standard-Kommando-String gleich Button-Label – nicht ungefährlich...
- Möglichkeit 2: Viele Controller-Objekte
 - Direkte Angabe von Eventhandlern
 - » am knappsten über anonyme innere Klasse

Die Steuerung (*Controller*)

```
class ButtonController implements ActionListener {  
  
    Counter myCounter;  
  
    public void actionPerformed (ActionEvent event) {  
        String cmd = event.getActionCommand();  
        if (cmd.equals("Count"))  
            myCounter.count();  
        if (cmd.equals("Reset"))  
            myCounter.reset();  
        if (cmd.equals("Exit"))  
            System.exit(0);  
    }  
  
    public ButtonController (Counter c) {  
        myCounter = c;  
    }  
}
```

Zähler-Beispiel: Anbindung des Controllers

```
class CounterFrame extends JFrame {
    ...
    JPanel buttonPanel = new JPanel();
    JButton countButton = new JButton("Count");
    JButton resetButton = new JButton("Reset");
    JButton exitButton = new JButton("Exit");

    public CounterFrame (Counter c) {
        ...
        ButtonController bc = new ButtonController(c);
        countButton.setActionCommand("Count");
        countButton.addActionListener(bc);
        buttonPanel.add(countButton);
        resetButton.setActionCommand("Reset");
        resetButton.addActionListener(bc);
        buttonPanel.add(resetButton);
        exitButton.setActionCommand("Exit");
        exitButton.addActionListener(bc);
        buttonPanel.add(exitButton);
        ...
    }
}
```

Alternative: Controller als anonyme innere Klasse

```
class CounterFrame extends JFrame { ...
    private Counter ctr;
    ...

    public CounterFrame (Counter c) {
        setTitle("Counter");
        ctr = c;
        ...
        countButton.addActionListener(new ActionListener() {
            public void actionPerformed (ActionEvent event) {
                ctr.count();
            }
        });
    }
    ...
}
```

Controller und View bilden eine Einheit: In der Praxis weit verbreitet.

Alles zusammen: CounterFrame (1)

```
class CounterFrame extends JFrame implements Observer {
    JPanel valuePanel = new JPanel();
    JTextField valueDisplay = new JTextField(10);
    JPanel buttonPanel = new JPanel();
    JButton countButton = new JButton("Count");
    JButton resetButton = new JButton("Reset");
    JButton exitButton = new JButton("Exit");

    public CounterFrame (Counter c) {
        setTitle("SwingCounter");
        valuePanel.add(new JLabel("Counter value"));
        valuePanel.add(valueDisplay);
        valueDisplay.setEditable(false);
        valueDisplay.setText(String.valueOf(c.getValue()));
        getContentPane().add(valuePanel, BorderLayout.NORTH);
        ButtonController bc = new ButtonController(c);
        countButton.setActionCommand("Count");
        countButton.addActionListener(bc);
        buttonPanel.add(countButton);
        resetButton.setActionCommand("Reset");
        resetButton.addActionListener(bc);
        buttonPanel.add(resetButton);
        exitButton.setActionCommand("Exit");
        exitButton.addActionListener(bc);
        buttonPanel.add(exitButton);
        getContentPane().add(buttonPanel, BorderLayout.SOUTH);
    }
}
```

Alles zusammen: CounterFrame (2)

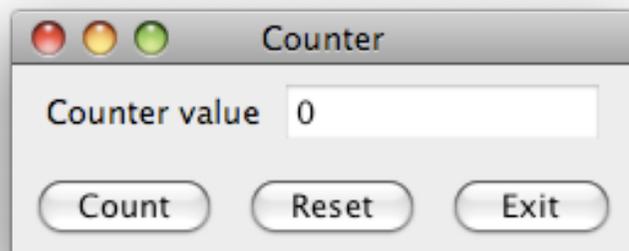
...

```
    c.addObserver(this);
    pack();
    setVisible(true);
    setDefaultCloseOperation(EXIT_ON_CLOSE);
}

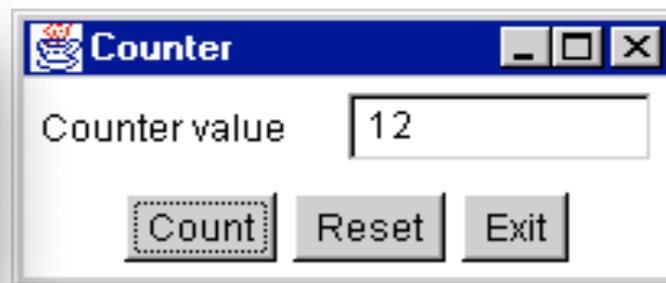
public void update (Observable o, Object arg) {
    Counter c = (Counter) o;
    valueDisplay.setText(String.valueOf(c.getValue()));
}
}
```

"Look-and-Feel"

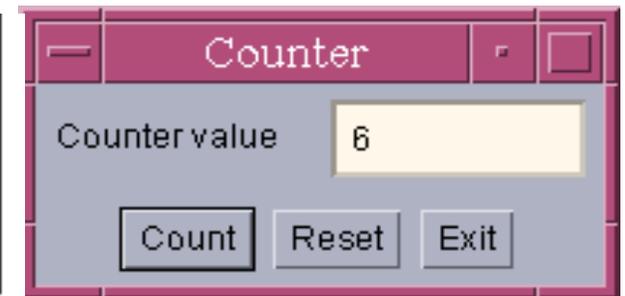
- Jede Plattform hat ihre speziellen Regeln für z.B.:
 - Gestaltung der Elemente von "Frames" (Titelbalken etc.)
 - Standard-Bedienelemente zum Bewegen, Schließen, etc. von "Frames"
- Einstellbares Look-and-Feel: Standard-Java oder plattformspezifisch
- Dasselbe Java-Programm mit verschiedenen "Look-and-Feels":



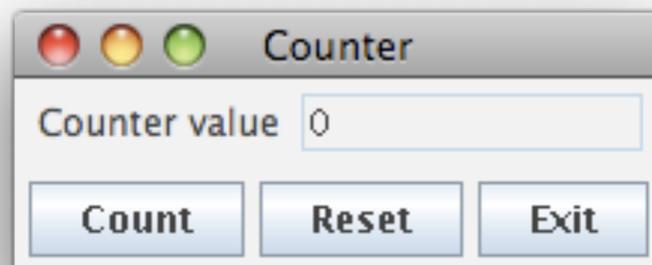
Macintosh (MacOS X)



Windows



Solaris (CDE)



Plattformunabhängiges
Java-Look-and-Feel (auf MacOS)