



Programmiermethodik

Swing & JavaDoc

SS 2002

Thomas Kühne

kuehne@informatik.tu-darmstadt.de

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

Trennung von Anwendung & Oberfläche

- **Anwendungsklassen sollen nicht von GUI-Klassen abhängig sein**
 - » GUI kann geändert werden (neue Version, Verbesserungen an der Benutzungsschnittstelle)
 - » ermöglicht Wiederverwendung von Teilen der Applikation in anderen Kontexten mit anderer GUI
- **Nebenläufige Behandlung**
 - » Benutzungsschnittstellen Operationen (z.B., Auffrischen) sind unabhängig vom Applikationsverlauf
 - » GUI bleibt ansprechbar, auch wenn sich die Applikation in einer intensiven Berechnung befindet

Ereignisbasierter Entwurf

- Benutzereingaben (Maus, Tastatur) werden in Ereignisse übersetzt, für die sich Objekte (**Controller**) anmelden können
 - » **Subjekt** = GUI (Widget), **Observer** = Applikation (Handler)
- Applikationsobjekte (**Model**) versenden Ereignisse wenn sie sich ändern
 - » **Subjekt** = Applikation (Daten), **Observer** = GUI (Views)

Die Kommunikation über Ereignisse *entkoppelt* die Komponenten. GUI-Eingabekomponenten müssen die Applikation nicht kennen und die Applikation muß keine GUI-Ausgabekomponenten kennen.

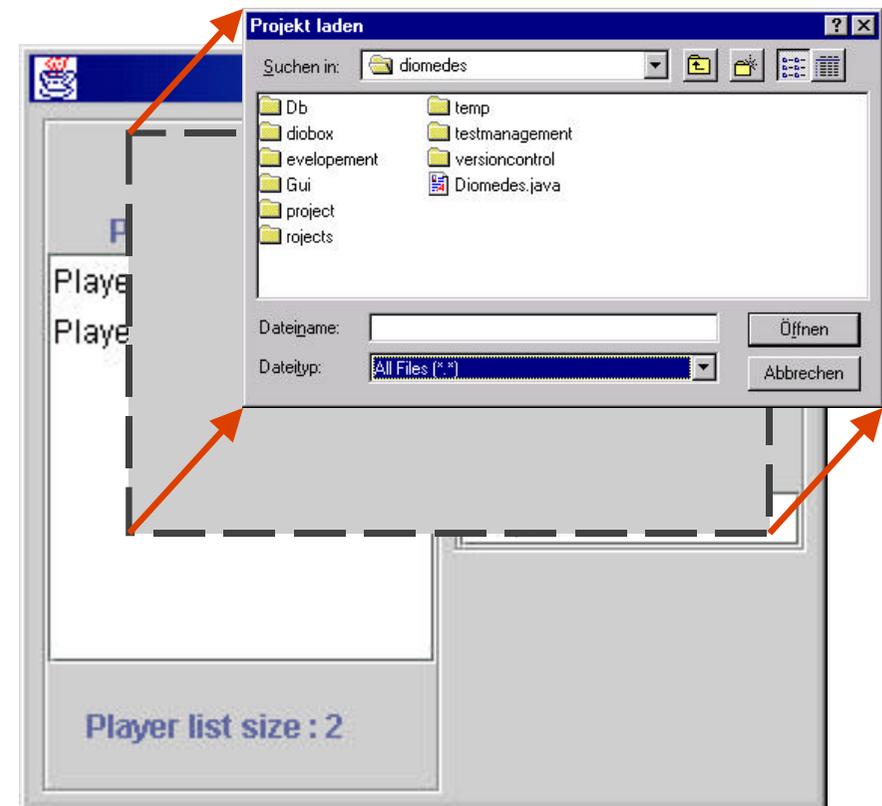
Benötigte Bausteine

- Erzeugen und Anordnung von Ein-/Ausgabekomponenten (Widgets)
- Definition der Aktionen bei Benutzerinteraktionen (Widgetbenutzung)
- Definition der Darstellung von eigenen Komponenten (z.B., Graphdarstellungen)
- Der Anwendungscode selbst
 - » erzeugt zunächst das GUI (erster Punkt oben)
 - » reagiert dann meist nur noch auf Ereignisse (zweiter Punkt)
 - » oder auf **Aufforderungen Sichten wiederherzustellen** (dritter Punkt)

Hollywood Prinzip: *"Don't call us, we call you"*

1. Dialogfenster wird verschoben
2. Inhalt des dahinterliegenden Fensters muß nachgezeichnet werden
3. Listendaten werden von der Applikation angefordert

Nicht die Applikation zeichnet aktiv, sondern wird nur aufgefordert Daten zu liefern





GUI-Widgets

The screenshot shows a window titled "AWT D 22:24 06-MAR-97" with a menu bar containing "File" and "Help". The window contains several GUI widgets:

- MenuBar**: The menu bar at the top.
- TextField**: A text input field labeled "Name:" with the text "essages".
- Choice**: A dropdown menu for "Favorite color:" with options "red", "green", and "blue" (selected).
- Checkbox & CheckboxGroup**: Radio buttons for "flavor:" with options "vanilla", "chocolate", and "strawberry" (selected).
- List**: A list box for "Favorite languages:" with items "Java", "C", "C++", "Smalltalk", "Lisp", "Modula-3", and "Forth".
- Button**: An "Okay" button at the bottom left.
- Canvas**: A large empty rectangular area labeled "Diagram".
- ScrollBar**: Vertical scroll bars on the right side of the "TextArea" and "Diagram" components.

Applikation vs Applet

- Java-Applikation

- » Standardfenster vom Typ **Frame** oder **JFrame**

Swing Variante
von **Frame**



- Java-Applet

- » Es muß kein Standardfenster erzeugt werden

- Die Klasse **Applet** ist bereits eine Unterklasse einer Fläche (**Panel**)

- Der Browser wird direkt als "Fenster" benutzt

- Applets können trotzdem auch Standardfenster verwenden

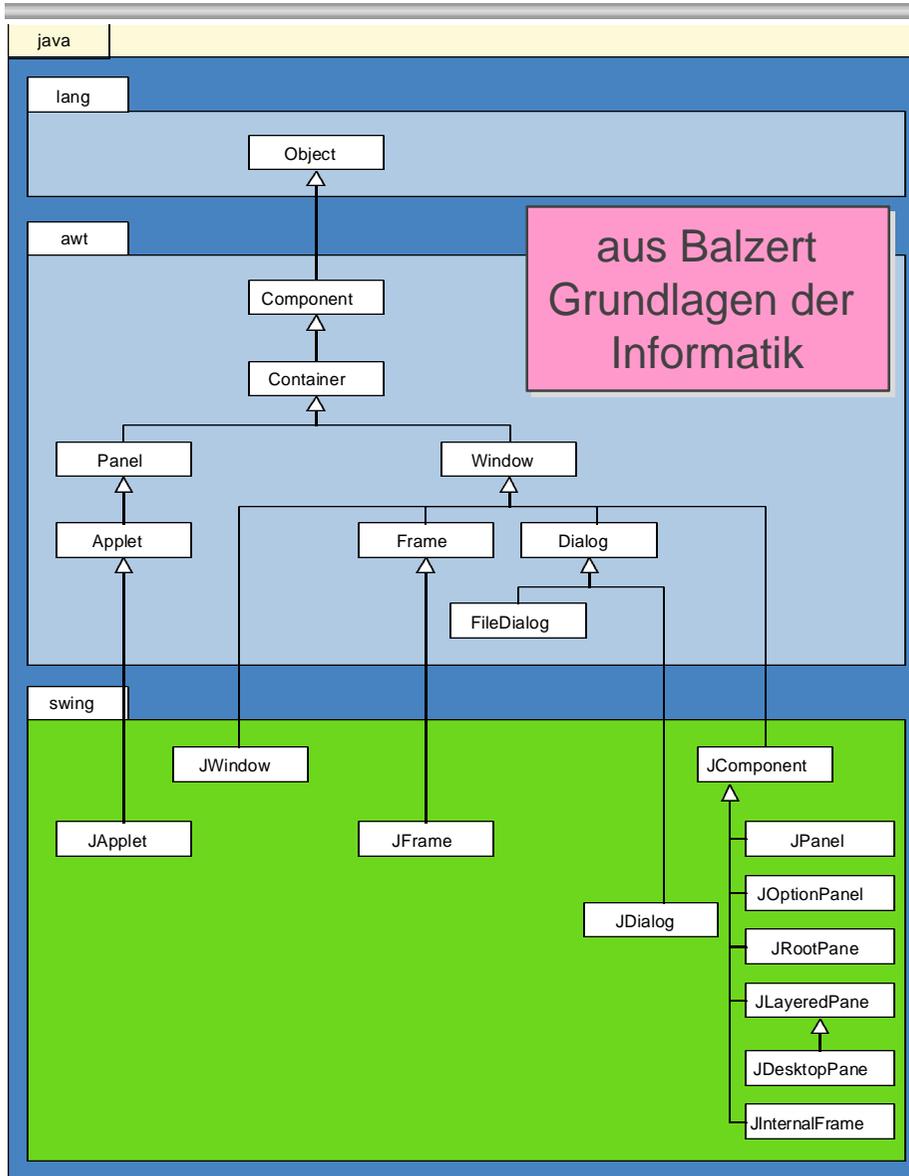


JDK GUI Bibliotheken

- Abstract Window Toolkit (AWT)
 - » plattformunabhängig, benutzt aber die jeweils vorhandenen Widgets
 - » bis JDK 1.1.6 Standard-Graphik-Bibliothek
- Swing
 - » Teil der Java Foundation Classes (JFC), die noch weitere nützliche Bibliotheken enthält
 - Java 2D: verbesserte 2D-Grafikroutinen
 - Drag-and-Drop
 - » plattformunabhängig, simuliert wahlweise jedes beliebige Look-and-Feel (Windows, Mac, etc.)



JDK GUI Bibliotheken



Vorteile von SWING

- wesentlich mehr und mächtigere Widgets stehen zur Verfügung
- keine Einigung auf den kleinsten gemeinsamen Nenner mehr nötig
- simuliertes Look&Feel
- Model-View Konzept
- "Assistive technologies" (z.B., Fenster vorlesen) © T. Kühne



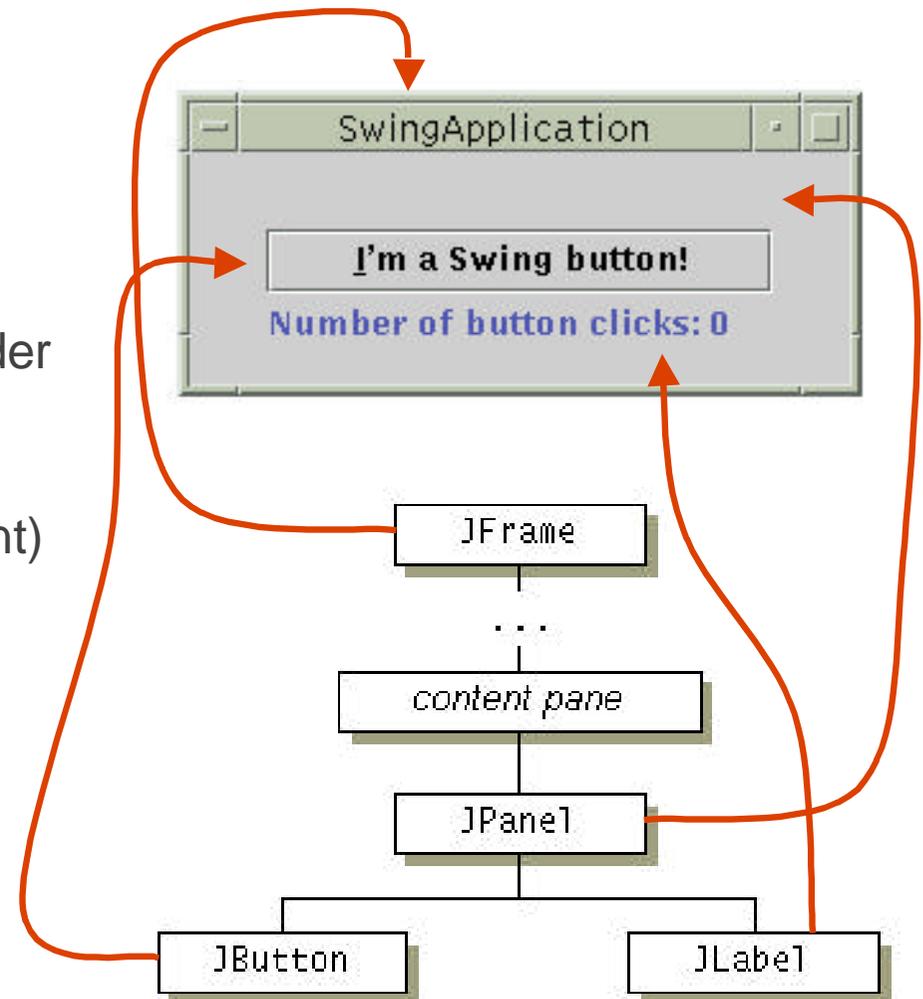
Behälterkonzept

Hierarchischer Widgetaufbau

- Fenster, Panels, usw. sind Behälter, die andere Komponenten enthalten können
- Hinzufügen neuer Komponenten
 - » Alle Behälter besitzen die Operation `add(Component)`
 - » Bei `JFrame`:
`getContentPane().add(component)`

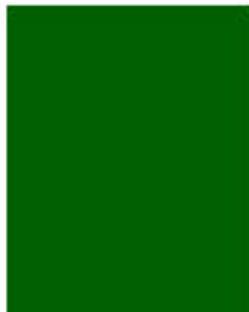
Fenster-Aufbau

- **Frame**: Hauptfenster einer Anwendung, kann eine Menüleiste enthalten
- **Panel**: Fläche mit Hintergrund auf der Darstellungen erfolgen können oder andere Komponenten angeordnet werden können (Gruppierungselement)
- **Button**: Schaltfläche mit einer Bezeichnung (z.B., "OK"), die bei Auslösung ein "Action Event" signalisiert
- **Label**: Textfeld, das einen vom Benutzer nicht veränderbaren Text anzeigt



- Darstellung der Behälterhierarchie von außen nach innen
 - » wird automatisch durch AWT durchgeführt

1. background
(if opaque)



2. custom
painting
(if any)



3. border
(if any)



4. children
(if any)



- Änderung von Inhalten (z.B., Text) von fertigen Komponenten führt zur Neudarstellung

- Benutzerdefinierte Komponenten
 - » automatischer Aufruf der `paintComponent(Graphics g)` Methode
 - » Diese zeichnet zunächst den Hintergrund (z.B., durch Aufruf der `paintComponent`-Methode der Oberklasse → oft `JPanel`)
 - » Danach werden Inhalte gezeichnet
- Inhaltsauffrischung der benutzerdefinierten Komponente erfolgt nie direkt
 - » Bitte nach Auffrischung durch `repaint()`

Zugriff auf die "Leinwand" zum Zeichnen

Hollywoodprinzip!



Außerordentliches Update

Abfolge der Aktionen

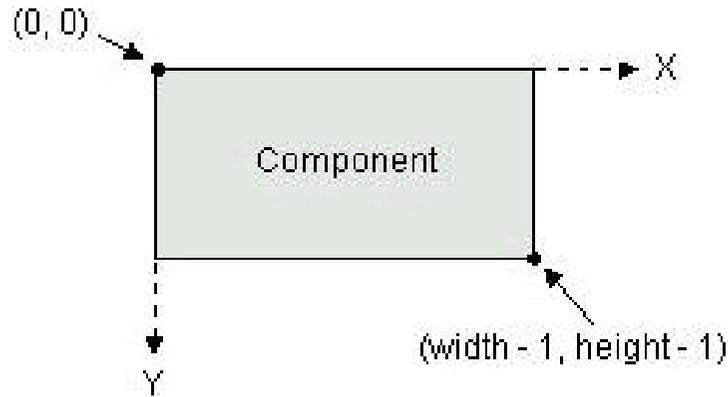
1. Senden der `repaint()` Nachricht an die Komponente
2. Die Aktion "Auffrischen der Komponente" wird in die Warteschlange des GUI-Threads eingefügt
3. Wenn die Aktion bearbeitet wird, werden ggf. mehrere updates zusammengefasst
4. Senden der Nachricht `paint` an die Komponente
5. Die Methode `paint` realisiert double-buffering und ruft `paintComponent` auf
6. Methode `paintComponent` zeichnet den eigentlichen Inhalt

- Bei sich rasch wechselnden Komponenteninhalten (Animationen, dynamische Darstellung von Daten) kann es zu unansehnlichen Flackern kommen
 - » Wenn ein neuer Zyklus gezeichnet wird, entsteht immer ein kurzer Moment in dem nur der neu gezeichnete Hintergrund zu sehen ist
 - » Erst dann werden die Vordergrundobjekte wieder gezeichnet
- Abhilfe durch Double-Buffering
 - » Der neue Inhalt wird nicht direkt auf den Bildschirm gezeichnet sondern in eine Bitmap
 - » Erst die fertige Bitmap wird mit einer Operation auf den Bildschirm kopiert
 - » So sind nur noch vollständige Bilder sichtbar und das Flackern entfällt



- Die Swing-Komponente **JComponent** unterstützt bereits Double-Buffering
 - » Die Unterstützung kann mit der Operation `public void setDoubleBuffered(boolean aFlag)` ein- und ausgeschaltet werden
 - » Erben von **JComponent**, wie z.B., **JPanel** erhalten diese Fähigkeit ebenfalls
 - » Die Kinder einer Komponente, die einen Double-Buffer besitzt, benutzen ebenfalls diesen Buffer, anstatt einen neuen zu erzeugen

Benutzerdefinierte Komponenten



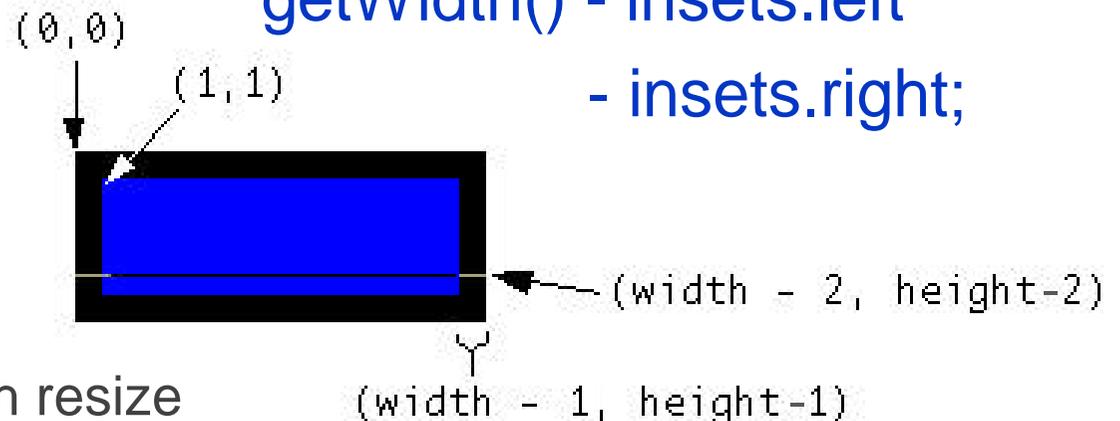
- Jede Komponente hat ihr eigenes Koordinatensystem
 - » Ursprung ist links oben
 - » Größe läßt sich erfragen
 - » Größenänderungen durch resize Operationen möglich

- Ein ggf. vorhandener Rahmen muß berücksichtigt werden

```
void paintComponent(Graphics g)
```

...

```
Insets insets=getInsets();
int currentWidth=
    getWidth() - insets.left
    - insets.right;
```





Ein Anwendungsbeispiel

- Eine Anwendung ändert laufend einen Teil ihrer Daten
 - » Hier ein einfaches Positionsmodell
- Die Daten sollen aktuell visualisiert werden
 - » Darstellung eines weißen Punkts auf grünem Hintergrund
- Anwendungs- und Darstellungscode soll nicht miteinander vermischt werden

Die Applikation

- Folgende Methode kann Teil irgendeiner Klasse sein (z.B., Model)

```
public static void main(String args[])
{
    Model model=new Model(); // create a new model

    // wait a bit for view to be fully displayed
    for (int j=0; j<500000; j++)
        Math.pow(2, 64);

    // continuously change model data for a while
    model.action();
}
```

Das Applikationsmodell
erzeugt die grafische
Sicht selbst



Das Modell

```
public class Model {  
    private int x=15;           // application  
    private View view;        // need to tell view about updates  
  
    public Model() {  
        view = new View (this); } // create a view for the model  
  
    public int position() { // enable access for view  
        return x; }  
  
    public void hop() { // change application data  
        x+=20; }  
  
    ...  
}
```

Eine neue Sicht soll genau dieses Modell "beobachten"





Das Modell

...

```
public void action() { // let the ball move a number of times
    for (int i=5; i<200; i+=20) {
        hop();          // make a change to the model
        view.repaint(); // ask view to refresh itself
```

```
// wait a bit in so that intermediate steps can be seen
```

```
for (int j=0; j<250000; j++)
    Math.pow(2, 64);
```

```
}
```

```
} // end class Model
```

Ohne diese Verzögerung
(*entsteht sonst durch Ausführung
anderer Anwendungscodes*)
würden alle **repaints** gesammelt
werden und es wäre nur der letzte
Zustand sichtbar



Die Sicht

```
class View extends JFrame { // inherit Window functionality

    public View (Model model) // accept model to observe
    {
        super("Swing Example"); // set window title
        setSize(300, 170); // set window size

        // replace content pane with new playfield component
        setContentPane(new PlayfieldPanel(model));
        setVisible(true); // show window
    }
}
```

Benutzerdefinierte
Darstellungs-
komponente

Statt Ersetzen ist normalerweise Hinzufügen üblich.
[getContentPane().add(new PlayfieldPanel (model));]
Da wir jedoch keinen gesonderten Hintergrund bzw.
Behälter brauchen, wird er hier einfach überschrieben



Die Darstellungskomponente

```
class PlayfieldPanel extends JPanel { // inherit double buffering
    private Model model; // the model to be visualized
```

```
public PlayfieldPanel (Model m) {
    model=m; // save model for later access
    setBackground(Color.green.darker()); // inherited feature
}
```

Default-Grün etwas
abdunkeln

```
public void paintComponent(Graphics g) {
    super.paintComponent(g); // draw (green) background first
    g.setColor(Color.white); // choose foreground color
    g.fillOval(model.position(), 50, 35, 35);
}
```



Ball an der momentanen
Position zeichnen



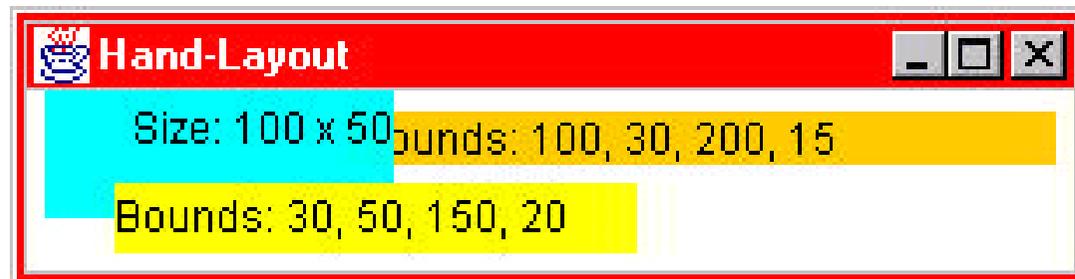
Mehr Komponenten

- Sobald mehr als eine Komponente in einem Fenster (oder einem Panel) angeordnet ist, ergibt sich die Frage der Anordnung
- Eine fixe Anordnung ist möglich hat aber Nachteile
 - » Größenänderung des Fensters nicht möglich
 - » Da z.B., Textgrößen auf verschiedenen Plattformen leicht unterschiedlich sein können, sind Überschneidungen vorprogrammiert



Warum Layout-Manager?

```
public class Fenster extends Frame {  
    public Fenster() {  
        Label satz; Kein Layout Manager  
        setLayout( null );  
        add( satz = new Label( "Bounds: 30, 50, 150, 20" ) );  
        satz.setBounds( 30, 50, 150, 20 ); satz.setBackground( Color.yellow );  
        add( satz = new Label( "Size: 100 x 50", Label.RIGHT ) );  
        satz.setSize( new java.awt.Dimension( 100, 50 ) );  
        satz.setLocation( 10, 10 ); satz.setBackground( Color.cyan );  
        add( satz = new java.awt.Label( "Bounds: 100, 30, 200, 15" ) );  
        satz.setBounds( 100, 30, 200, 15 );  
        satz.setBackground( Color.orange );  
        setSize( new Dimension( 310, 80 ) );  
        setVisible( true );  
    }  
}
```



Aufwendig und nicht flexibel



Layout Manager

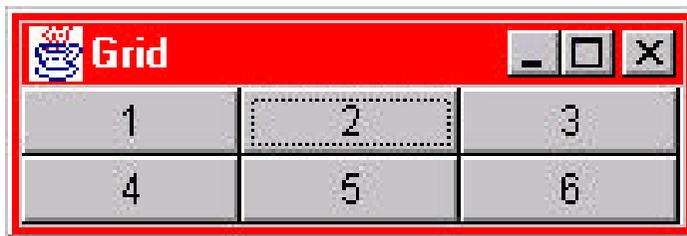
```
setLayout( new BorderLayout( ) );  
add( new Button( "Norden" ), BorderLayout.NORTH );  
add( new Button( "Zentrum", BorderLayout.CENTER );
```

BorderLayout

FlowLayout

GridLayout

und noch
viele mehr...



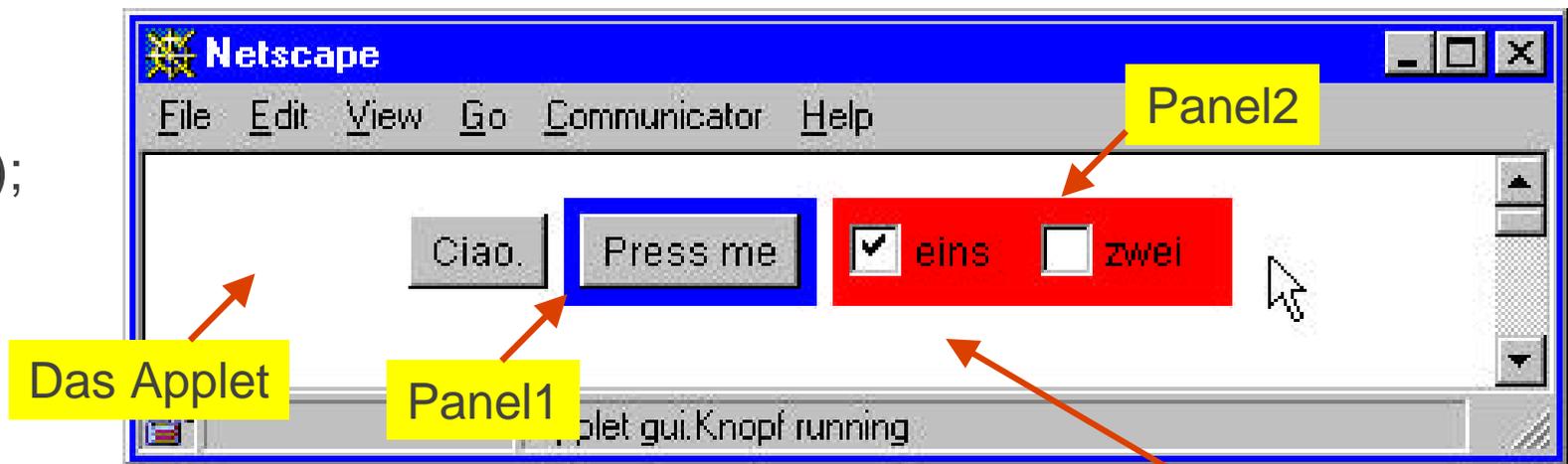
Panels zur Gruppierung

- **JPanel:** Behälter mit Hintergrund

```
Panel p1 = new Panel();
p1.setBackground( Color.blue );
p1.add( new Button( „press me“ ) );
```

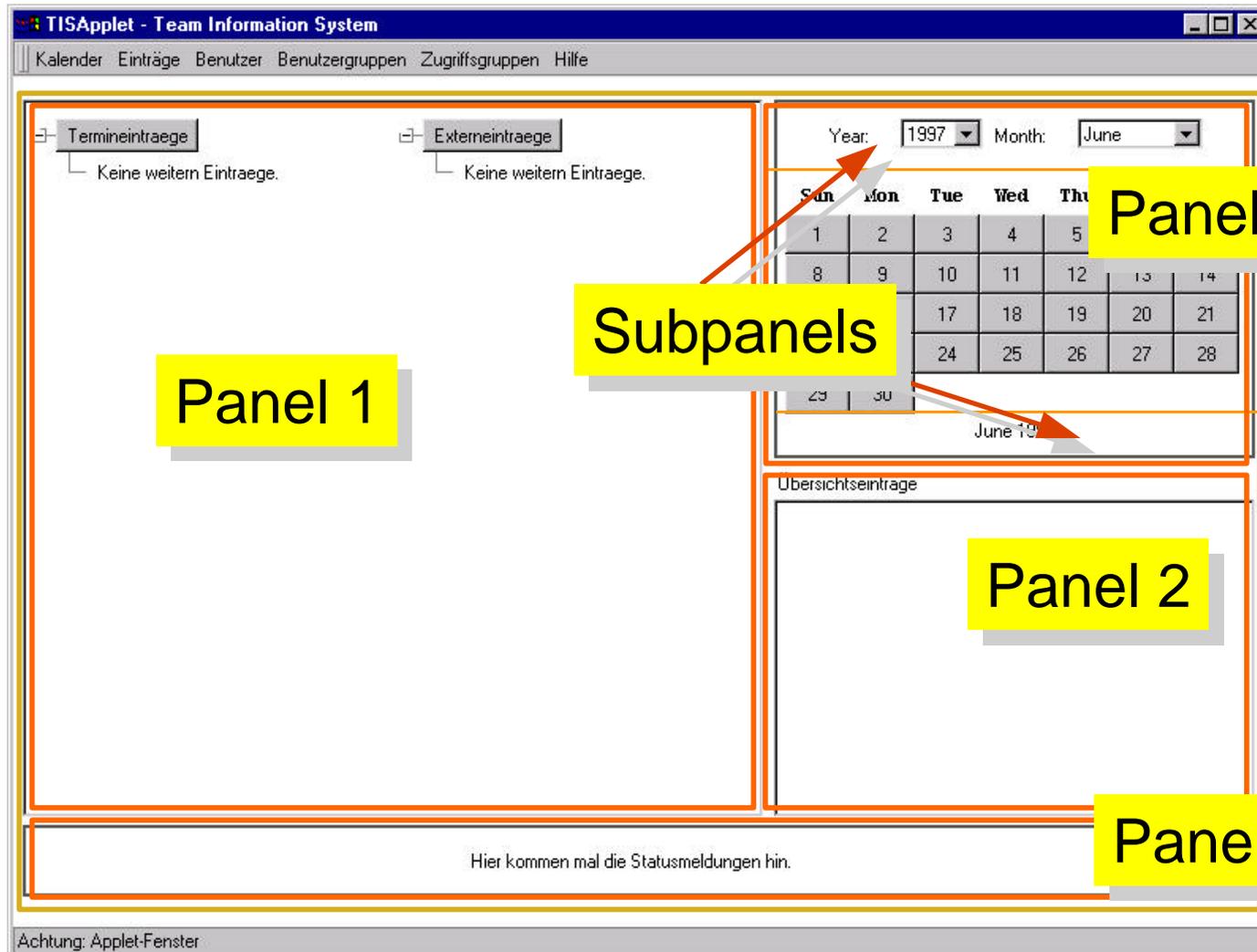
```
Panel p2 = new Panel();
p1.setBackground( Color.red );
p1.add( new Checkbox("eins") );
p1.add( new Checkbox("zwei") );
```

```
add( knopf );
add( p1 );
add( p2 );
```



FlowLayout: zeilenweise von links

Layout & Panels



- Pro Gruppierungseinheit ist ein anders Layout-Prinzip möglich
- Jedes Panel hat seinen eigenen Layout-Manager
- Immun gegen "Resizing"

- Wie werden die Benutzerinteraktionen...
 - » Schaltflächen auslösen
 - » Mausbewegungen und Mausclicks
 - » Schieber bewegen, usw.
- ... umgesetzt in entsprechende Reaktionen der Applikation?
 - » Schaltflächenaktionen ausführen
 - » Mauszeigeränderungen und Elementselektion
 - » Werte der Anwendung ändern

Ereignisbehandlung

- Nach der Erzeugung eines Schaltflächenelements können sich Interessenten für das Auslöseereignis registrieren

```

Button b = new Button("OK");
b.addActionListener( new Eventhandler( ) );
...
    
```

Registrierung des **Ereignis-Empfängers** bei der Komponente, welche die Action-Events auslöst

```

class Eventhandler implements ActionListener {
    public void actionPerformed((ActionEvent e) {
        // do something
    }
}
    
```

- Wird die Schaltfläche bedient, dann wird bei jedem Abonnementen die **actionPerformed** Methode aufgerufen

Mehrfachregistrierung

- Ein Ereignis (Schaltflächen-auslösung) kann von mehreren Abonnements ausgewertet werden



Kopiert Nachricht nur in das obere Fenster

```

Button bla = new Button("Blah blah blah");
Button yds = new Button("You don't say!");
bla.addActionListener( new Eventhandler1( ) );

yds.addActionListener( new Eventhandler1( ) );
yds.addActionListener( new Eventhandler2( ) );
...
    
```

Kopiert Nachricht in beide Fenster

- Je nach Ereigniskategorie wird ein entsprechender Ereignisbehandler benötigt
 - » **MouseListener**: Mausaktionen
 - » **MouseMotionListener**: Mausbewegungen
 - » **WindowListener**: Fensteraktivitäten
 - » **ActionListener**: Aktivierungen (Button, Listenauswahl)
 - » **AdjustmentListener**: Veränderung von Werten (Scrollbar)
 - » **TextListener**: Änderung von Texteinträgen
- Ereignisbehandler entsprechen dem **Controller** der MVC Architektur

Mehrere Ereignisarten pro Ereigniskategorie

```
public interface MouseListener extends EventListener
```

Method Index

```
mouseClicked(MouseEvent)
```

Invoked when the mouse has been clicked on a component

```
mouseEntered(MouseEvent)
```

Invoked when the mouse enters a component

```
mouseExited(MouseEvent)
```

Invoked when the mouse exits a component

```
mousePressed(MouseEvent)
```

Invoked when a mouse button has been pressed on a component

```
mouseReleased(MouseEvent)
```

Invoked when a mouse button has been released on a component



Eventhandler Definition

● Problem

- » wenn man nur eine Ereignisreaktion (z.B., `mouseClicked`) definieren will, muß der Eventhandler trotzdem alle Ereignisarten (leer) implementieren, um die Listener- (hier `MouseListener`) Schnittstelle zu erfüllen

● Lösung

- » Der EventHandler erbt leer definierte Methoden

```
public class MouseAdapter implements MouseListener {  
    public void mouseClicked(MouseEvent e) { }  
    public void mouseEntered(MouseEvent e) { }  
    ...  
}
```

```
class MouseEventHandler extends MouseAdapter {  
    public void mouseClicked( MouseEvent e ) {  
        System.out.println("Mouse button click");  
    }  
}
```

Eventhandler Definition

- Problem

- » oft muß der Handler auf das Modell oder die Sicht zugreifen
- » bei separiertem Handler sind entspr. Referenzen nötig

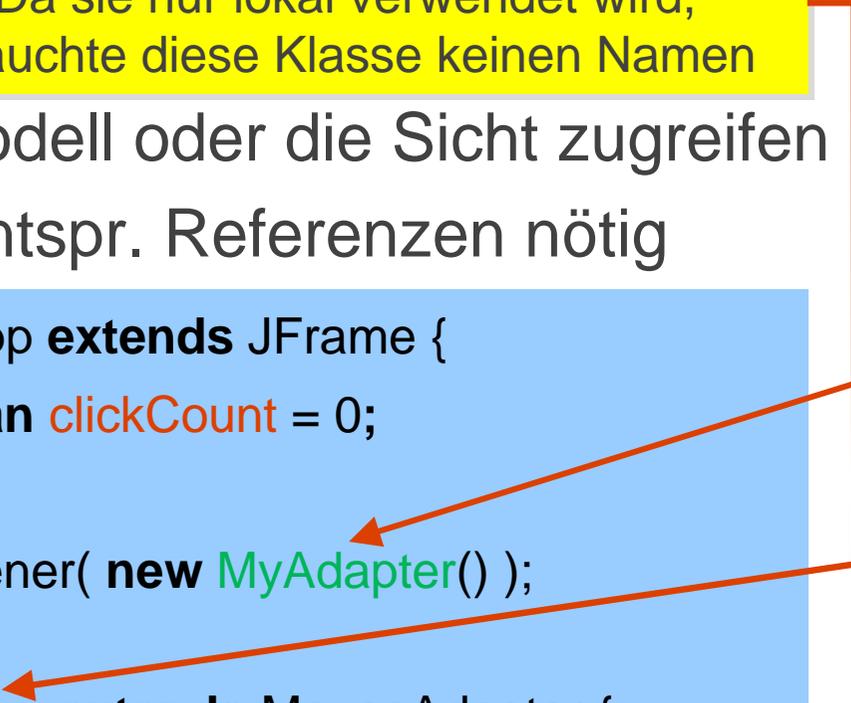
Da sie nur lokal verwendet wird, bräuchte diese Klasse keinen Namen

- Lösung

- » Inner-Classes:
Die Event-Handling Klasse wird Teil des Modells oder der Sicht

```

class MyWindowApp extends JFrame {
    private boolean clickCount = 0;
    ...
    addMouseListener( new MyAdapter() );
    ...
    class MyAdapter extends MouseAdapter {
        public void mouseClicked( MouseEvent e ) {
            clickCount++; }
    } // inner class MyAdapter mit Zugriff auf clickCount
}
    
```





Anonymer Eventhandler

```
public View (Model model)           // accept model to observe
{
    super("Swing Example");           // set window title
    setSize(300, 170);                // set window size

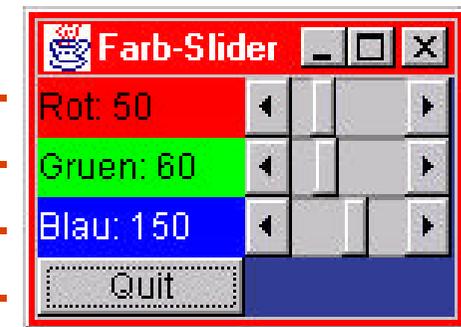
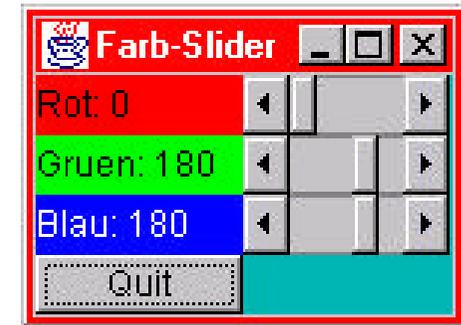
    // make window react to close box event
    addWindowListener(
        new WindowAdapter() {
            public void windowClosing(WindowEvent e) {
                System.exit(0);
            }
        }
    );
    ...
}
```

Schliessen der
Sicht soll die
Anwendung
beenden

Definition einer anonymen Klasse, die alle
WindowAdapter Methoden enthält, plus
die hier definierte **windowClosing** Methode

Ziel

- Drei Slider bestimmen die Farbe eines Feldes
 - » Einsatz von drei Scrollbars mit den entsprechenden Eventhandlern
- Schaltfläche zum Abbruch
 - » Einsatz eines Buttons mit entsprechendem Eventhändler
- Layout wie rechts sichtbar
 - » Benutzung von `GridLayout(4, 2)`



MVC in einer Klasse

```
public class FarbSlider implements ActionListener {
```

```
    protected Frame f;
```

```
    protected Scrollbar sRed, sGreen, sBlue;
```

```
    protected Label IRed, IGreen, IBlue;
```

```
    protected int r, g, b;
```

RGB-Farbwerte des Panel-Hintergrunds

```
public FarbSlider( ) {
```

```
    f = new Frame( "Farb-Slider" );
```

```
    f.setLayout( new GridLayout( 4, 2 ) );
```

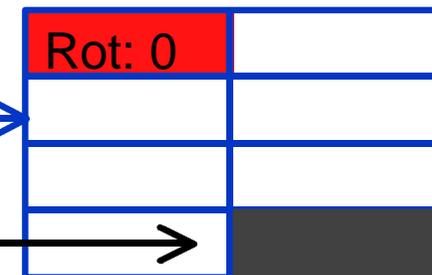
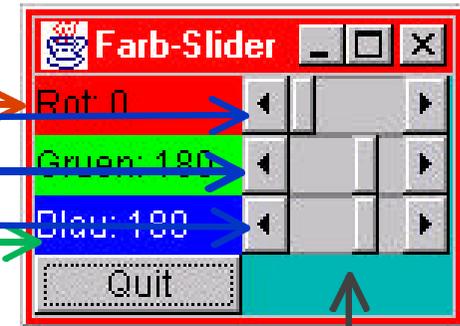
```
    f.setBackground( Color.black );
```

```
    r = g = b = 0;
```

```
    IRed = new Label( "Rot: 0" );
```

```
    IRed.setBackground( Color.red );
```

```
    f.add( IRed );
```



MVC in einer Klasse

- Fortsetzung des FarbSlider-Konstruktors

```

...
sRed = new Scrollbar(Scrollbar.HORIZONTAL, 0, 1, 0, 256);
sRed.addAdjustmentListener( new RedSliderListener( ) );
f.add( sRed );
... // etc. Mit Grün und Blau ...

```

```

q = new Button( "Quit" );
q.addActionListener( this );
f.add( q );

```

```

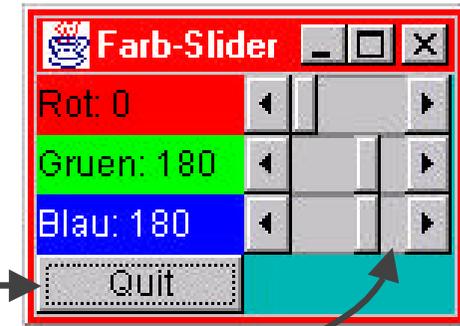
f.setSize( 150, 100 );
f.setVisible( true );

```

```

}

```



Für jeden Scrollbar einen eigenen Handler

Klasse selbst definiert die Ereignisbehandlungsmethode

Anzeigen des Fensters und seiner Komponenten

MVC in einer Klasse

- Innerhalb der FarbSlider-Klasse

```
public void actionPerformed(ActionEvent e) {
    System.out.println( "Ciao!" );
    System.exit( 0 );
}
```

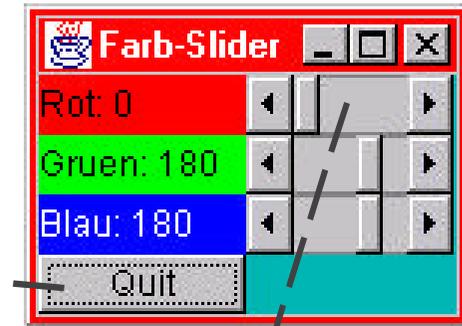
Beenden des Programmlaufs

```
protected class RedSliderListener implements AdjustmentListener {
    public void adjustmentValueChanged( AdjustmentEvent e ) {
        r = e.getValue( );
        lRed.setText( "Rot: " + r );
        f.setBackground( new Color( r, g, b ) );
    }
}
```

Sliderposition

Labeltext
auffrischen

Farbe des Feldes (Hintergrunds) setzen



Wieviele Klassen?

- Eine Applikation mit **Daten**, **grafischer Darstellung** und **Reaktionen** auf Benutzerereignisse lässt sich in ein bis drei (oder mehr) Klassen realisieren
 - » eine Klasse: sehr einfach Kommunikation, schlechte Trennung
 - » zwei Klassen: Trennung von Model & Sicht, Controller jeweils wo sinnvoll
 - » drei Klassen: alles gut getrennt, Kommunikation aufwendiger



Javadoc-Dokumentation

Dokumentationsgeneration aus Code

- Erzeugen der Klassendokumentation in HTML
- Lesen des Inhalts von speziellen Kommentaren: `/** .. */`
- Die gesamte Java API ist in dieser Form dokumentiert

Aufruf:

```
javadoc [options] [packages] [sourcefiles] [classnames]
```

Sichtbarkeitslevel

Zielverzeichnis

HTML-Titel

Beispiel:

```
javadoc -package -d doc -windowtitle "Doku" ..\*.java  
kareltherobot\*.java
```

Quellpfade



Javadoc-Dokumentation

● Vorteile

- » Signaturen werden direkt aus dem Code generiert
- » Code und Dokumentation können leichter synchron gehalten werden
- » → Verwendung spezieller Tags:

<code>@see <classname></code>	Referenz auf eine andere Klasse
<code>@author <author></code>	Autor des Codes
<code>@param <name> <description></code>	Methodenparameter
<code>@return <description></code>	Beschreibung des Rückgabewerts
<code>@exception <name> <description></code>	Beschreibung der Ausnahme



Dokumentation im Code

```
/** Berechnet die Quadratzahl des Arguments.  
    @author <A HREF="http://...">Ulrik Schroeder</A>  
    @version Februar 1999  
    @see java.lang.Integer  
*/  
public class Square {  
  
    /** Einziger Eintrittspunkt der Klasse (das Hauptprogramm)  
        @param args die Programmargumente (jeweils String)  
        @return kein Return-Wert  
        @exception java.lang.NumberFormatException  
            falls das Argument nicht als ganze Zahl interpretierbar ist  
        @exception java.lang. ArrayOutOfBoundsException  
            falls kein Argument angegeben wurde  
    */  
public static void main( String args[ ] )  
    throws NumberFormatException, ArrayOutOfBoundsException {  
        Integer iobj = new Integer( args[ 0 ] );  
        int i = iobj.intValue( );  
        System.out.println( „Square: " + (i*i) );  
    }  
}
```

HTML möglich

Hyperlinks zu anderen Klassen/Methoden

Kommentare zuerst

Mehrfache Nutzung von Tags

HTML-Ergebnis

```

Class Square - Netscape
File Edit View Go Communicator Help

public class Square extends Object
Berechnet die Quadratzahl des Arguments.

Version:
  Februar 1999
Author:
  Ulrik Schroeder
See Also:
  Integer

main

public static void main(String args[]) throws NumberFormatException:

  Einziger Eintrittspunkt der Klasse (das Hauptprogramm)
Parameters:
  args - die Programmargumente (jeweils String)
Returns:
  kein Return-Wert
Throws: NumberFormatException
  falls das Argument nicht als ganze Zahl interpretierbar ist
Throws: ArrayIndexOutOfBoundsException
  falls kein Argument angegeben wurde
    
```

Klassenkomentar

@version tag

@author tag

Signatur



Feedback

Lob & Kritik

- Was war gut?
 - » Was hat gefallen/geholfen?
 - » Was soll beibehalten werden?
- Was war schlecht?
 - » Was war nicht optimal?
 - » Was könnte verbessert werden?
 - » Möglichst mit Verbesserungsvorschlägen
- Anonyme Meinungsabgabe:

<http://www-agce.informatik.uni-kl.de/~kuehne/pm-feedback.html>