

Medientechnik

Andreas Butz

Ludwig-Maximilians-Universität München

Sommersemester 2008

Vorlesungskonzept: Heinrich Hußmann

1b. Ein-/Ausgabebetonte Programmierung

B1.1 Mensch-Maschine-Kommunikation



B1.2 Modell-Sicht-Paradigma

B1.3 Bausteine für grafische Oberflächen

B1.4 Ereignisgesteuerte Programme

Mensch und Maschine

Sinnesorgane
"Fürs Überleben ausgestattet"
Soziales Wesen
Assoziatives Denken



```
010100011000100010  
010000111001111001  
001100111000010100  
111001001001001000
```

...

- "Sinnesorgane" für Computer?
- Soziales Verhalten?
- Anpassung an menschliches Denken, Fühlen und Tun

„Ohren und Mund“ des Computers

- Konventionell:
 - Eingabe: Tastatur, Zeigeegeräte
 - Ausgabe: Bildschirmanzeige, Drucker
- Multimedial:
 - Eingabe: Kameras, Scanner, Mikrofone, Musikinstrumente, ...
 - Ausgabe: Fernsehgeräte, Lautsprecher, ...
- „*Ubiquitous Computing*“
(„Allgegenwärtige“ Rechnerunterstützung):
 - Eingabe: Alltagsgegenstände, Anwesenheit, Bewegungen, ...
 - Ausgabe: Alltagsgegenstände, Beleuchtung, beliebige Geräte, ...
 - Zusätzlich Verwendung diverser mobiler Geräte
(z.B. PDA, Mobiltelefon, digitaler Musikspieler)

Kommunikationsqualität bei konventioneller Ein-/Ausgabe

- Ausgabe („Mund“):
 - Bildschirmanzeige erlaubt hochwertige Bilder (Farbtiefe besser als wahrnehmbar, Bildfrequenz angemessen, Auflösung akzeptabel)
 - Tonausgabe mit gleicher Qualität möglich wie bei Musikwiedergabegeräten
- Eingabe („Ohren“):
 - Tastatur/Maus wesentlich langsamer als vom Menschen generierte Informationsraten
 - Spracherkennung, Handschrifterkennung etc. qualitativ noch unzureichend
- Computer können sich gut ausdrücken, aber dem Menschen nur sehr schlecht zuhören!

(nach: Chris Crawford, The Art of Interactive Design)
- Konsequenzen:
 - Eingabekanäle auf guten Durchsatz optimieren
 - Ausgabekanäle auf Übersichtlichkeit optimieren

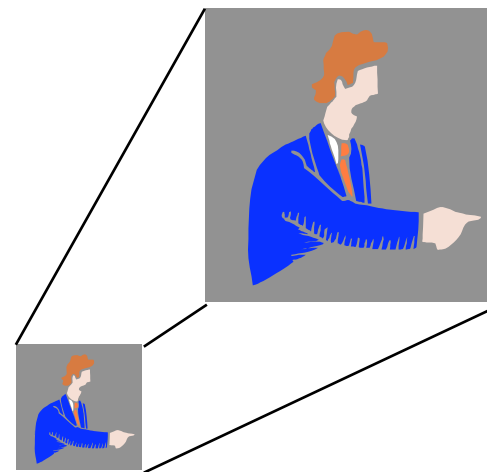
Software-Ergonomie (*usability*)

- Grenzgebiet zur Psychologie
- Gestaltung von Software unter dem Aspekt der Benutzbarkeit



Angemessen
zur Lösung der Aufgabe

Flexibel für
verschiedene
Arbeitsweisen
und Zugänge



Erlaubt Weiterentwicklung:
Lernen während der
Arbeit

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

B1. Ein-/Ausgabebetonte Programmierung

B1.1 Mensch-Maschine-Kommunikation

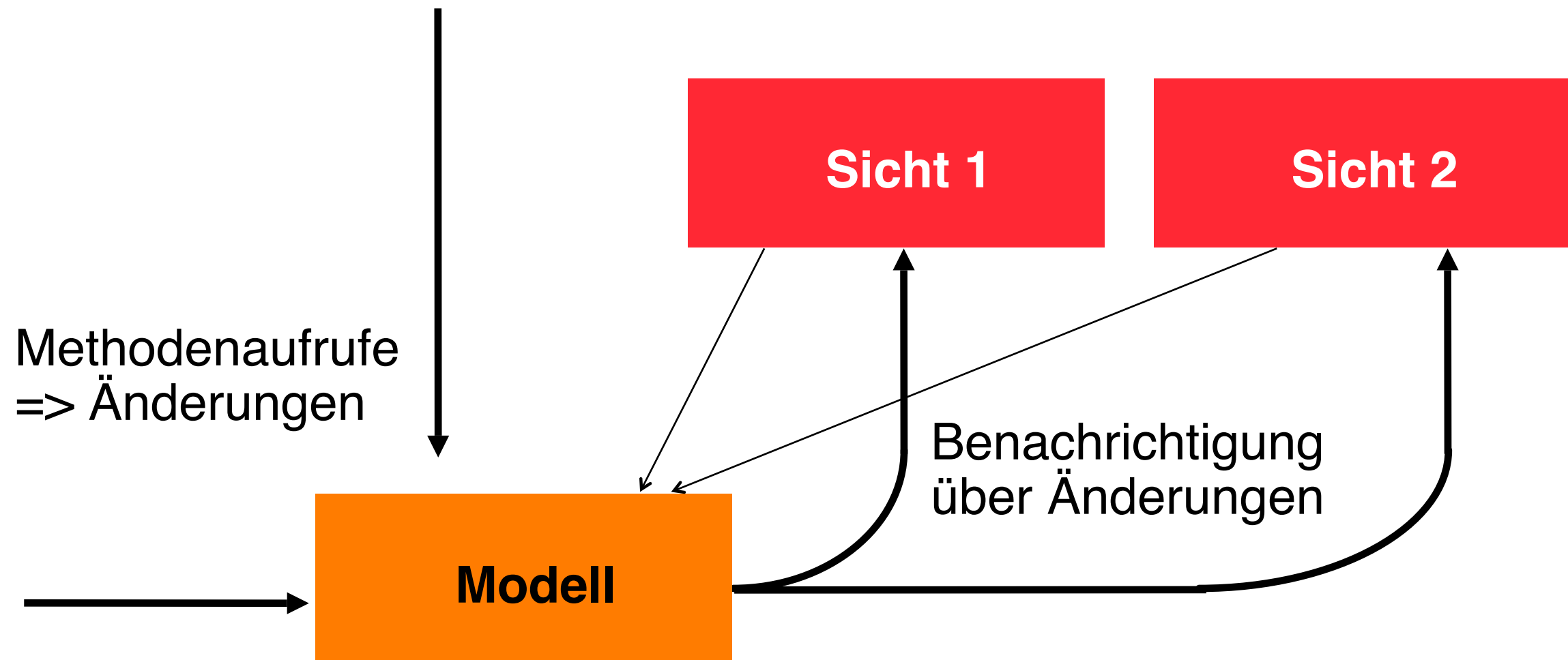
B1.2 Modell-Sicht-Paradigma



B1.3 Bausteine für grafische Oberflächen

B1.4 Ereignisgesteuerte Programme

Modell und Sicht

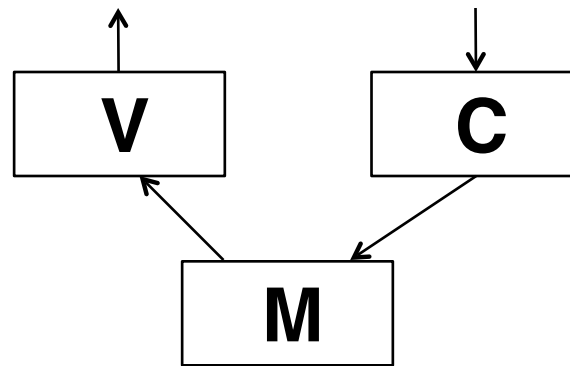


Beispiele: Verschiedene Dokumentenansichten, Statusanzeigen, Verfügbarkeit von Menüpunkten

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

Muster "Observer"

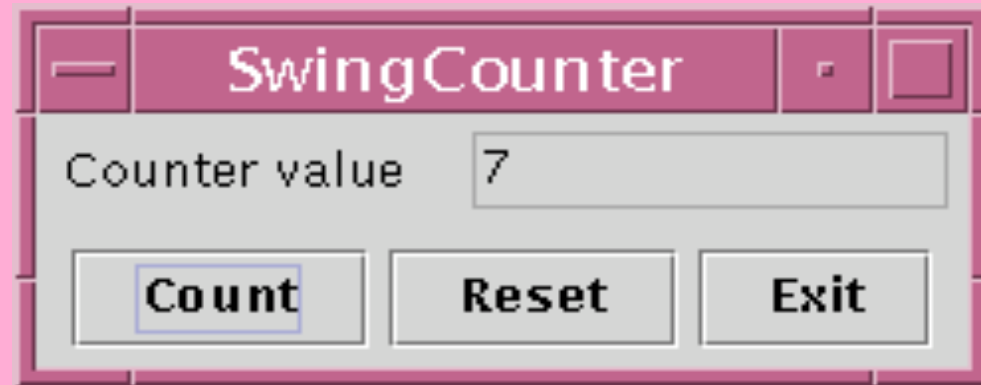
Model-View-Controller-Architektur (MVC)



- 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

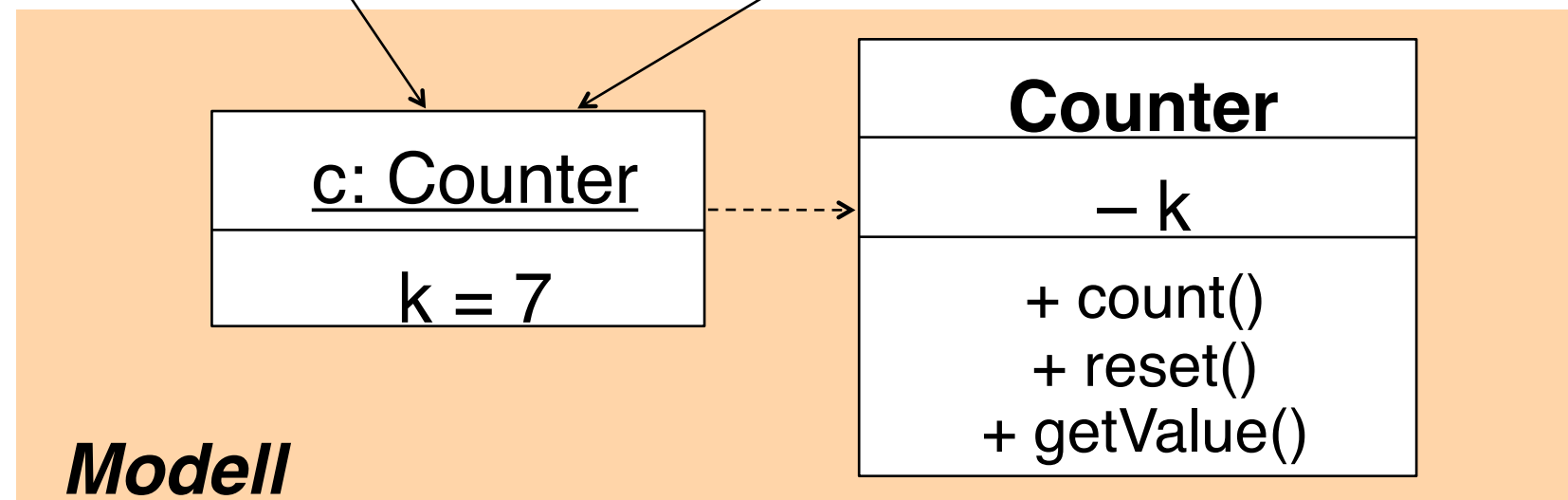
Sichten: Motivierendes Beispiel

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():

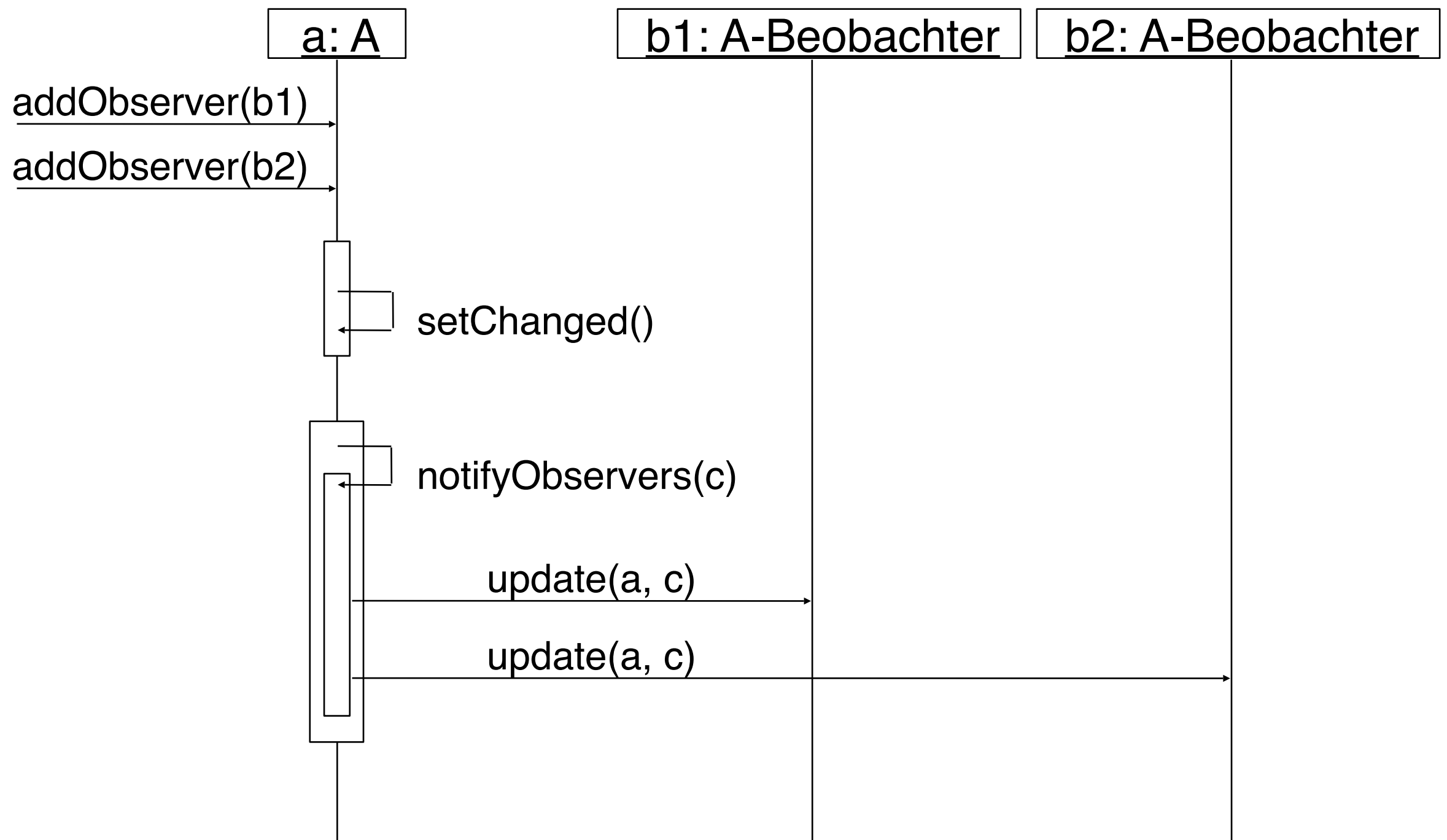
meist nur Art der Änderung, nicht gesamte Zustandsinformation

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

Beispielablauf

a extends Observable;

b1, b2 implements Observer;



B1. Ein-/Ausgabebetonte Programmierung

B1.1 Mensch-Maschine-Kommunikation

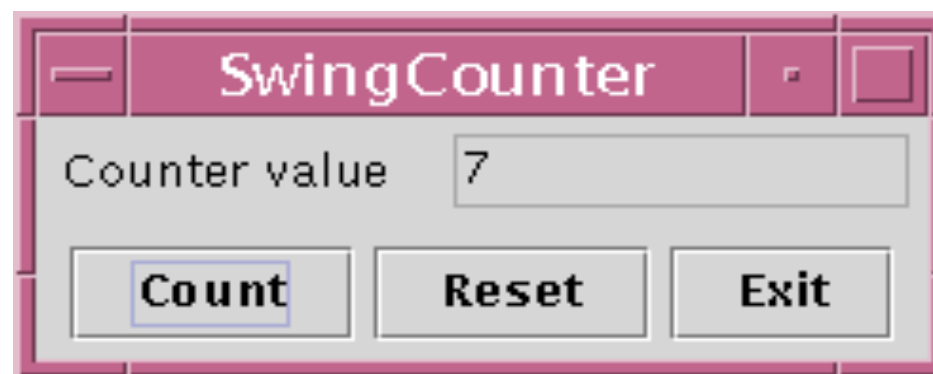
B1.2 Modell-Sicht-Paradigma

B1.3 Bausteine für grafische Oberflächen 

B1.4 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)
- 1995-2001: Windows 95/NT/98/2000/ME/XP (Microsoft)
- 1995: Java **Abstract Window Toolkit AWT** (SunSoft)
- 1997: **Swing** Components for Java (SunSoft)

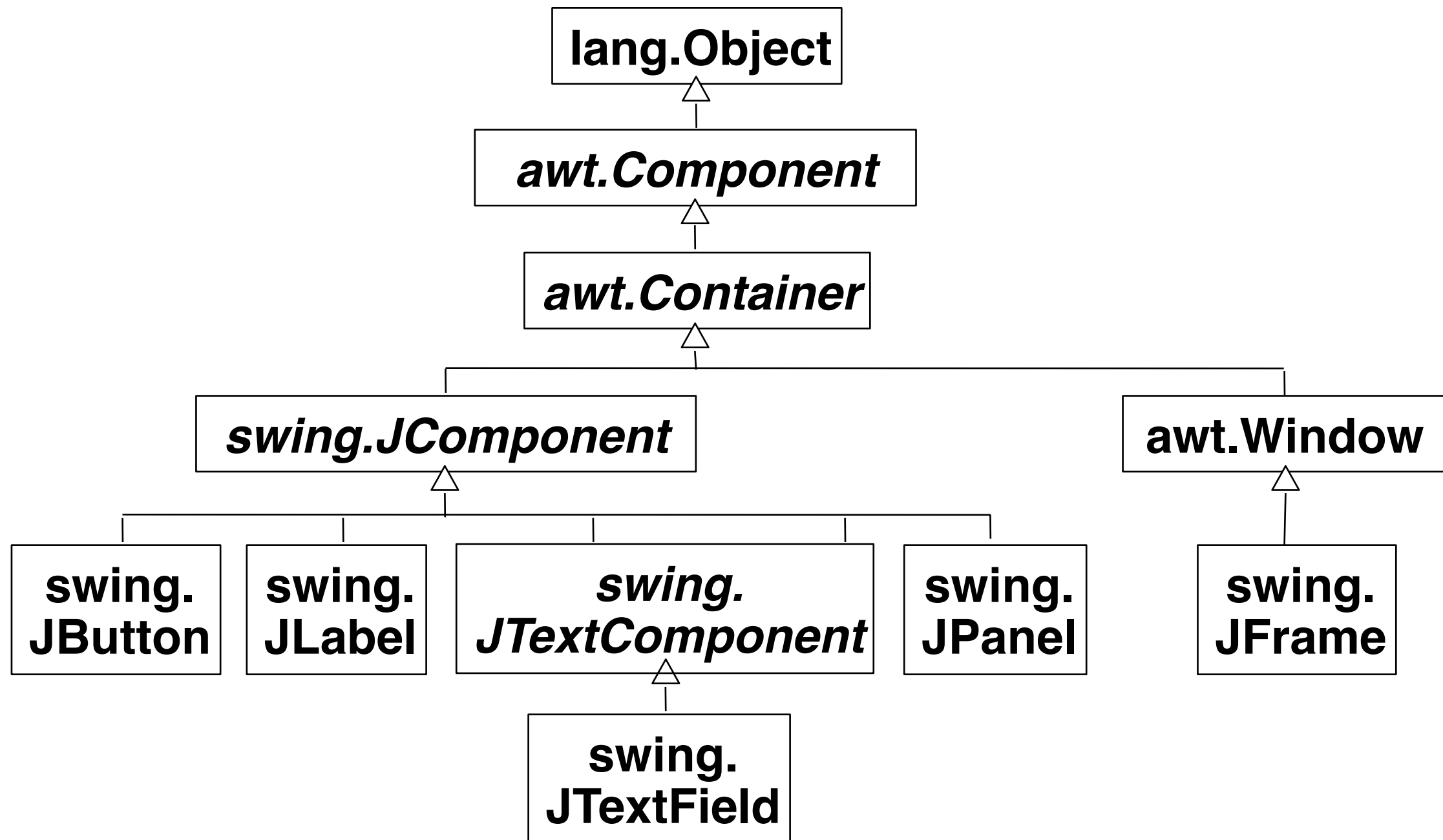


Bibliotheken von AWT und Swing

- 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
```

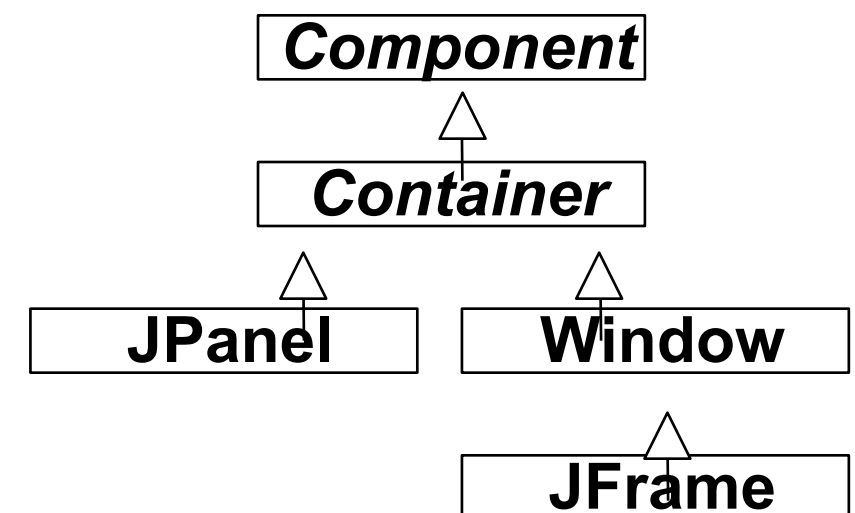
- **swing.JFrame**

- 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)

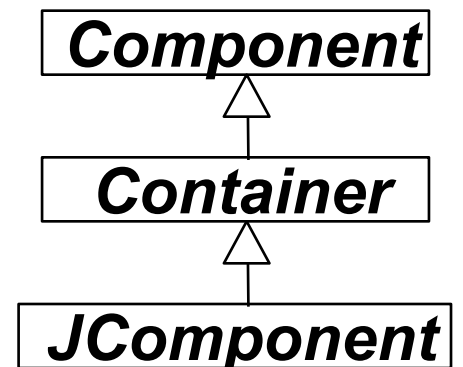
- ToolTips -- Kurzbeschreibungen, die auftauchen, wenn der Cursor über der Komponente liegt

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

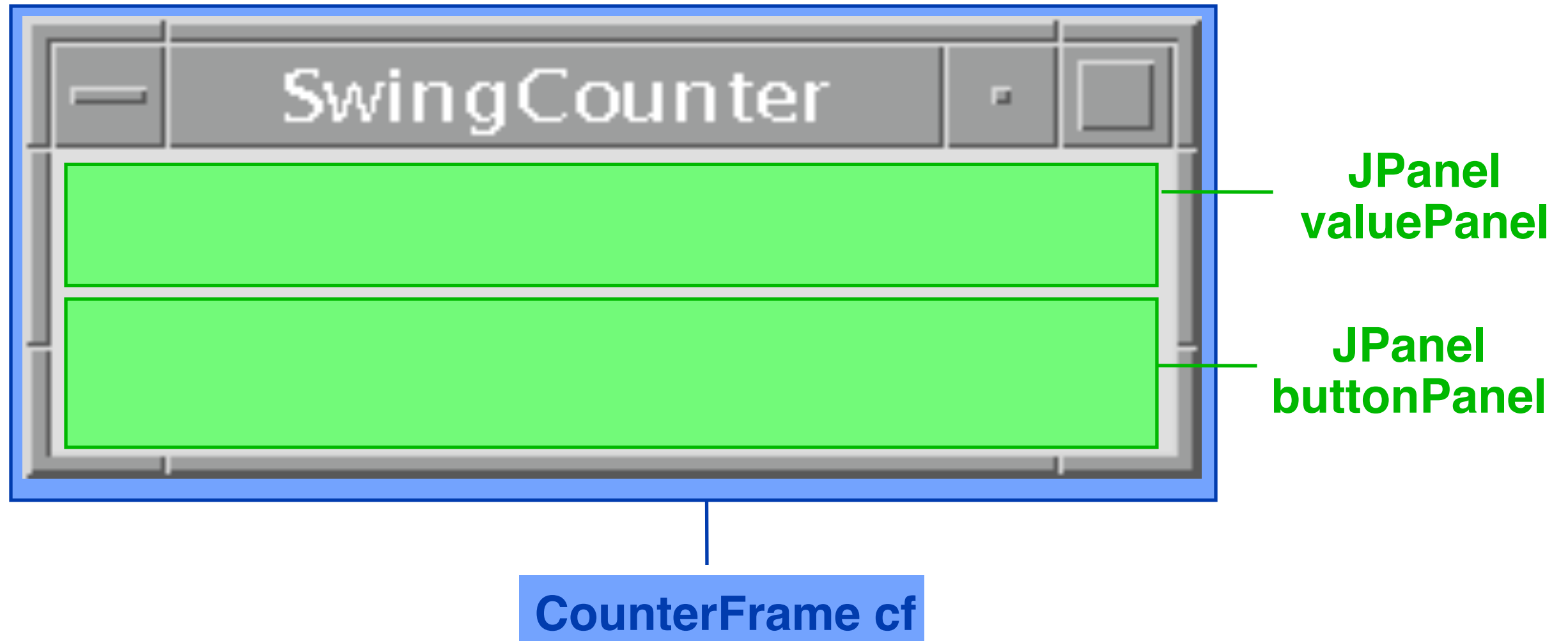
- Automatisches Scrolling

- 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, 1. Versuch

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

    JPanel buttonPanel = new JPanel();

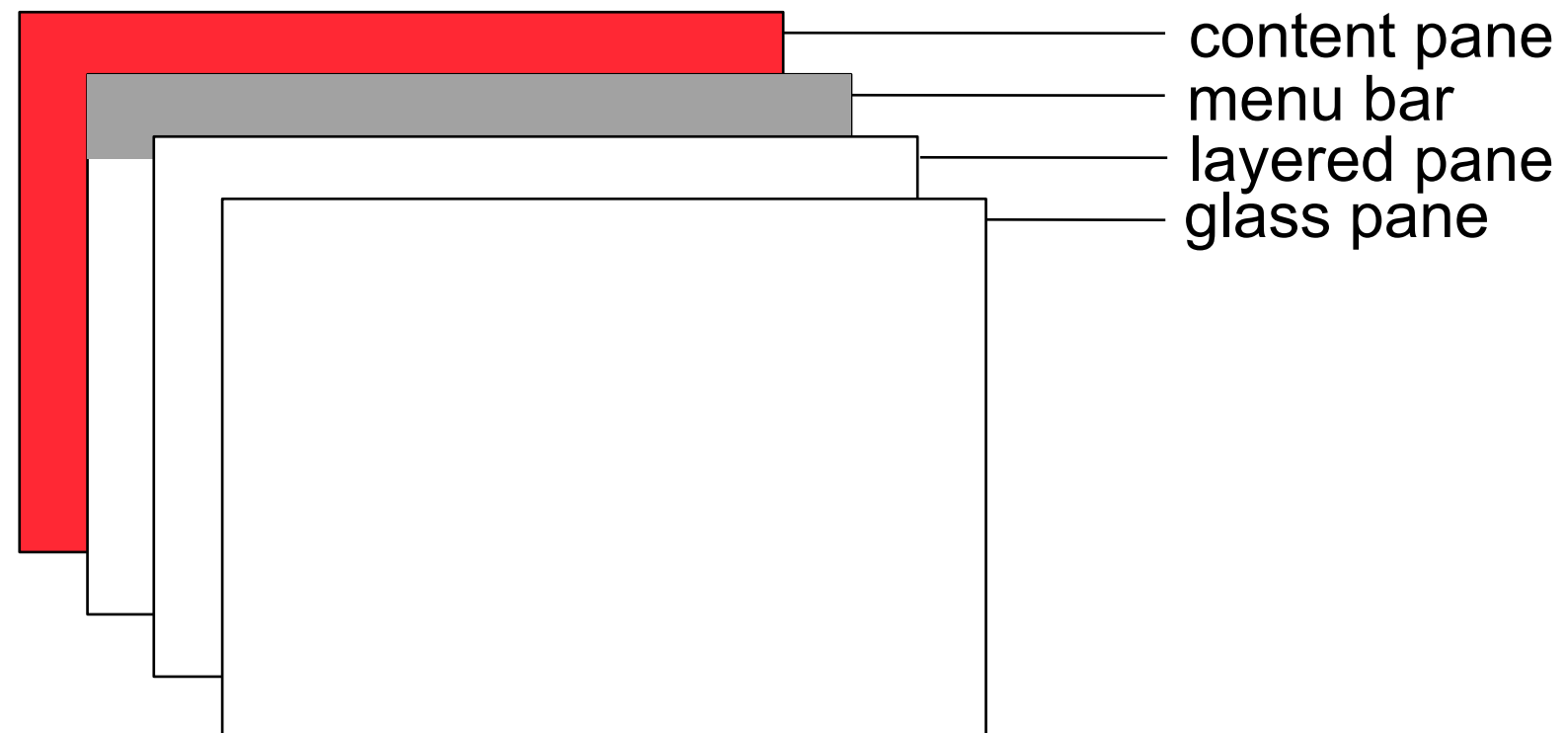
    public CounterFrame (Counter c) {
        setTitle("SwingCounter");

        ... valuePanel zu this hinzufügen

        ... buttonPanel zu this hinzufügen
        pack();
        setVisible(true);
    }
}
```

Hinzufügen von Komponenten zu JFrame

- Ein JFrame ist ein "Container", d.h. dient zur Aufnahme weiterer Elemente.
- Ein JFrame ist intern in verschiedene "Scheiben" (*panes*) organisiert. Die wichtigste ist die *content pane*.



- In JFrame ist definiert:
`Container getContentPane () ;`

Die Sicht (*View*): Gliederung, 2. Versuch

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

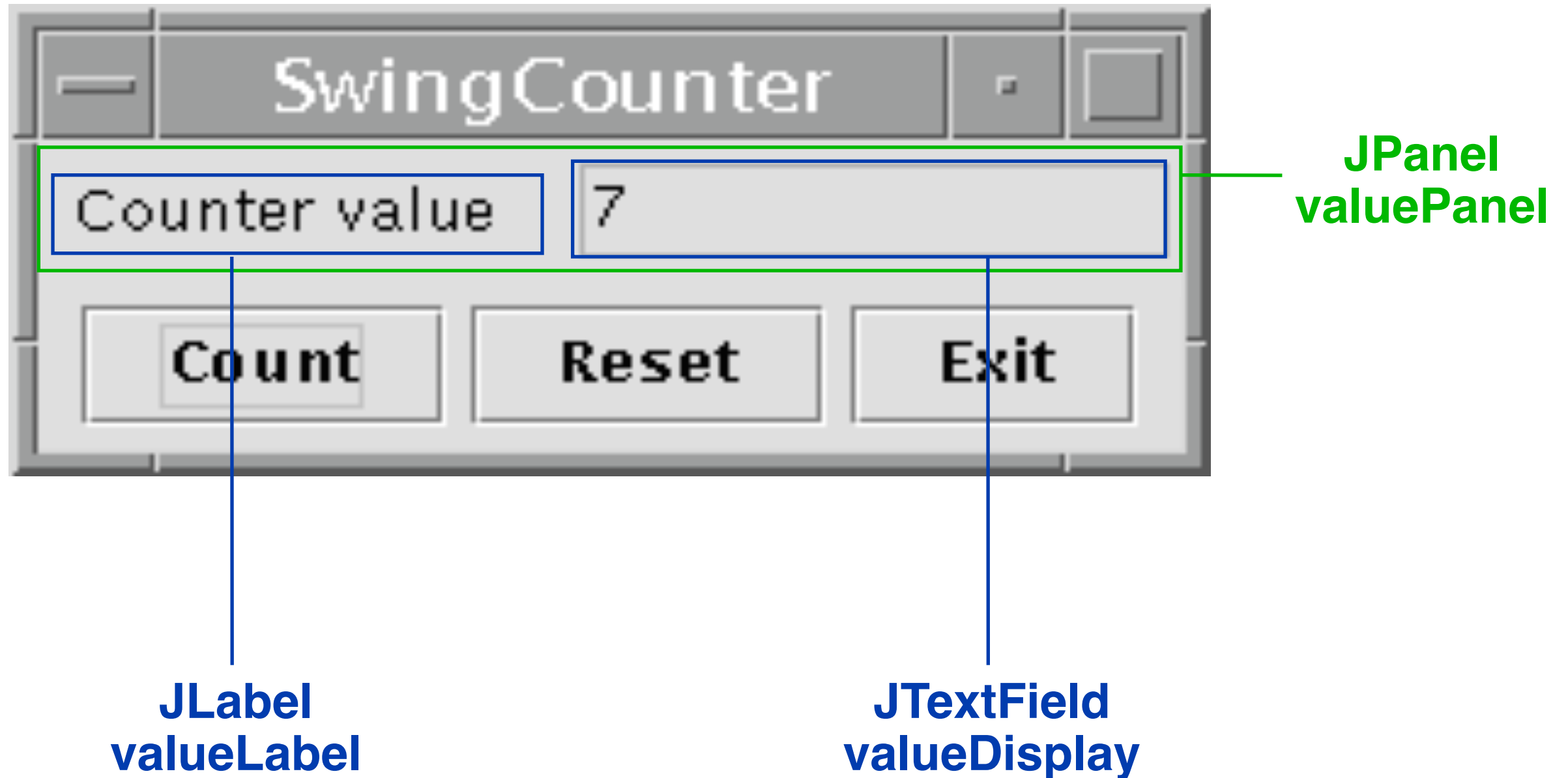
    JPanel buttonPanel = new JPanel();

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

        getContentPane().add(valuePanel);

        getContentPane().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:**

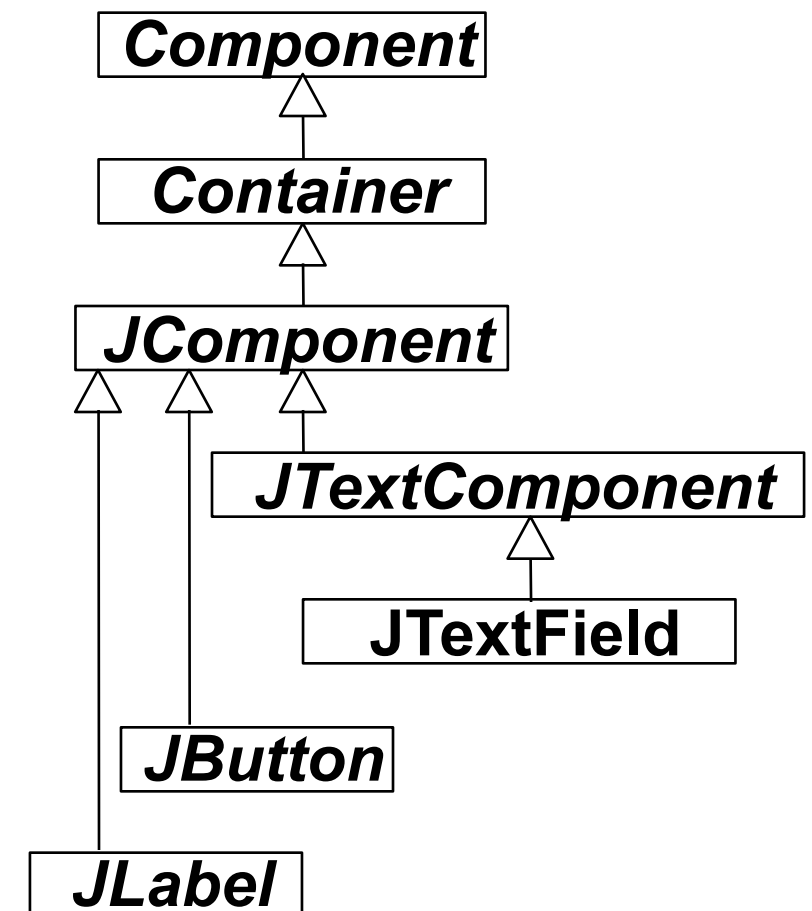
- Einzeiliger unveränderbarer Text

- ```
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);
        getContentPane().add(valuePanel);

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

B1. Ein-/Ausgabebetonte Programmierung

B1.1 Mensch-Maschine-Kommunikation

B1.2 Modell-Sicht-Paradigma

B1.3 Bausteine für grafische Oberflächen

B1.4 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();
    }
}
```

Event1.java

Einfaches Fenster mit Schaltfläche (Button)

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

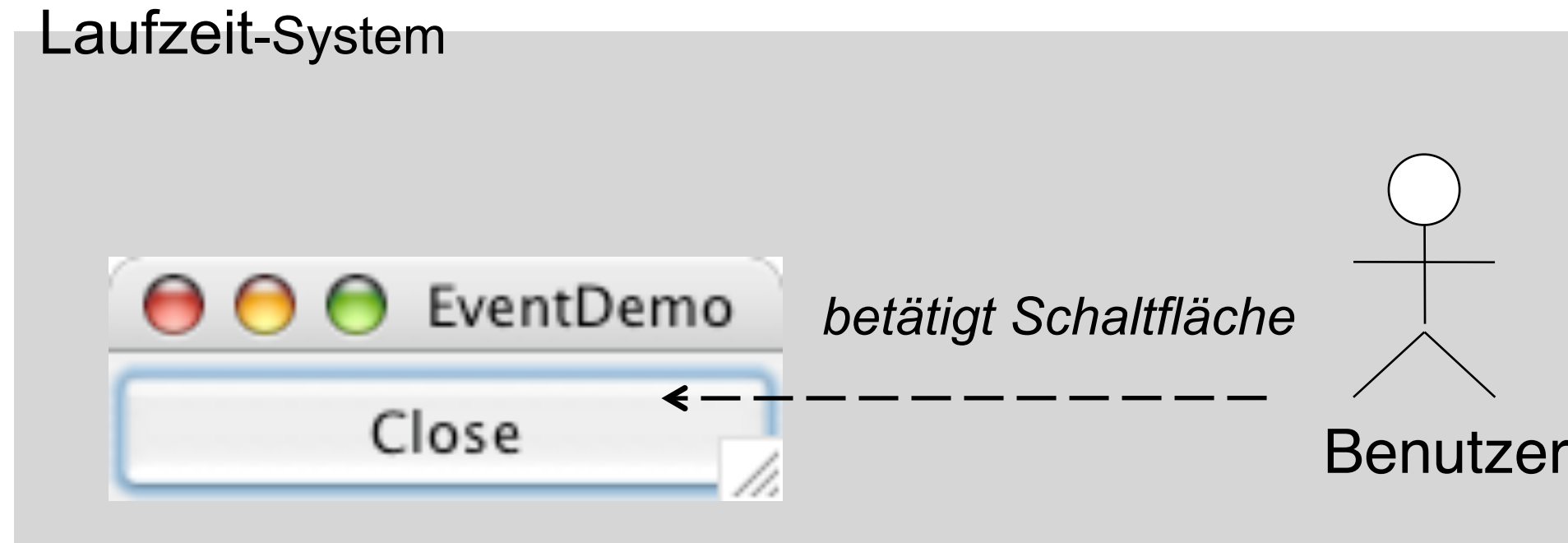
class EventDemoFrame extends JFrame {

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

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

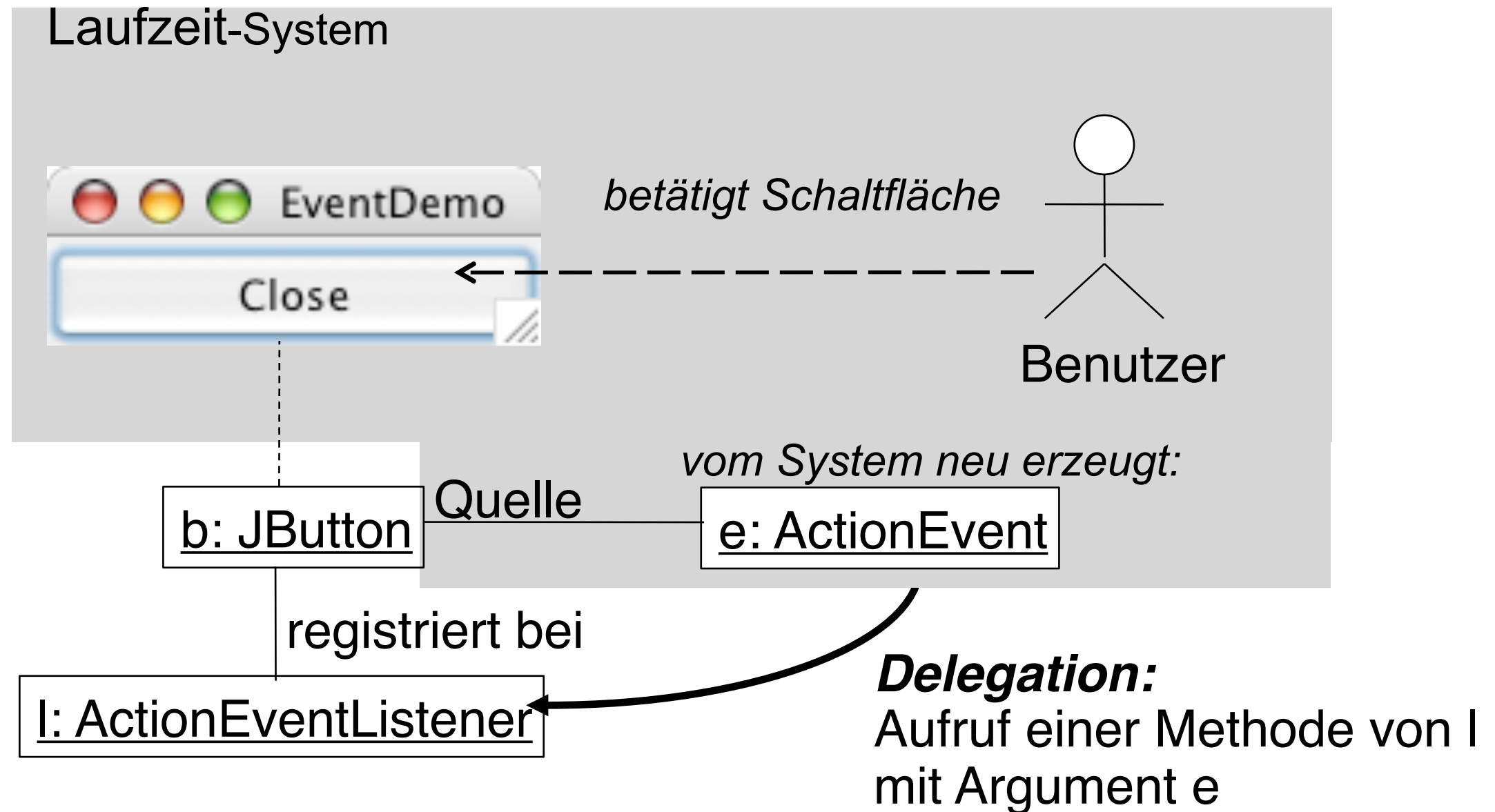
Event2.java

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:
 - Frame 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"

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

class CloseEventHandler implements ActionListener {
    ... siehe vorhergehende Folie ...
}

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

Event3.java

Vereinfachung 1: Innere Klasse

```
import ...

class EventDemoFrame extends JFrame {

    class CloseEventHandler implements ActionListener {
        ... siehe vorhergehende Folie ...
    }

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

class Event4 {
    public static void main (String[] argv) {
        EventDemoFrame f = new EventDemoFrame();
    }
}
```

Event4.java

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);
    }
}

class Event4 {
    public static void main (String[] argv) {
        EventDemoFrame f = new EventDemoFrame();
    }
}
```

Weiteres Beispiel: Fenster-Schließsymbol

- Standard-Verhalten bei Bedienung des Schließsymbols (X in Windows):
 - Fenster wird geschlossen, Programm läuft weiter
 - Wie Programm automatisch beenden?
- Einfachstes Verfahren (erst seit Java 1.4):
 - In JFrame:

```
setDefaultCloseOperation(EXIT_ON_CLOSE);
```
- Alternativ mit Ereignisbehandlung
 - WindowEvent wird vom Laufzeitsystem erzeugt
 - Geeigneten Listener registrieren
 - In der vom Listener aufgerufenen ("Callback"-)Methode das Programm beenden

java.awt.event.WindowListener

```
public interface WindowListener
    extends EventListener {
    public void windowClosed (WindowEvent ev);
    public void windowOpened (WindowEvent ev);
    public void windowIconified (WindowEvent ev);
    public void windowDeiconified (WindowEvent ev);
    public void windowActivated (WindowEvent ev);
    public void windowDeactivated (WindowEvent ev);
    public void windowClosing (WindowEvent ev);
}
```

java.util.EventListener:

Basisinterface für alle "Listener" (keine Operationen)

WindowListener für Ereignis "Schließen"

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

class WindowCloser implements WindowListener {

    public void windowClosed (WindowEvent ev) {}
    public void windowOpened (WindowEvent ev) {}
    public void windowIconified (WindowEvent ev) {}
    public void windowDeiconified (WindowEvent ev) {}
    public void windowActivated (WindowEvent ev) {}
    public void windowDeactivated (WindowEvent ev) {}

    public void windowClosing(WindowEvent event) {
        System.exit(0);
    }
}
```

Gesamtprogramm für schließbares Fenster

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

class WindowCloser implements WindowListener {
    ... siehe vorhergehende Folie ...
}

class ExampleFrame extends JFrame {

    public ExampleFrame () {
        setTitle("untitled");
        setSize(150, 50);
        addWindowListener(new WindowCloser());
        setVisible(true);
    }
}

class GUI2 {
    public static void main (String[] argv) {
        ExampleFrame f = new ExampleFrame();
    }
}
```

GUI2.java

java.awt.event.WindowAdapter

```
public abstract class WindowAdapter
    implements WindowListener {

    public void windowClosed (WindowEvent ev) {}
    public void windowOpened (WindowEvent ev) {}
    public void windowIconified (WindowEvent ev) {}
    public void windowDeiconified (WindowEvent ev) {}
    public void windowActivated (WindowEvent ev) {}
    public void windowDeactivated (WindowEvent ev) {}
    public void windowClosing (WindowEvent ev) {}

}
```

Vereinfachung: WindowAdapter benutzen

```
import ...
```

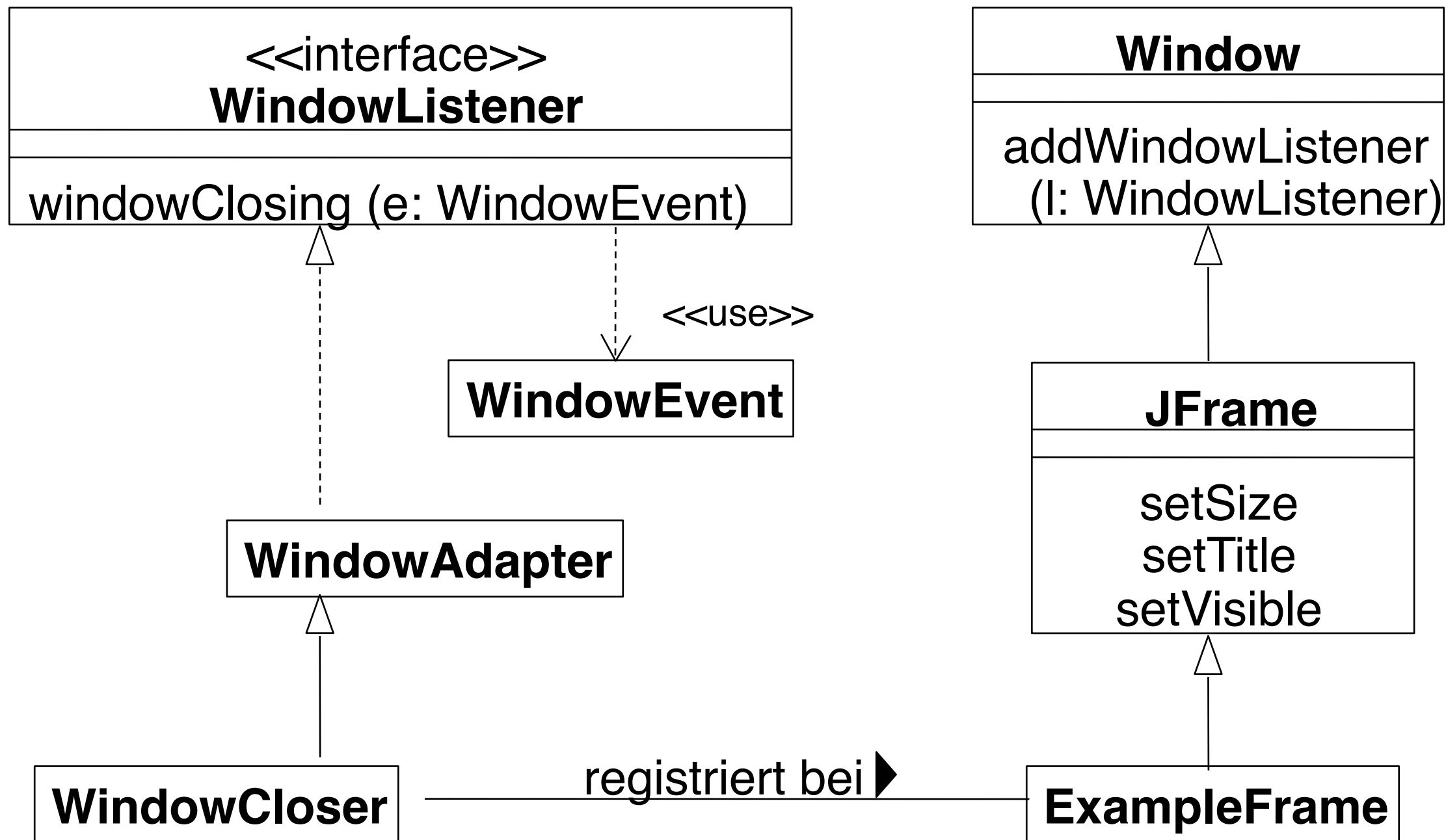
```
class WindowCloser extends WindowAdapter {  
    public void windowClosing(WindowEvent event) {  
        System.exit(0);  
    }  
}
```

```
class ExampleFrame extends JFrame {  
  
    public ExampleFrame () {  
        setTitle("untitled");  
        setSize(150, 50);  
        addWindowListener(new WindowCloser());  
        setVisible(true);  
    }  
}
```

```
class GUI3 {  
    public static void main (String[] argv) {  
        ExampleFrame f = new ExampleFrame();  
    }  
}
```

GUI3.java

Schließbares Fenster: Klassenstruktur



Mit anonymer innerer Klasse

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

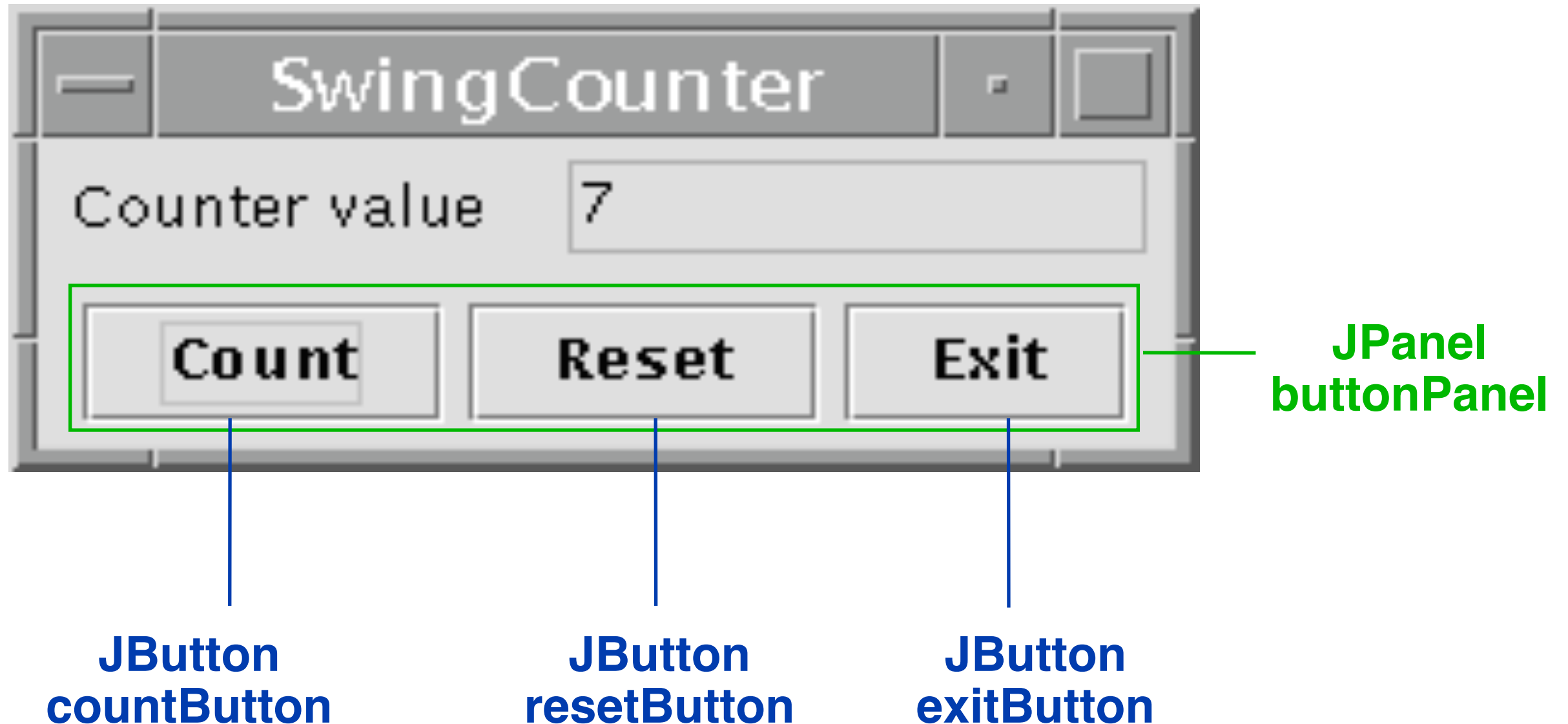
class ExampleFrame extends JFrame {

    public ExampleFrame () {
        setTitle("untitled");
        setSize(150, 50);
        addWindowListener(new WindowAdapter() {
            public void windowClosing(WindowEvent event) {
                System.exit(0);
            }
        });
        setVisible(true);
    }
}

class GUI5 {
    public static void main (String[] argv) {
        ExampleFrame f = new ExampleFrame();
    }
}
```

GUI5.java

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);
        getContentPane().add(valuePanel);
        buttonPanel.add(countButton);
        buttonPanel.add(resetButton);
        buttonPanel.add(exitButton);
        getContentPane().add(buttonPanel);
        pack();
        setVisible(true);
    }
}
```

Layout-Manager

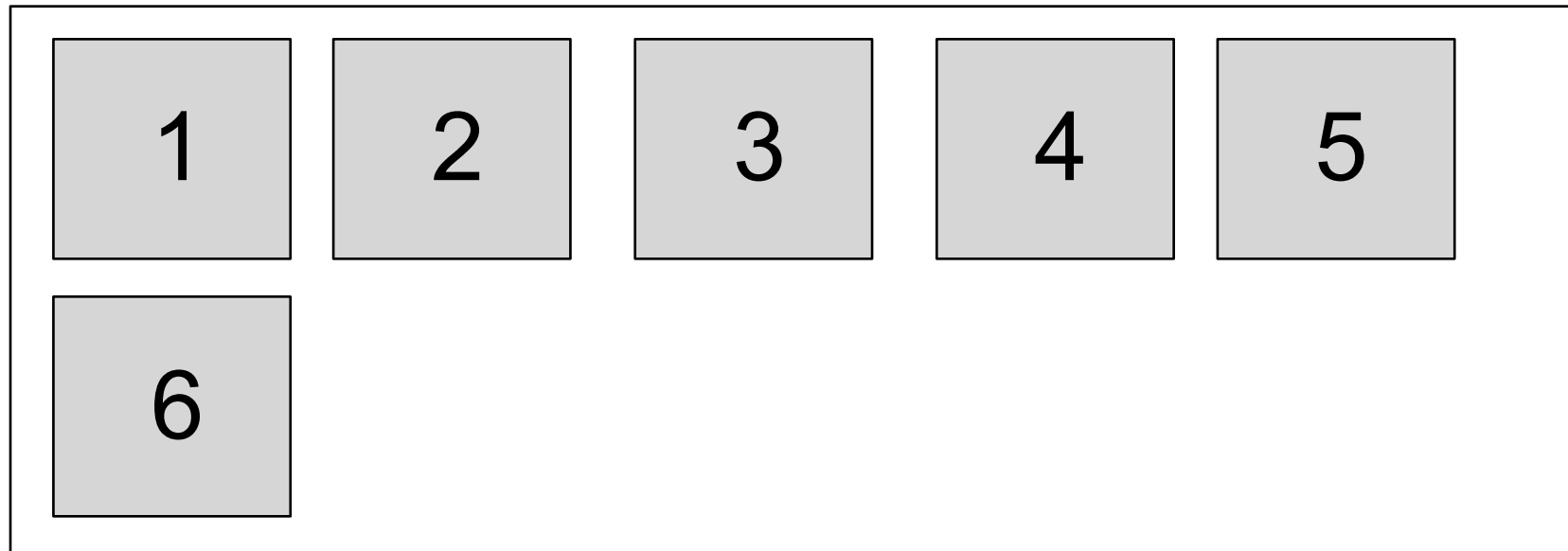
- **Definition** Ein *Layout-Manager* ist ein Objekt, das Methoden bereitstellt, um die graphische Repräsentation verschiedener Objekte innerhalb eines Container-Objektes anzuordnen.
- Formal ist `LayoutManager` ein Interface, für das viele Implementierungen möglich sind.
- In Java definierte Layout-Manager (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

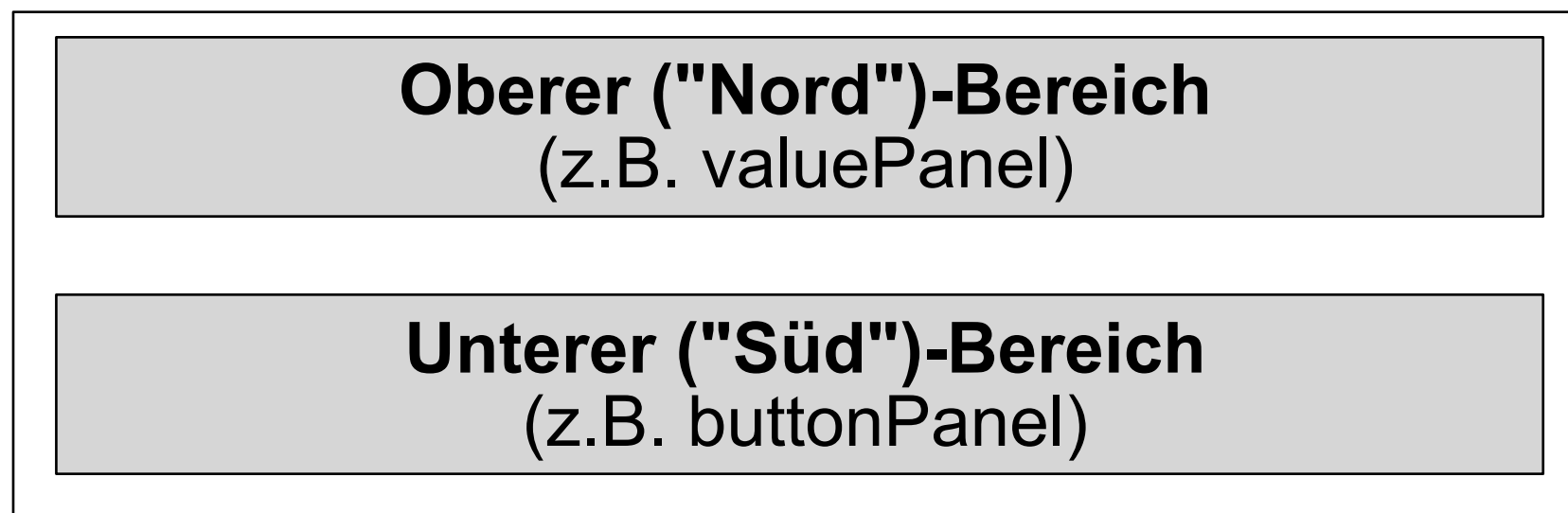
Flow-Layout

- 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



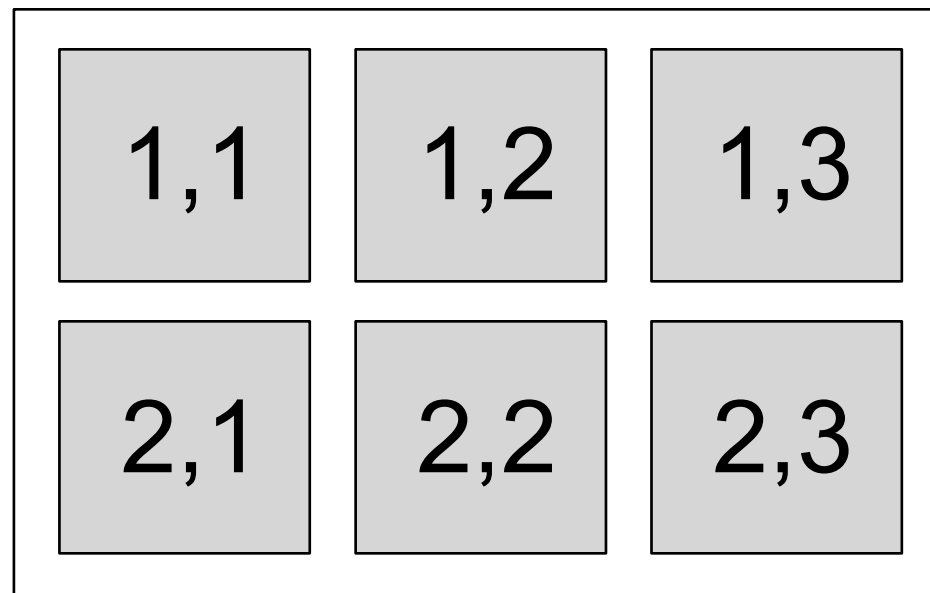
Border-Layout

- 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

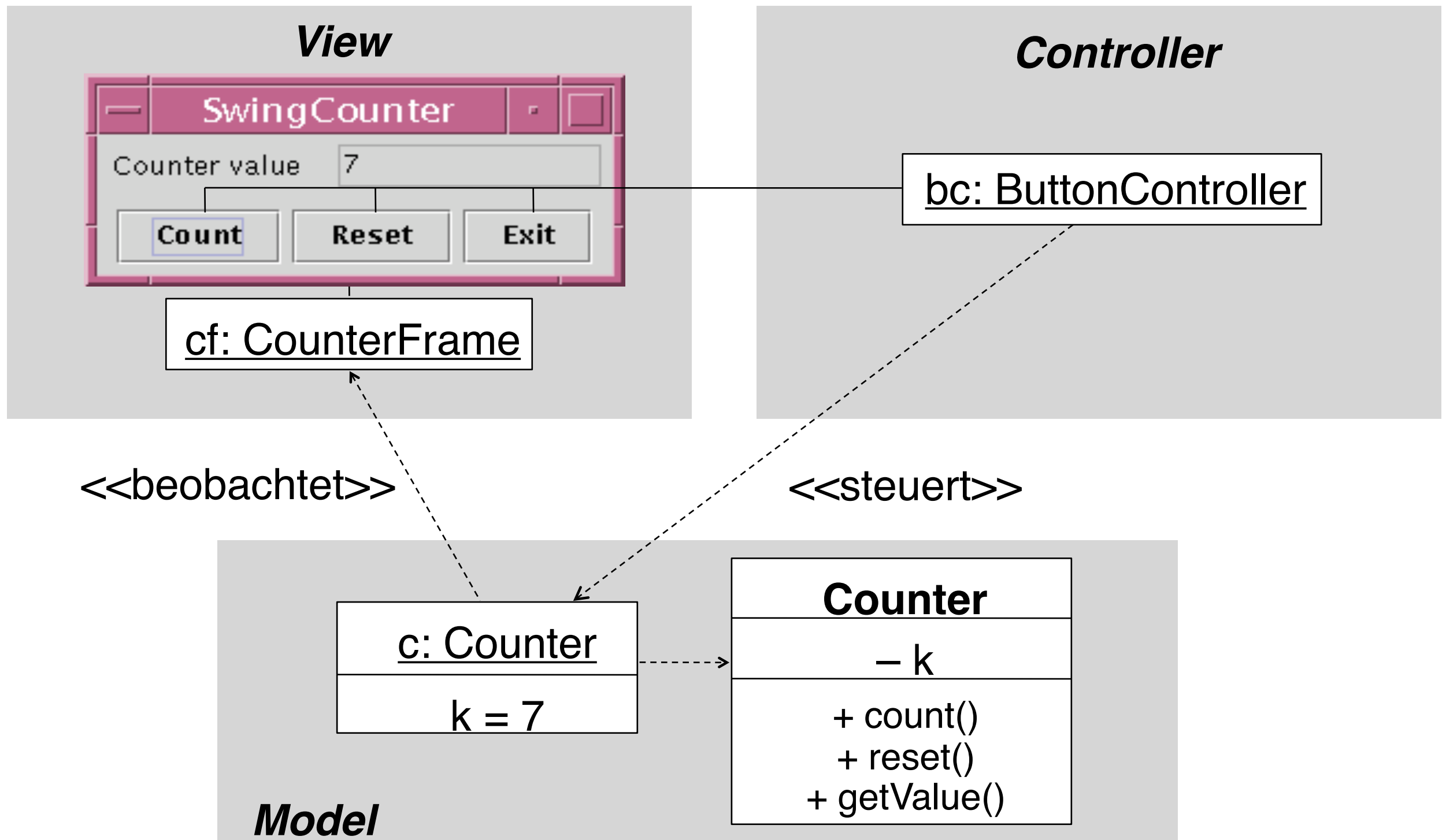


Die Sicht (*View*): Alle sichtbaren Elemente

```
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);
        getContentPane().add(valuePanel, BorderLayout.NORTH);
        buttonPanel.add(countButton);
        buttonPanel.add(resetButton);
        buttonPanel.add(exitButton);
        getContentPane().add(buttonPanel, BorderLayout.SOUTH);
        pack();
        setVisible(true);
    }
}
```


Model-View-Controller-Architektur



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()));
    }
}
```

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);  
        }  
    }  
}
```

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:
 - Direkte Angabe von Eventhandlern
 - » am knappsten über anonyme innere Klassen
 - Viele Controller-Objekte
 - Siehe weiter hinten

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);
        ...
    }
}
```

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)

```
        addWindowListener(new WindowCloser());
        c.addObserver(this);
        pack();
        setVisible(true);
    }

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

class ButtonController implements ActionListener {
    ... (wie oben) ...
}

class WindowCloser implements WindowListener
    extends WindowAdapter {
        public void windowClosing(WindowEvent event) {
            System.exit(0);
        }
    }
}
```

Controller durch anonyme Klassen

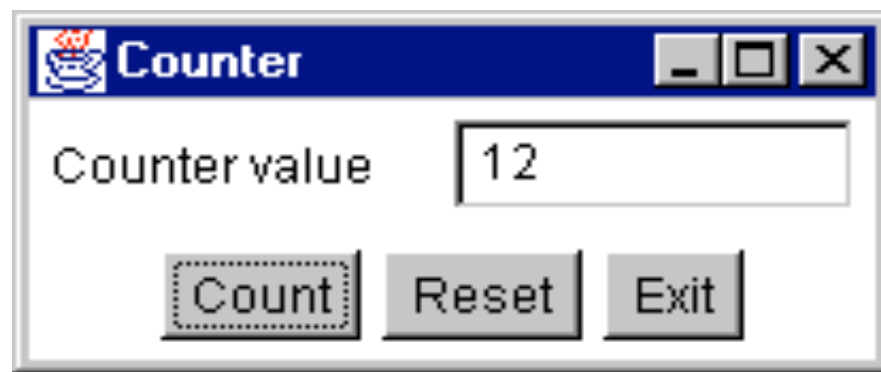
```
class CounterFrame extends JFrame { ...
    private Counter ctr;
    ...

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

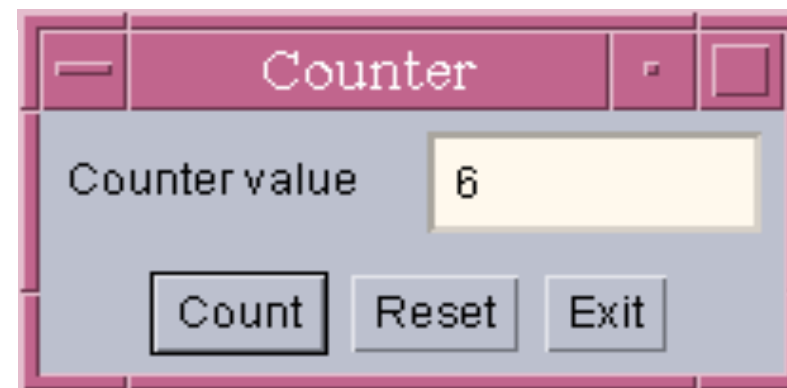
Controller und View bilden eine Einheit: In der Praxis weit verbreitet.

"Look-and-Feel"

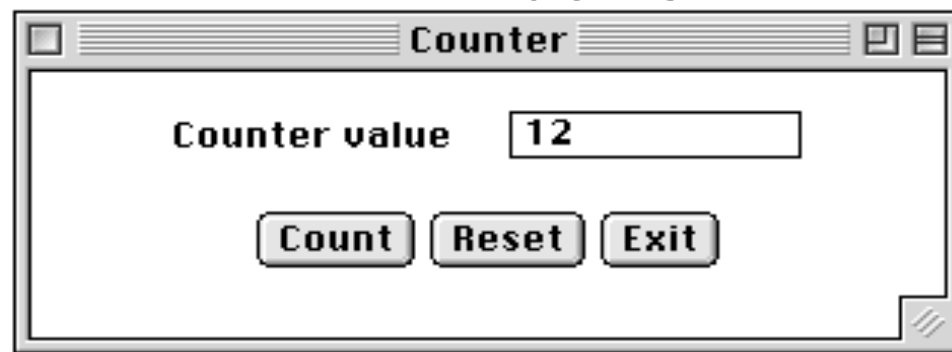
- Jede Plattform hat ihre speziellen Regeln für z.B.:
 - Gestaltung der Elemente von "Frames" (Titelbalken etc.)
 - Standard-Bedienelemente zum Bewegen, Schließen, Vergrößern, von "Frames"
- Dasselbe Java-Programm mit verschiedenen "Look and Feels":



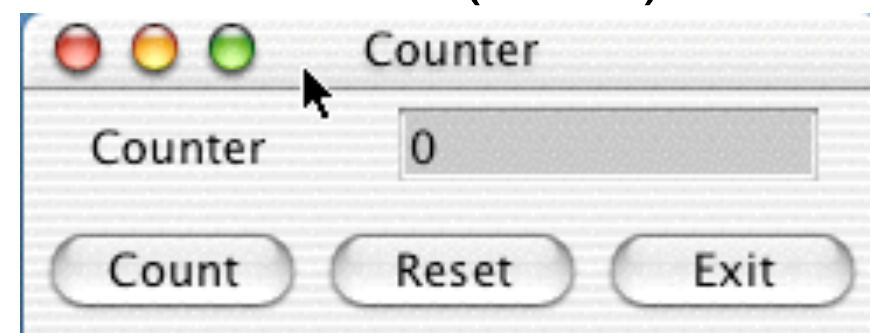
Windows



Solaris (CDE)



Macintosh (Classic)



Macintosh (MacOS X)

- Einstellbares Look-and-Feel: Standard-Java oder plattformspezifisch