Grundlagen der Programmierung II
SS 2005

Dr. Michael Thies

Ziele der Vorlesung

Ziele der Vorlesung Grundlagen der Programmierung II

Die Studierenden sollen

- **graphische Bedienungsoberflächen** mit objektorientierten Techniken entwickeln können,
- die Grundlagen **paralleler Prozesse und deren Synchronisation** verstehen und parallele Prozesse in Java programmieren lernen.
- ihre Kenntnisse in der objektorientierten Programmierung in Java festigen und verbreitern.

Voraussetzungen aus Grundlagen der Programmierung I:

Die Studierenden sollen

- die **Programmentwicklung in Java von Grund auf erlernen**.
- lernen, Sprachkonstrukte sinnvoll und mit **Verständnis** anzuwenden.
- grundlegende **Konzepte der objektorientierten Programmierung** verstehen und anzuwenden lernen. Objektorientierte Methoden haben zentrale Bedeutung im **Software-Entwurf** und in der **Software-Entwicklung**.
- lernen, **Software aus objektorientierten Bibliotheken wiederzuverwenden**.
- **eigene praktische Erfahrungen** in der Entwicklung von **Java**-Programmen erwerben. Darauf bauen größere praktische Entwicklungen in Java oder anderen Programmiersprachen während des Studiums und danach auf.

Inhalt

Nr. d. Vorl.	*Inhalt*	*Abschnitte in „Java lernen, 2. Auflage“*
1	1. Einführung, GUI, Swing (AWT)	10.1
2	2. Zeichenflächen	10.2
3	3. Komponenten erzeugen und platzieren	10.3
4	4. Hierarchisch strukturierte Fensterinhalte	
	5. Ereignisse an graphischen Benutzungsoberflächen	10.3, 11.1
5	Eingabeverarbeitung	11.1
6	6. Beispiel: Ampelsimulation	10.5
7	7. Entwurf von Ereignisfolgen	11.4
8	8. Model/View-Paradigma für Komponenten	—
9	9. Java-Programme in Applets umsetzen	12.1, 12.2
10	10. Parallele Prozesse, Grundbegriffe, Threads	13.1, 13.2
11	11. Unabhängige parallele Prozesse,	13.1, 13.2
12	12. Monitore, Synchronisation gegenseitiger Ausschluss	13.3
13	13. Bedingungssynchronisation im Monitor	
14	14. Verklemmungen, Beispiel: Dining Philosophers	
15	15. Zusammenfassung	

Literaturhinweise

Elektronisches Skript zur Vorlesung:

- **M. Thies: Vorlesung GP II, 2005, http://ag-kastens.upb.de/lehre/material/gpii**
- **U. Kastens: Vorlesung SWE II, 2004, http://ag-kastens.upb.de/lehre/material/sweii**
- **U. Kastens: Vorlesung SWE, 1998/99 (aktualisiert), http://…/swei**

fast ein Textbuch zur Vorlesung, mit dem Vorlesungsmaterial (in älterer Version) auf CD:

- **J. M. Bishop: Java lernen, Addison-Wesley, 2. Auflage, 2001**
- **J. M. Bishop: Java Gently - Programming Principles Explained, Addison-Wesley, 1997 3rd Edition (Java 2)**

zu allgemeinen Grundlagen der Programmiersprachen in der Vorlesung:

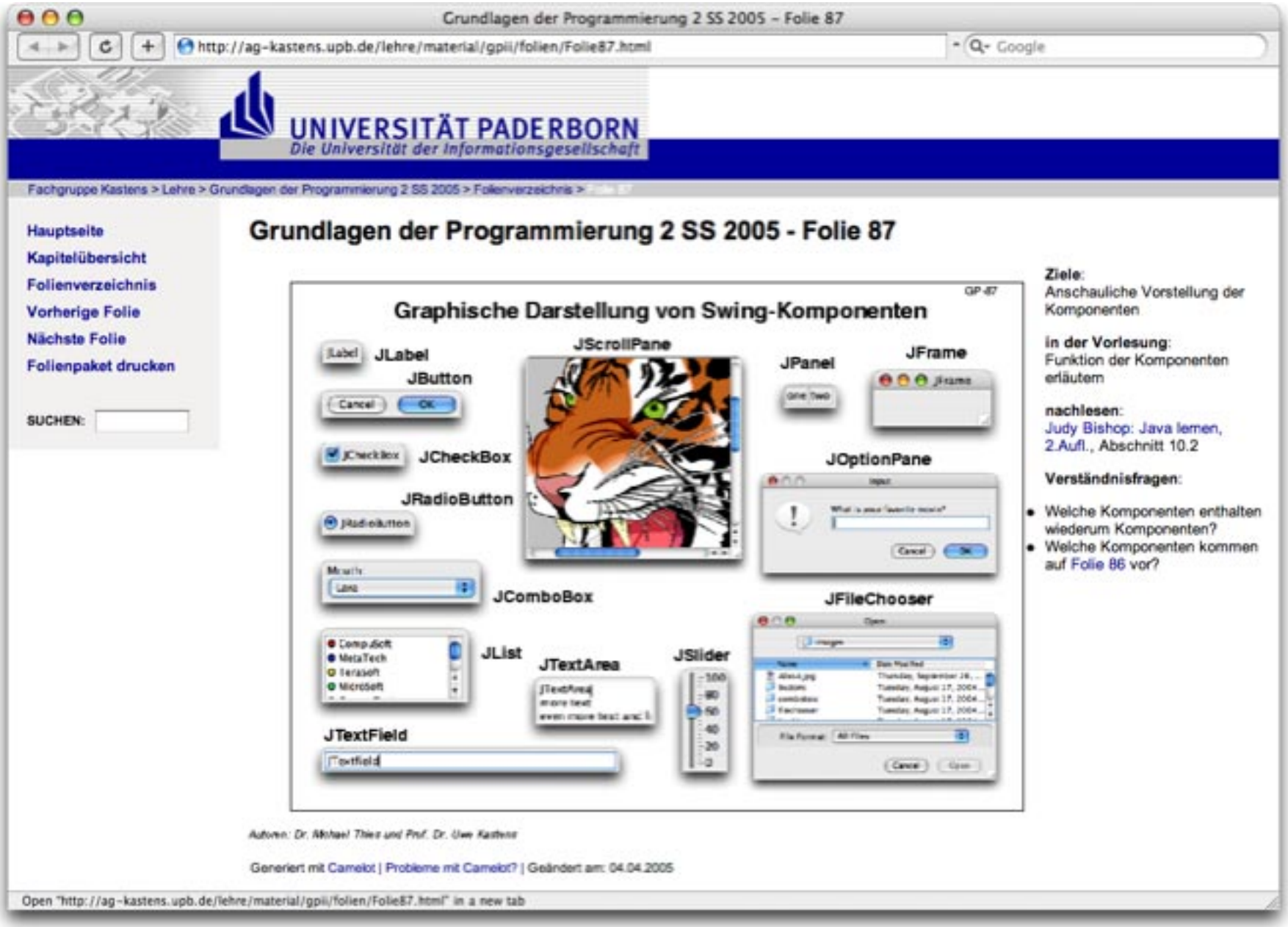
- **U. Kastens: Vorlesung Grundlagen der Programmiersprachen, Skript, 2003 http://ag-kastens.upb.de/lehre/material/gdp**
- **D. A. Watt: Programmiersprachen - Konzepte und Paradigmen, Hanser, 1996**

eine Einführung in Java von den Autoren der Sprache:

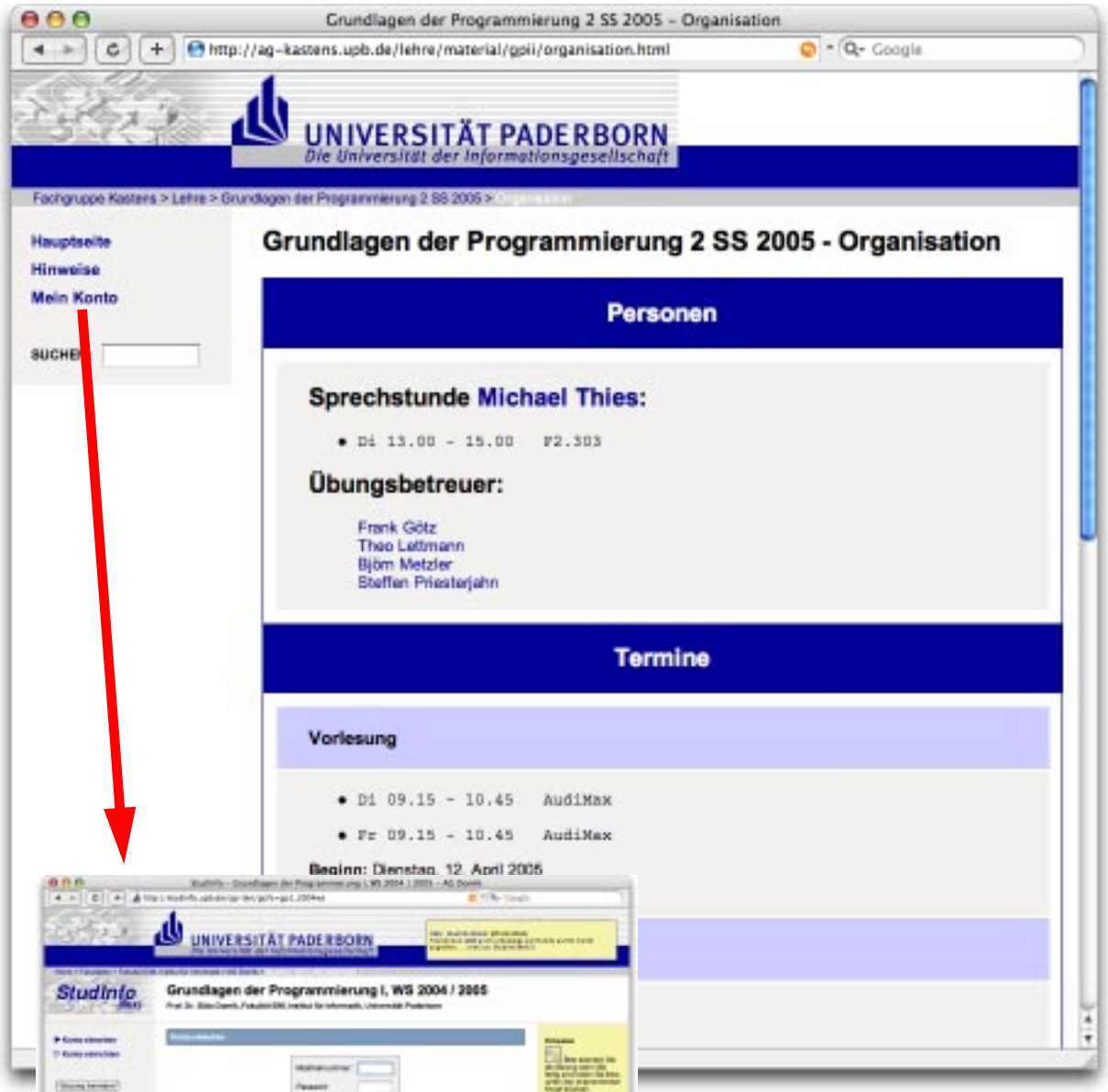
- **Arnold, Ken / Gosling, James: The Java programming language, Addison-Wesley, 1996.**
- **Arnold, Ken / Gosling, James: Die Programmiersprache Java™, 2. Aufl. Addison-Wesley, 1996**

Elektronisches Skript: Startseite

Elektronisches Skript: Folien im Skript



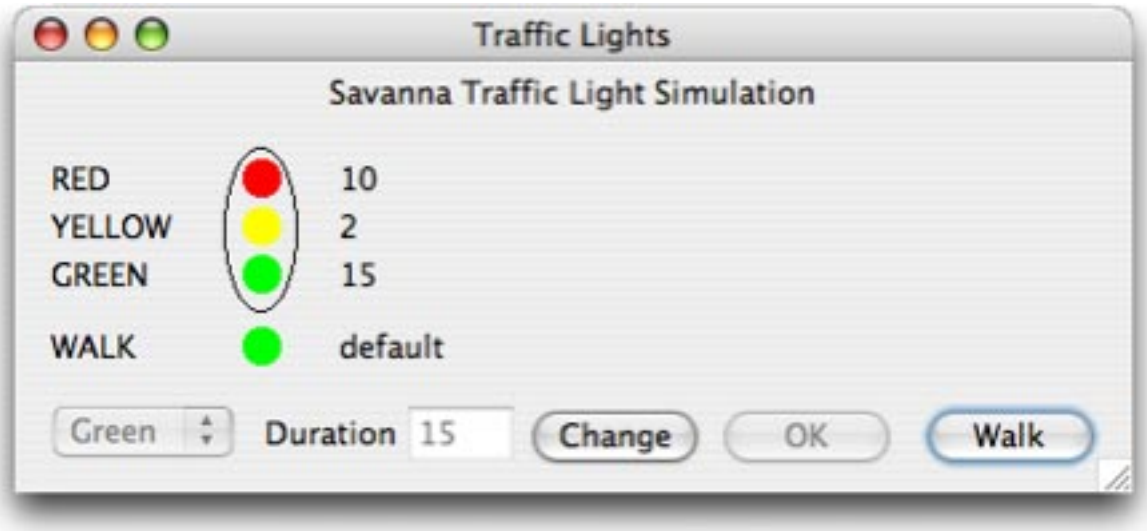
Elektronisches Skript: Organisation der Vorlesung



1. Einführung in graphische Benutzungsoberflächen

Graphische Benutzungsoberflächen (graphical user interfaces, **GUI**) dienen zur

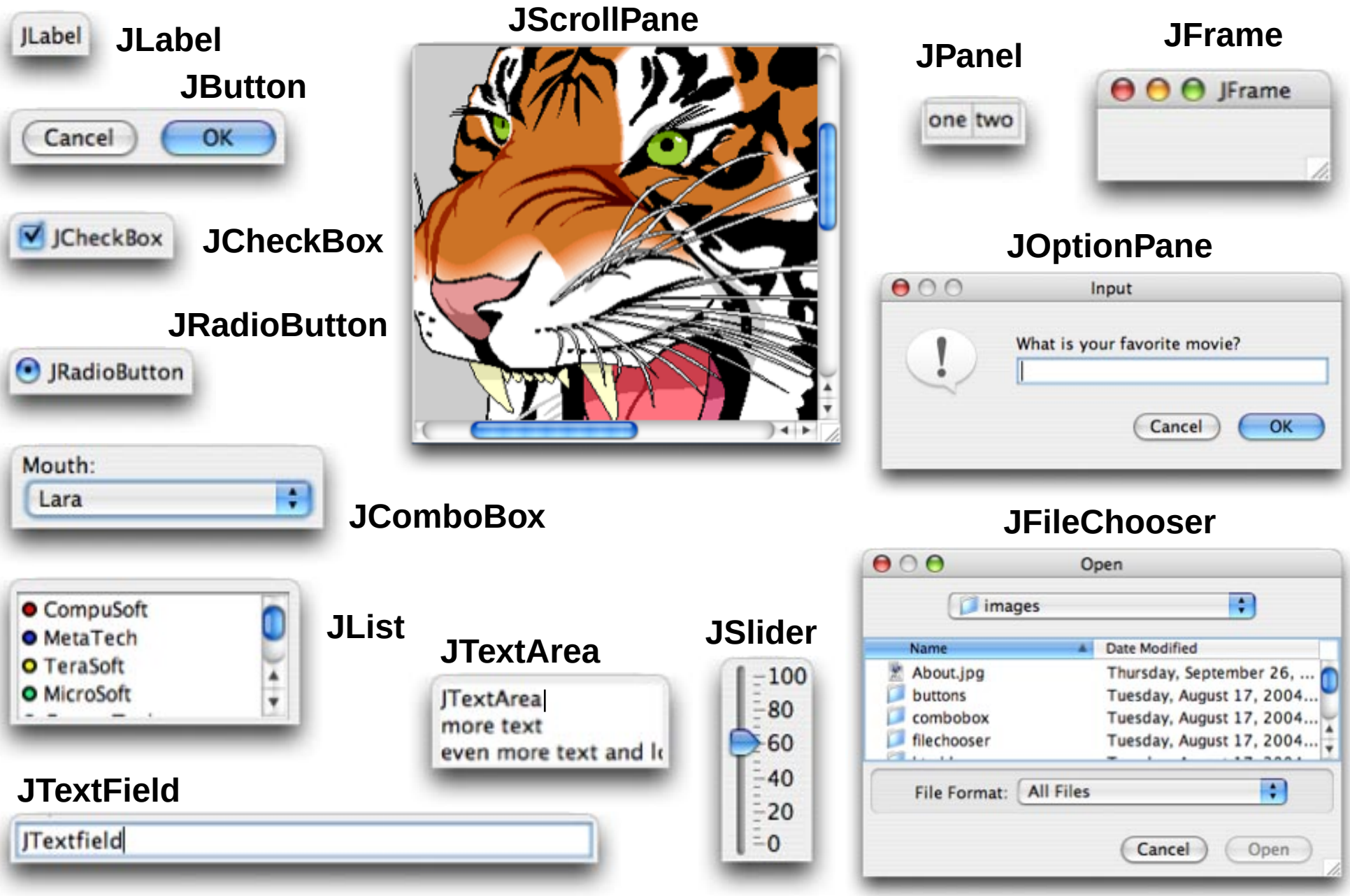
- interaktiven Bedienung von Programmen,
- Ein- und Ausgabe mit graphischen Techniken und visuellen Komponenten



Javas Standardbibliothek `javax.swing`
(Java foundation classes, JFC) enthält wiederverwendbare Klassen zur Implementierung und Benutzung der wichtigsten GUI-Komponenten:

- Graphik
- GUI-Komponenten (siehe GP-87)
- Platzierung, Layoutmanager
- Ereignisbehandlung (**`java.awt.event`**)
- baut auf dem älteren AWT (abstract windowing toolkit) auf

Graphische Darstellung von Swing-Komponenten

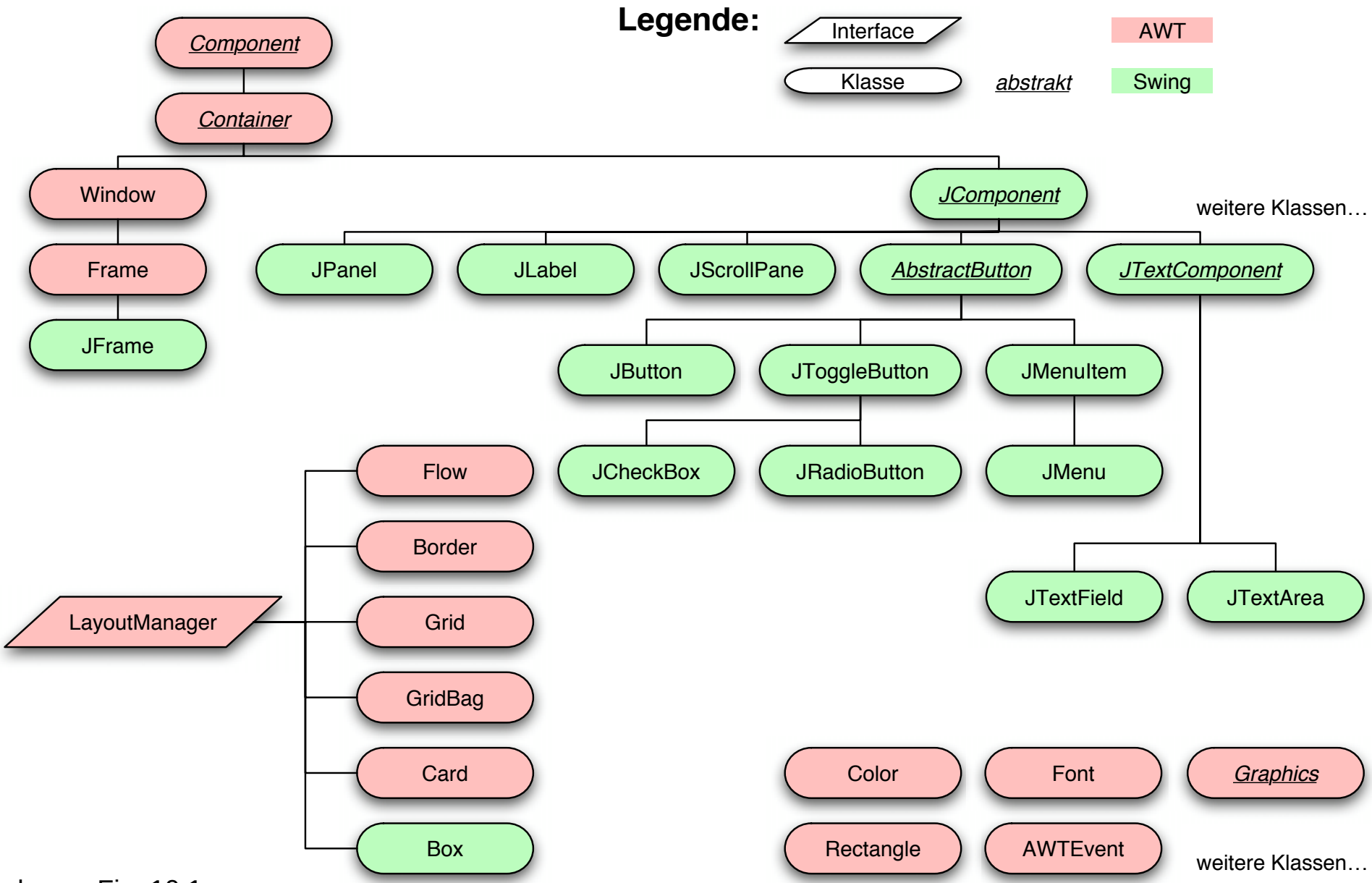


Klassenhierarchie für Komponenten von Benutzungsoberflächen

Teil der erweiterten Standardbibliothek **`javax.swing`** (Java foundation classes, JFC)

Klasse in der Hierarchie	Kurzbeschreibung
Component (abstrakt, AWT)	darstellbare Komponenten von Benutzungsoberflächen
Container (abstrakt, AWT)	Behälter für Komponenten
Window (AWT)	Fenster (ohne Rand, Titel, usw.); Wurzel der Objektbäume
Frame (AWT)	Fenster mit Rand, Titel, usw.
JFrame	Swing-Fenster mit Rand, Titel, usw.
JComponent (abstrakt)	darstellbare Swing-Komponenten
JPanel	konkrete Klasse zu Container, Behälter für Komponenten
JScrollPane	Sicht auf große Komponente, 2 Rollbalken
JFileChooser	Fenster zur interaktiven Dateiauswahl
AbstractButton (abstr.)	Komponenten, die auf einfachen Klick reagieren
JButton	Schaltfläche (“Knopf”)
JToggleButton	Komponenten mit Umschaltverhalten bei Klick
JCheckBox	An/Aus-Schalter (“Ankreuzfeld”), frei einstellbar
JRadioButton	An/Aus-Schalter, symbolisiert gegenseitigen Ausschluß
JComboBox	Auswahl aus einem Aufklappmenü von Texten
JList	Auswahl aus einer mehrzeiligen Liste von Texten
JSlider	Schieberegler zur Einstellung eines ganzzahligen Wertes
JLabel	Textzeile zur Beschriftung, nicht editierbar
JTextComponent (abstr.)	edierbarer Text
JTextField	einzelne Textzeile
JTextArea	mehrzeiliger Textbereich

Klassenhierarchie der Swing-Bibliothek



analog zu Fig. 10.1 aus
Java Gently, 3rd ed, p. 385

Wiederverwendung von Klassen aus Bibliotheken

Die Klasse **`javax.swing.JFrame`** implementiert **gerahmte Fenster** in graphischen Benutzungs-oberflächen (GUI). Sie ist eine Blattklasse in der Hierarchie der GUI-Komponenten:

`java.lang.Object`	
`java.awt.Component`	GUI-Komponenten
`java.awt.Container`	solche, die wieder Komponenten enthalten können
`java.awt.Window`	Fenster ohne Rahmen
`java.awt.Frame`	Fenster mit Rahmen (AWT)
`javax.swing.JFrame`	Fenster mit Rahmen (Swing)

Methoden zum Zeichnen, Platzieren, Bedien-Ereignisse Behandeln, etc. sind auf den jeweils passenden Hierarchieebenen implementiert.
In der **`abstract class Component`** ist die Methode

```
public void paint (Graphics g)
```

definiert, aber nicht ausgefüllt. Mit ihr wird auf der Fläche des Fensters gezeichnet.

Benutzer definieren Unterklassen von `JFrame`, die die Funktionalität der Oberklassen erben. Die Methode **`paint`** **wird überschrieben** mit einer Methode, die das Gewünschte zeichnet:

```
public class Rings extends JFrame
{  public Rings () { super("Olympic Rings"); setSize(300, 150); }
   public void paint (Graphics g) { /* draw olympic rings ... */ }
   public static void main (String[] args)
   {  JFrame f = new Rings(); ... }
}
```

Einige Eigenschaften auf den Ebenen der `JFrame`-Hierarchie

Klasse	**Datenelemente**	**Ereignisse**	**Methoden**
Component	Location, Size, Bounds, Visible	Key, Mouse, Focus, Component	paint
Container	Layout	Container	add, getComponents, paint
Window	Locale	Window	setVisible, pack, toBack, toFront
Frame	Title, MenuBar, Resizable, IconImage		
JFrame	ContentPane, RootPane, DefaultCloseOperation		

Namenskonventionen:

zum Datenelement *XXX*
gibt es die Methoden
`get`*`XXX`* und ggf. **`set`*`XXX`***

Ereignisse sind Objekte
der Klassen ***`YYY`*`Event`**

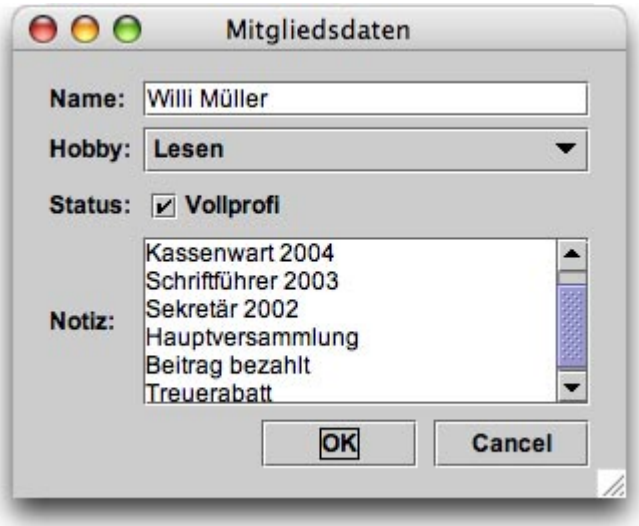
Vergleich: Swing und AWT

	Swing	AWT
zu finden im Java-Paket	**`javax.swing.*`** optionale, aber standardisierte Java Extension seit Java 1.2 bzw. 1.1	**`java.awt.*`** Teil der Java Standard Edition seit Java 1.0
Zeichnen der GUI-Komponenten	in Java implementiert (leichtgewichtige Komp.), nur Fenster vom Betriebssystem verwaltet	durch das Betriebssystem, in *Peer*-Objekten gekapselt (schwergewichtige Komp.)
(visuelle) Rückmeldungen für Benutzeraktionen	in Java implementiert, außer Manipulation ganzer Fenster	durch das Betriebssystem realisiert
Reaktion auf Benutzeraktionen im Programm	in Java implementiert durch sog. *Listener*-Objekte	in Java implementiert durch sog. *Listener*-Objekte
angebotene GUI-Komponenten	einfach bis sehr komplex: z.B. Schaltflächen, Tabellen, Baumstrukturen zum Aufklappen	einfach bis mittel: Schnittmenge über verschiedene Betriebssysteme, z.B. Listen, aber keine Tabellen

Look&Feel-Module in Swing

Es gibt mehrere Sätze von Java-Klassen, die die Swing-Komponenten unteschiedlich grafisch darstellen. Ein vollständiger Satz von Klassen für alle Komponenten bildet ein Look&Feel-Modul:

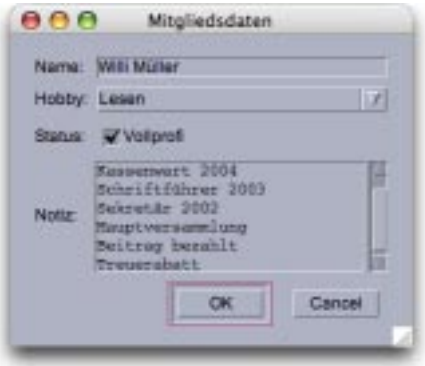
- Look&Feel eines Programms ist beim Start des Programms frei wählbar oder sogar während der Laufzeit unimttelbar umschaltbar.
- Das generische Java-Look&Feel-Modul *Metal* ist auf jedem System verfügbar.
- Hinzu kommen unterschiedlich genaue Imitationen verschiedener Betriebssysteme.
- **Realisierung:** Look&Feel-Modul enthält eine konkrete Unterklasse für jede abstrakte Look&Feel-Klasse in Swing.



Metal

Aqua (MacOS X)

Motif



JGoodies Plastic XP

2. Zeichenflächen benutzen, Programmschema


```
import javax.swing.JFrame; import java.awt.*;
        // Hauptklasse als Unterklasse von JFrame:
public class GraphicWarning extends JFrame
{  GraphicWarning (String title)                          // Konstruktor
   {  super (title);                          // Aufruf des Konstruktors von JFrame









   }

   public void paint (Graphics g)    // überschreibt paint in einer Oberklasse
   {  super.paint(g);                 // Hintergrund der Fensterfläche zeichnen


   }

   public static void main (String[] args)
   {  JFrame f = new GraphicWarning ("Draw Warning"); // Objekt erzeugen
   }
}
```

Programmschema: Eigenschaften und Ereignisbehandlung


```
import javax.swing.JFrame; import java.awt.*;
        // Hauptklasse als Unterklasse von JFrame:
public class GraphicWarning extends JFrame
{  GraphicWarning (String title)                                  // Konstruktor
   {  super (title);                          // Aufruf des Konstruktors von JFrame
      Container content = getContentPane();          // innere Fensterfläche
      content.setBackground (Color.cyan);           // Farbe dafür festlegen
      setSize (35*letter, 6*line);                    // Größe festlegen
      setDefaultCloseOperation (EXIT_ON_CLOSE);
      // Verhalten des Fensters: Beim Drücken des Schließknopfes Programm beenden.
      setVisible (true);                        // führt zu erstem Aufruf von paint
   }
   private static final int line = 15, letter = 5;   // zur Positionierung
   public void paint (Graphics g)    // überschreibt paint in einer Oberklasse
   {  super.paint(g);                 // Hintergrund der Fensterfläche zeichnen



   }

   public static void main (String[] args)
   {  JFrame f = new GraphicWarning ("Draw Warning"); // Objekt erzeugen
   }
}
```

Programmschema: paint-Methode ausfüllen


```
import javax.swing.JFrame; import java.awt.*;
        // Hauptklasse als Unterklasse von JFrame:
public class GraphicWarning extends JFrame
{  GraphicWarning (String title)                                  // Konstruktor
   {  super (title);                          // Aufruf des Konstruktors von JFrame
      Container content = getContentPane();          // innere Fensterfläche
      content.setBackground (Color.cyan);           // Farbe dafür festlegen
      setSize (35*letter, 6*line);                    // Größe festlegen
      setDefaultCloseOperation (EXIT_ON_CLOSE);
      // Verhalten des Fensters: Beim Drücken des Schließknopfes Programm beenden.
      setVisible (true);                        // führt zu erstem Aufruf von paint
   }
   private static final int line = 15, letter = 5;   // zur Positionierung
   public void paint (Graphics g)    // überschreibt paint in einer Oberklasse
   {  super.paint(g);              // Rest der Fensterfläche zeichnen (Hintergrund)
      g.drawRect (2*letter, 2*line, 30*letter, 3*line);// auf der Fläche g
      g.drawString ("W A R N I N G", 9*letter, 4*line); // zeichnen und
   }                                                                  // schreiben

   public static void main (String[] args)
   {  JFrame f = new GraphicWarning ("Draw Warning"); // Objekt erzeugen
   }
}
```

Ablauf des Zeichen-Programms

1. **`main`** aufrufen:
    1.1. **`GraphicWarning`**-Objekt erzeugen, Konstruktor aufrufen:
        1.1.1 **`JFrame`**-Konstruktor aufrufen
        1.1.2 **`Container`**-Objekt für innere Fensterfläche abfragen**`,`**
        1.1.3 Eigenschaften setzen, z. B. Farben,
        1.1.4 ggf. weitere Initialisierungen des Fensters
        1.1.5 Größe festlegen, **`setSize(…, …)`**,
        1.1.6 Verhalten des Schließknopfes festlegen, **`setDefaultCloseOperation`**,
        1.1.7 Fenster sichtbar machen, **`setVisible(true)`**
            parallele Ausführung von (2) initiieren
    1.2 Objekt an **`f`** zuweisen
    1.3 ggf. weitere Anweisungen zur Programmausführung

in Methoden der Oberklassen:

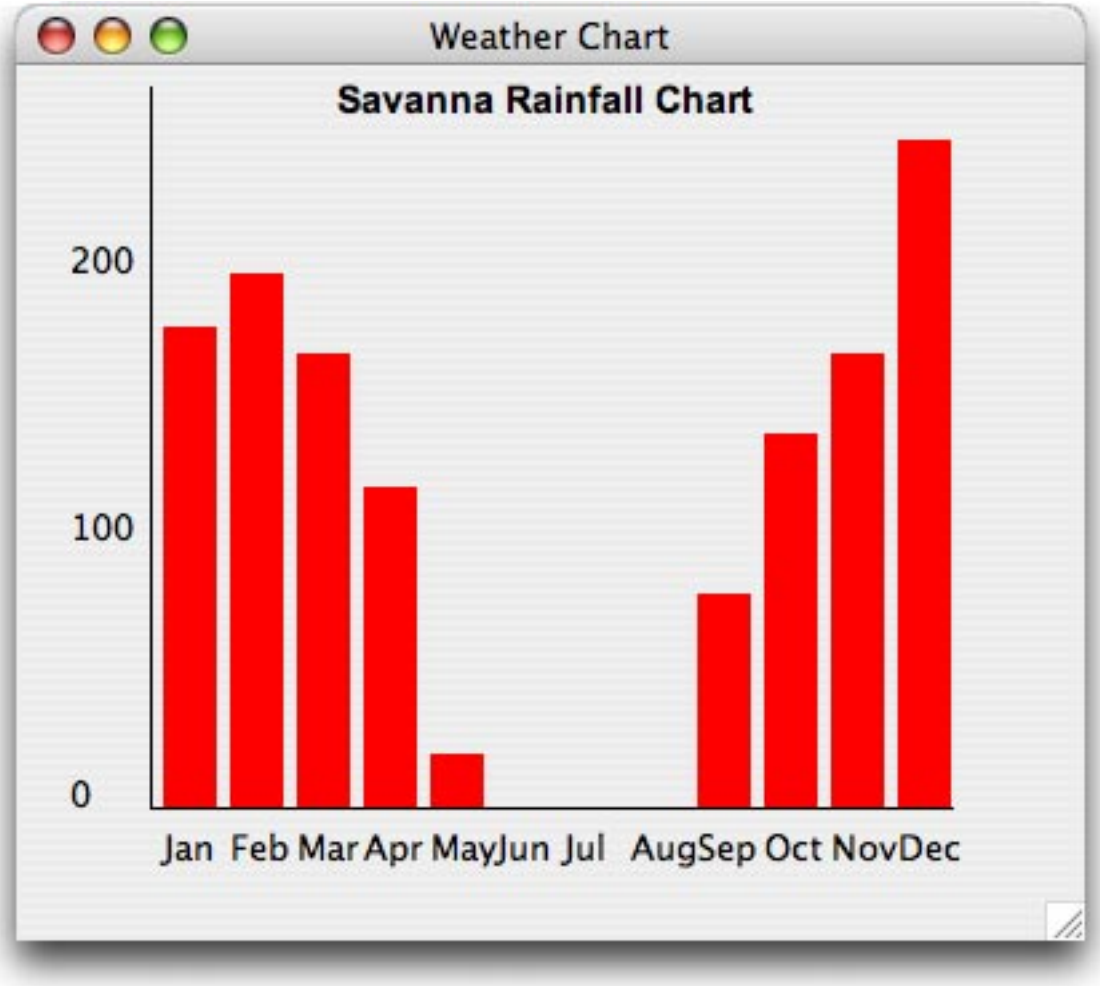
2. **`Graphics`** Objekt erzeugen
    2.1 damit erstmals **`paint`** aufrufen (immer wieder, wenn nötig):
    weiter in **`paint`** von **`GraphicWarning`**
        2.1.1 Methode aus Oberklasse den Hintergrund der Fensterfläche zeichnen lassen
        2.1.2 auf der Zeichenfläche des Parameters **`g`** schreiben und zeichnen

3. Schließknopf Drücken

in Methoden der Oberklassen:

    3.1 Programm beenden, **`System.exit(0)`**

Beispiel: Balkendiagramm zeichnen



Beispiel: Balkendiagramm zeichnen

```
public void paint (Graphics g)
{  super.paint(g);                                  // Hintergrund zeichnen
   int x = 50, y = 300;                            // Schnittpunkt der Achsen
   int width = 20, gap = 5;                        // Balken und Zwischenraum

   g.drawLine (x, y, x+12*(width+gap), y);                    // x-Achse
   g.drawLine (x, y, x, 30);                                  // y-Achse

   for (int m = 0; m < 12; m++)                    // Monate an der x-Achse
      g.drawString(months[m], m*(width+gap)+gap+x, y+20);

   for (int i = 0; i < y; i+=100)                  // Werte an der y-Achse
      g.drawString(String.valueOf(i), 20, y-i);

   g.setFont(new Font("SansSerif", Font.BOLD, 14));          // Überschrift
   g.drawString("Savanna Rainfall Chart", 120, 40);

   g.setColor(Color.red);                                     // die Balken
   for (int month = 0; month < 12; month++)
   {  int a = (int) rainTable[month]*10;
      g.fillRect(month*(width+gap)+gap+x, y-a, width, a);
   }
}
private double[] rainTable = new double[12];
private static String months [] = {"Jan", "Feb", "Mar",…, "Oct", "Nov", "Dec"};
```

3. Swing-Komponenten erzeugen und platzieren

Ein einfaches Beispiel für Text und Schaltknöpfe:

Aufgaben zur Herstellung:

Aussehen:

- JLabel- und JButton-Objekte generieren
- Anordnung der Komponenten festlegen

Ereignisse:

- ein *Listener*-Objekt mit den Buttons verbinden
- *call-back*-Methode für Buttons implementieren

Komponenten platzieren

Jedes Fenster (**JFrame**-Objekt) besitzt ein **Container**-Objekt, das den Inhalt des Fenster aufnimmt, die sogenannte *content pane*.
Die Klasse **Container** sorgt für die Platzierung der Komponenten, die ein **Container**-Objekt enthält.

Dazu wird für den **Container** ein **LayoutManager** installiert; z. B. mit folgender Anweisung im Konstruktor der Unterklasse von **JFrame**:

```
Container content = getContentPane();
content.setLayout (new FlowLayout (FlowLayout.CENTER));
```

Ein **LayoutManager** bestimmt die Anordnung der Komponenten nach einer speziellen Strategie; z. B. **FlowLayout** ordnet zeilenweise an.

Komponenten werden generiert und mit der Methode **add** dem **Container** zugefügt, z. B.

```
content.add (new JLabel ("W A R N I N G")); …

JButton waitButton = new JButton ("Wait");
content.add (waitButton); …
```

Die Reihenfolge der **add**-Aufrufe ist bei manchen **LayoutManager**n relevant.

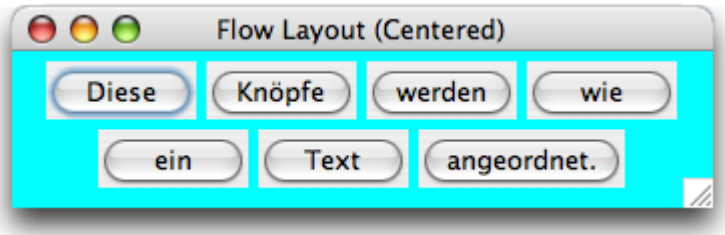
Wird die Gestalt des **Container**s verändert, so ordnet der **LayoutManager** die Komponenten ggf. neu an.

Programmschema zum Platzieren von Komponenten

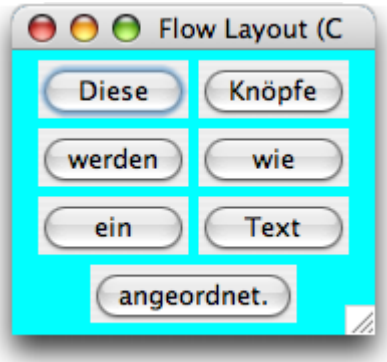
```
import javax.swing.*; import java.awt.*;

class FlowTest extends JFrame                    // Definition der Fenster-Klasse
{  FlowTest ()                                                    // Konstruktor
   {  super ("Flow Layout (Centered)"); // Aufruf des Konstruktors von JFrame
      Container content = getContentPane();          // innere Fensterfläche
      content.setBackground (Color.cyan);          // Eigenschaften festlegen

      content.setLayout(new FlowLayout(FlowLayout.CENTER));
         // LayoutManager-Objekt erzeugen und der Fensterfläche zuordnen

      content.add(new JButton("Diese")); …       // Komponenten zufügen

      setSize(350,100);
      setDefaultCloseOperation (EXIT_ON_CLOSE);
      // Verhalten des Fensters: Beim Drücken des Schließknopfes Programm beenden.
      setVisible(true);
   }
}

public class LayoutTry
{  public static void main(String[] args)
   {  JFrame f = new FlowTest();              // Ein FlowTest-Objekt erzeugen
   }
}
```

LayoutManager FlowLayout

```
class FlowTest extends JFrame
{  FlowTest ()
   {  super("Flow Layout (Centered)");
      Container c = getContentPane(); c.setBackground(Color.cyan);

      c.setLayout(new FlowLayout(FlowLayout.CENTER));

      c.add(new JButton("Diese")); c.add(new JButton("Knöpfe"));
      c.add(new JButton("werden")); c.add(new JButton("wie"));
      c.add(new JButton("ein")); c.add(new JButton("Text"));
      c.add(new JButton("angeordnet."));

      setSize(350,100); setDefaultCloseOperation(EXIT_ON_CLOSE);
      setVisible(true);
   }
}
```

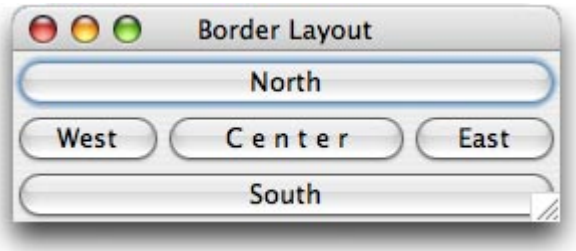


nach Ändern der Gestalt:

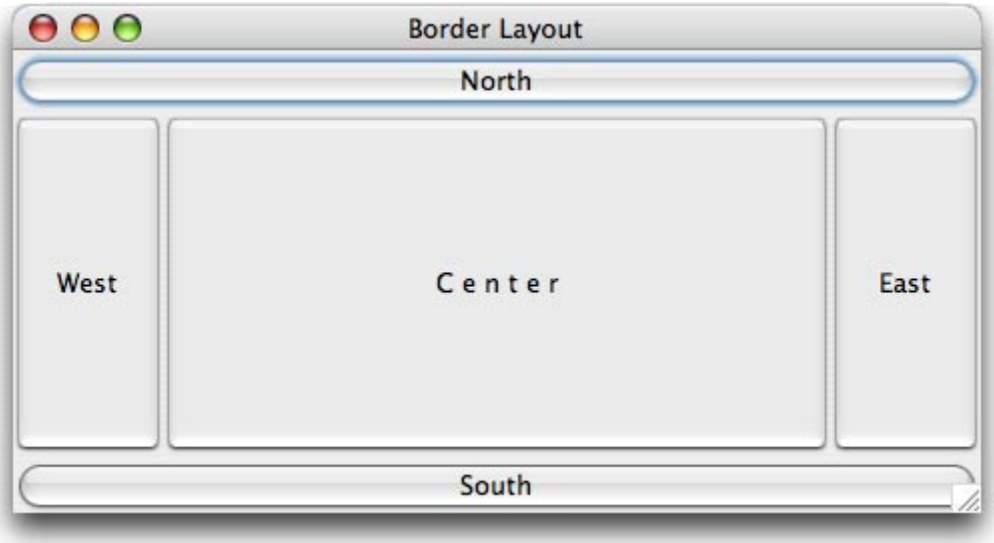


LayoutManager BorderLayout

```
class BorderTest extends JFrame
{  BorderTest ()
   {  super("Border Layout");
      Container c = getContentPane(); c.setBackground(Color.cyan);

      c.setLayout(new BorderLayout());

      c.add(new JButton("North"), BorderLayout.NORTH);
      c.add(new JButton("East"), BorderLayout.EAST);
      c.add(new JButton("South"), BorderLayout.SOUTH);
      c.add(new JButton("West"), BorderLayout.WEST);
      c.add(new JButton("C e n t e r"), BorderLayout.CENTER);
      setSize(250,100); setDefaultCloseOperation(EXIT_ON_CLOSE);
      setVisible(true);
   }
}
```

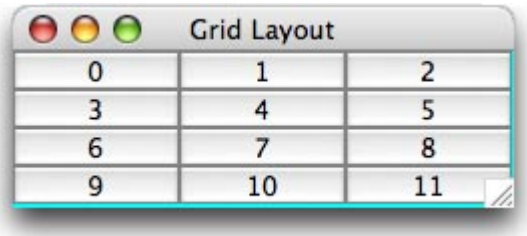


nach Ändern der Gestalt:

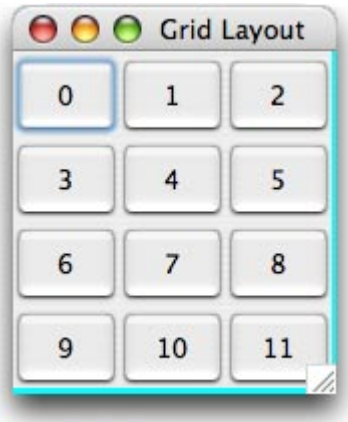


LayoutManager GridLayout

```
class GridTest extends JFrame
{  GridTest ()
   {  super("Grid Layout");
      Container c = getContentPane(); c.setBackground(Color.cyan);

      c.setLayout(new GridLayout(4, 3));

      for (int i = 0; i < 12; i++)
         c.add(new JButton(String.valueOf(i)));

      setSize(250,100); setDefaultCloseOperation(EXIT_ON_CLOSE);
      setVisible(true);
   }
}
```

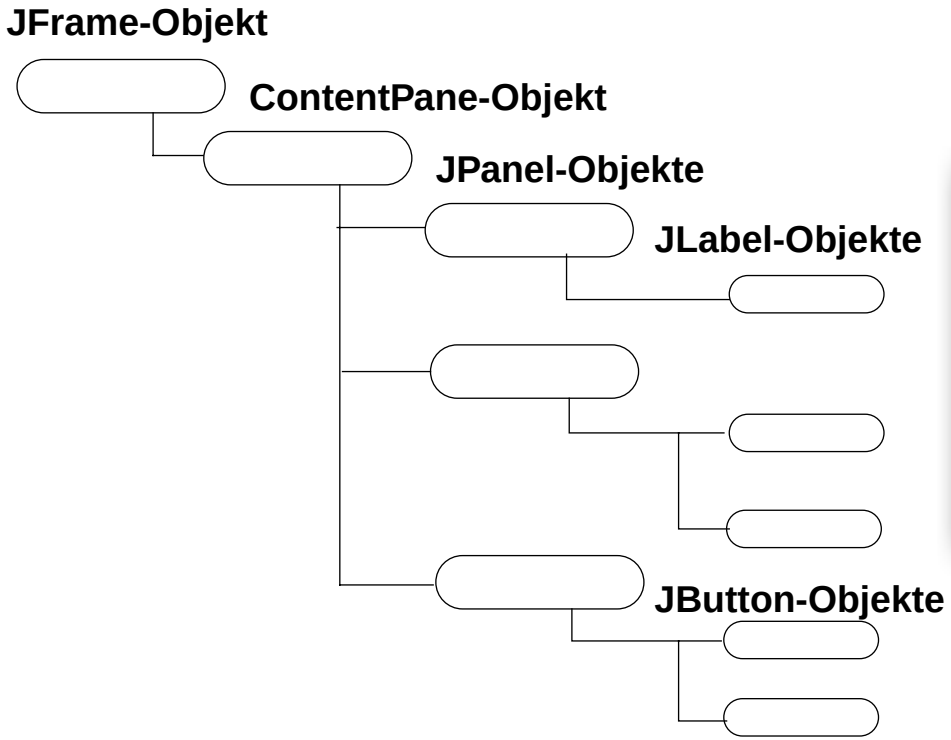


nach Ändern der Gestalt:



4. Hierarchisch strukturierte Fensterinhalte

- **Zusammengehörige Komponenten** in einem Objekt einer **`Container`**-Unterklasse unterbringen (**`JPanel`**, *content pane* eines **`JFrame`**s oder selbstdefinierte Unterklasse).
- Anordnung der im **`Container`** gruppierten Objekte wird dann gemeinsam bestimmt, indem man dem **`Container`** einen geeigneten **`LayoutManager`** zuordnet.
- Mit **`Container`**-Objekten werden beliebig tiefe **Baumstrukturen** von Swing-Komponenten erzeugt. In der visuellen Darstellung sind sie **ineinander geschachtelt**.



Programm zu hierarchisch strukturiertem Fenster

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

class LabelContainer extends JPanel                  // Klasse zur Herstellung von
{  LabelContainer (String[] words)                   // Fließtext aus String-Array
   {  super(new FlowLayout(FlowLayout.CENTER));      // Konstr. der Oberklasse
      for (int i = 0; i < words.length; i++)         // legt LayoutManager fest
         add (new JLabel (words[i]));
   }
}

class LayoutComp extends JFrame   // Fensterkonstr. erzeugt Komponentenbaum
{  LayoutComp (String title)
   {  super(title);
      String[] message = {"Possible", "virus", "detected.", "Reboot",
                          "and", "run", "virus", "remover", "software"};
      JPanel warningText = new LabelContainer (message);

      ... Erzeugung des Komponentenbaumes einfügen ...

      setSize (180, 200); setDefaultCloseOperation(EXIT_ON_CLOSE);
      setVisible(true);
   }

   public static void main (String[] args)
   {  JFrame f = new LayoutComp("Virus Warning"); }
}
```

Komponentenbaum erzeugen

```
// Text der Warnung im Array message zusammengefasst auf LabelContainer:
JPanel warningText = new LabelContainer (message);

// Überschrift als JPanel mit einem zentrierten JLabel:
JPanel header = new JPanel (new FlowLayout(FlowLayout.CENTER));
header.setBackground(Color.yellow);
header.add (new JLabel ("W a r n i n g"));

// Knöpfe im JPanel mit GridLayout:
JPanel twoButtons = new JPanel (new GridLayout(1, 2));
twoButtons.add (new JButton ("Wait"));
twoButtons.add (new JButton ("Reboot"));

// in der Fensterfläche mit BorderLayout zusammenfassen:
Container content = getContentPane();
content.setBackground(Color.cyan);
content.setLayout(new BorderLayout());
content.add(header, BorderLayout.NORTH);
content.add(warningText, BorderLayout.CENTER);
content.add(twoButtons, BorderLayout.SOUTH);
```

5. Ereignisse an graphischen Benutzungsoberflächen

Interaktion zwischen Bediener und Programmausführung über **Ereignisse (***events***)**:

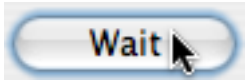
- **Bedien-Operationen lösen Ereignisse aus**, z. B. Knopf drücken, Menüpunkt auswählen, Mauszeiger auf ein graphisches Element bewegen.
- **Programmausführung reagiert auf solche Ereignisse** durch Aufruf bestimmter Methoden

Aufgaben:

- **Folgen von Ereignissen** und Reaktionen darauf **planen und entwerfen**
  Modellierung z. B. mit endlichen Automaten oder StateCharts
- **Reaktionen auf Ereignisse** systematisch implementieren
  Swing: Listener-Konzept; Entwurfsmuster „Observer“ (aus AWT übernommen)

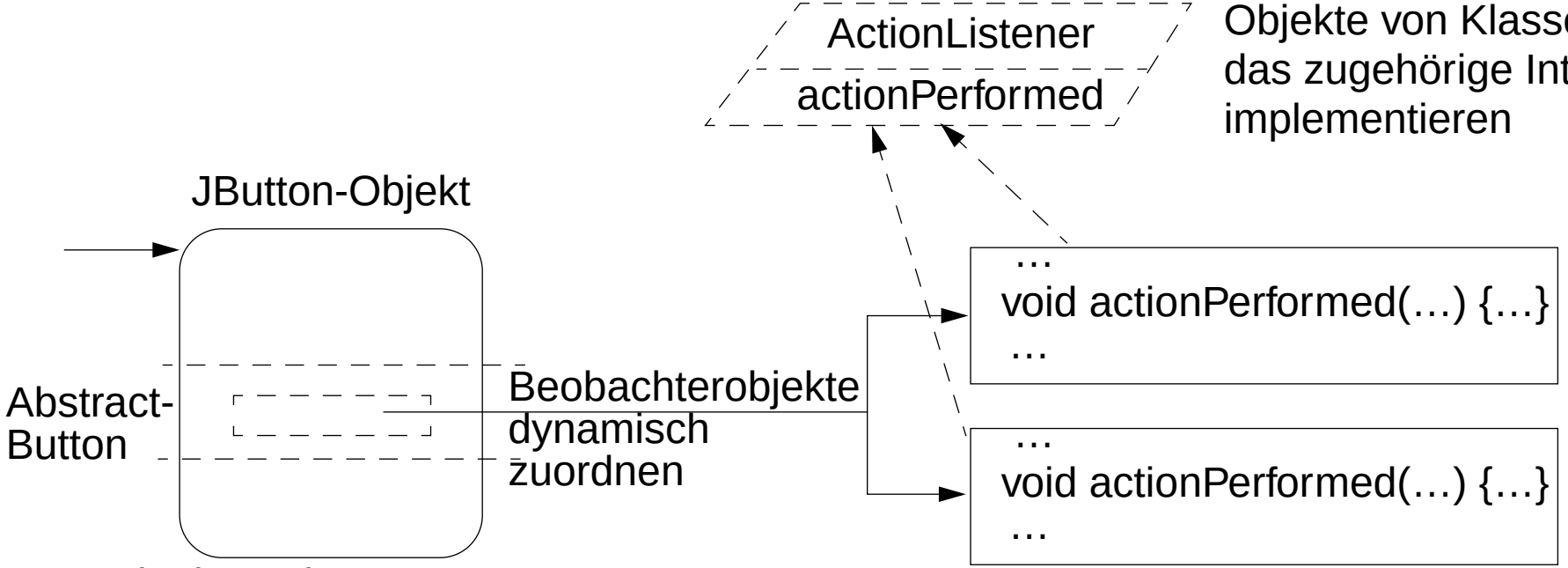
„Observer“-Prinzip in der Ereignisbehandlung

An Swing-Komponenten werden Ereignisse ausgelöst, z. B. ein ActionEvent an einem JButton-Objekt:



“click”

Beobachter (LIstener) für bestimmte Ereignistypen: Objekte von Klassen, die das zugehörige Interface implementieren



Ereignis auslösen: zugehörige Methode in jedem Beobachter-Objekt aufrufen

Methoden implementieren die gewünschten Reaktionen

Entwurfsmuster „Observer“: Unabhängigkeit zwischen den Beobachtern und dem Gegenstand wegen Interface und dynamischem Zufügen von Beobachtern.

Ereignisbehandlung für eine Schaltfläche

Im **`java.awt.event`** Package gibt es zum Ereignistyp **`ActionEvent`** ein Interface **`ActionListener`**:

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

Um auf das Anklicken der Schaltfläche zu reagieren, wird im Programm eine Klasse deklariert, die das **Interface implementiert**. Die **Methode** aus dem Interface wird mit der gewünschten Reaktion **überschrieben**:

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

Von dieser Klasse wird ein Objekt erzeugt und als Beobachter dem Schaltflächen-Objekt zugefügt:

```
JButton quitButton = new JButton("Quit");
quitButton.addActionListener (new ProgramTerminator());
```

Programmiertechnik für `Listener`

Im **`java.awt.event`** Package gibt es zu jedem Ereignistyp **`XXXEvent`** ein Interface **`XXXListener`**:

```
public interface WindowListener extends EventListener
{ void windowActivated (WindowEvent); void windowClosed (WindowEvent);
  void windowClosing (WindowEvent); … void windowOpened (WindowEvent);
}
```

Eine abstrakte Klasse **`XXXAdapter`** mit leeren Methodenimplementierungen:

```
public abstract class WindowAdapter implements WindowListener
{  public void windowActivated (WindowEvent) { } …
   public void windowOpened (WindowEvent) { }
}
```

Anwendungen, die nicht auf alle Sorten von Methodenaufrufen des Interface reagieren, deklarieren eine Unterklasse und **überschreiben die benötigten Methoden des Adapters**, meist als innere Klasse, um den Zustand eines Objektes zu verändern:

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

Zufügen eines Listener-Objektes zu einer Swing-Komponente:

```
f.addWindowListener (new WindowCloser());
```

Innere Klassen

Innere Klassen können z. B. als Hilfsklassen zur Implementierung der umgebenden Klasse verwendet werden:

```
class List { ... static class Node { ...} ...}
```

Die **`List`**-Objekte und **`Node`**-Objekte sind dann **unabhängig voneinander**.

Es wird nur die Gültigkeit des Namens **`Node`** auf die Klasse **`List`** eingeschränkt.

In **inneren Klassen, die nicht `static`** sind, können Methoden der inneren Klasse auf Objektvariable der äusseren Klasse zugreifen. Ein Objekt der inneren Klasse ist dann immer in ein Objekt der äusseren Klasse eingebettet; z. B. die inneren **`Listener`** Klassen, oder auch:

```
interface Einnehmer { void bezahle (int n); }

class Kasse                          // Jedes Kassierer-Objekt eines Kassen-Objekts
{  private int geldSack = 0;                        // zahlt in denselben Geldsack

   class Kassierer implements Einnehmer
   {  public void bezahle (int n)
      {  geldSack += n; }
   }

   Einnehmer neuerEinnehmer ()
   {  return new Kassierer (); }
}
```

Anonyme Klasse

Meist wird zu der Klasse, mit der Implementierung der Reaktion auf einen Ereignistyp **nur ein einziges Objekt** benötigt:

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

Zufügen eines **`Listener`**-Objektes zu einer Swing-Komponente:

```
f.addWindowListener (new WindowCloser());
```

Das läßt sich kompakter formulieren mit einer **anonymen Klasse**:
Die Klassendeklaration wird mit der **`new`**-Operation (für das eine Objekt) kombiniert:

```
f.addWindowListener
   (  new WindowAdapter ()
      {  public void windowClosing (WindowEvent e)
         { System.exit (0); }
      }
   );
```

In der **`new`**-Operation wird der **Name der Oberklasse** der deklarierten anonymen Klasse (hier: **`WindowAdapter`**) **oder** der **Name des Interface**, das sie implementiert, angegeben!

Reaktionen auf Buttons

Swing-Komponenten **JButton**, **JTextField**, **JMenuItem**, **JComboBox**,… lösen **ActionEvents** aus.
Sie werden von **ActionListener**-Objekten beobachtet, mit einer einzigen Methode:

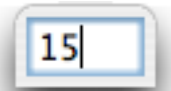
```
public void actionPerformed (ActionEvent e) {...}
```

Beispiel der Virus-Warnung (Abweichung vom Stil im Buch Java Gently!):

```
import javax.swing.*; import java.awt.*; import java.awt.event.*;
class LayoutComp extends JFrame
{  private JButton waitButton, rebootButton; int state = 0;
   LayoutComp (String title)
   {  …
      waitButton.addActionListener     // Listener für den waitButton
         (  new ActionListener ()    // anonyme Klasse direkt vom Interface
            {  public void actionPerformed(ActionEvent e)
               { state = 1; setBackground(Color.red); } });
      rebootButton.addActionListener // Listener für den rebootButton
         (  new ActionListener ()
            {  public void actionPerformed(ActionEvent e)
               { state = 2; setVisible(false); System.exit(0); } });
} }
```

Die Aufrufe von **setBackground** und **setVisible** beziehen sich auf das umgebende **LayoutComp**-Objekt — nicht auf das unmittelbar umgebende **ActionListener**-Objekt.

Eingabe von Texten



Komponente `JTextField`: einzeiliger, edierbarer Text

Ereignisse: **`ActionEvent`** (wie bei **`JButton`**) ausgelöst bei der Eingabe von <**`Return`**>

einige Methoden (aus der Oberklasse JTextComponent):

```
String getText ()                          Textinhalt liefern
void setText (String v)                    Textinhalt setzen
void setEditable (boolean e)               Edierbarkeit festlegen
boolean isEditable ()                      Edierbarkeit abfragen
void setCaretPosition (int pos)            Textcursor positionieren
```

Typischer **`ActionListener:`**

```
addActionListener
   (  new ActionListener ()
      {  public void actionPerformed (ActionEvent e)
         { String str = ((JTextField) e.getSource()).getText(); …
         } });
```

Eingabe von Zahlen: Text in eine Zahl konvertieren, Ausnahme abfangen:

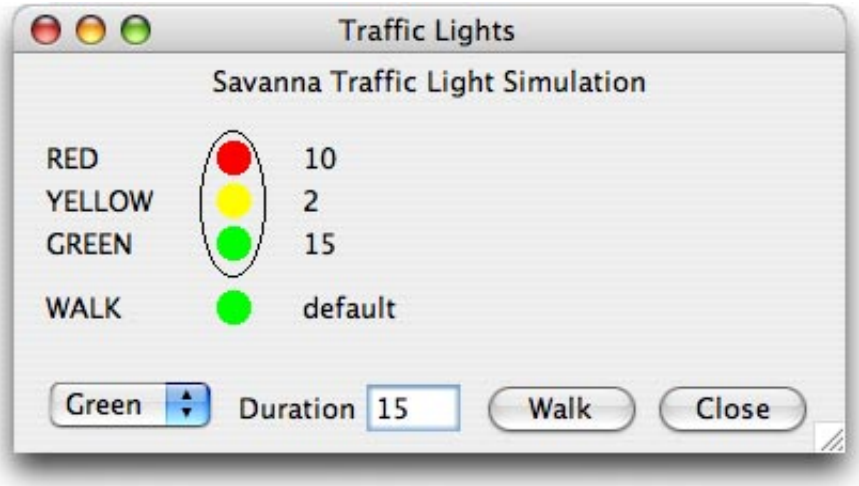
```
int age;
try { age = Integer.parseInt (str); }
catch (NumberFormatException e) { … } …
```

6. Beispiel: Ampel-Simulation

Aufgabe: Graphische Benutzungsoberfläche für eine Ampel-Simulation entwerfen

Eigenschaften:

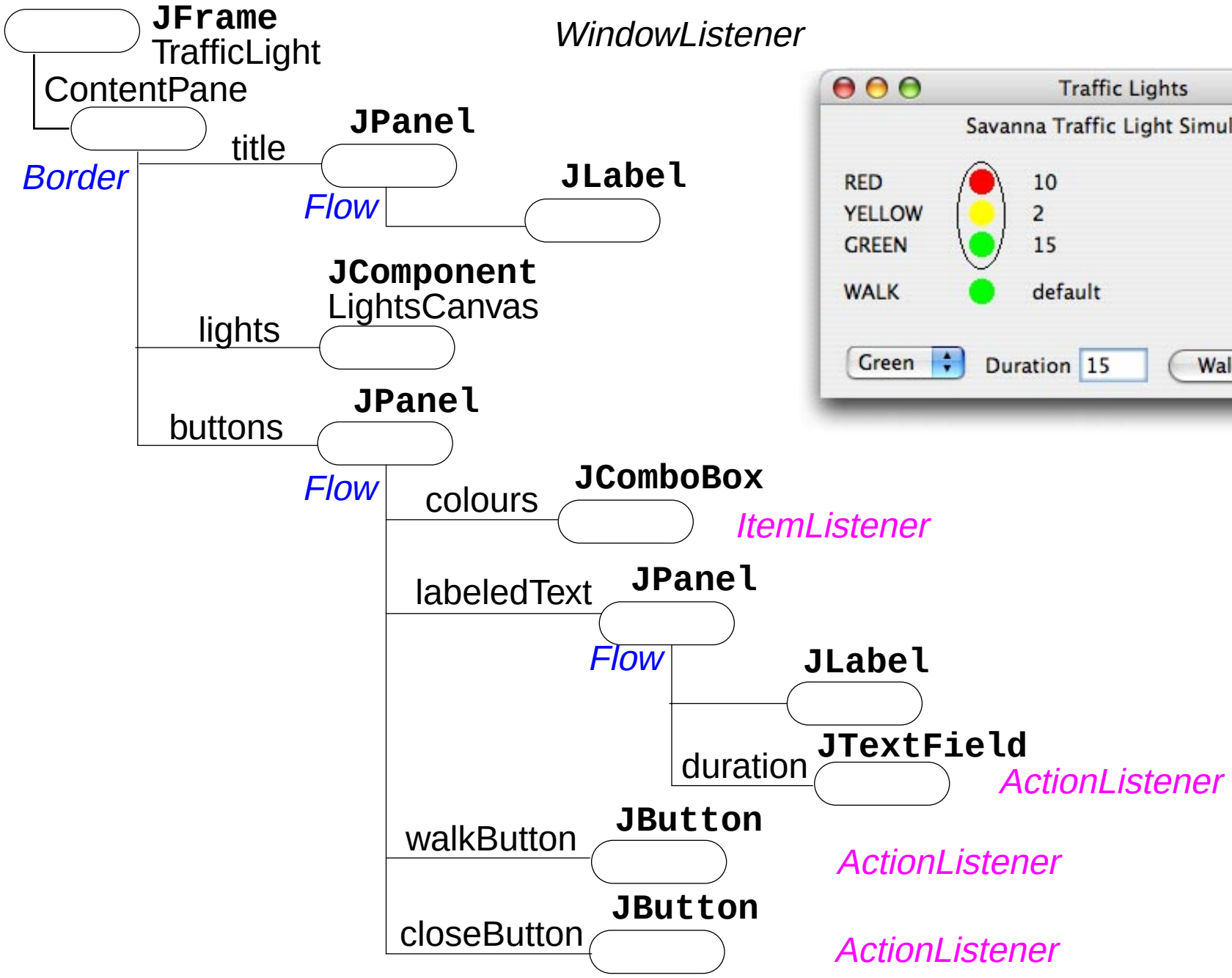
- Ampel visualisieren
  mit Knopf und Licht für Fußgänger
  (später auch animieren)
- Phasenlängen der Lichter
  einzeln einstellbar
- Einstellungen werden angezeigt



Entwicklungsschritte:

- Komponenten strukturieren
- zeichnen der Ampel (**`paint`** in eigener Unterklasse von **`JComponent`**)
- Komponenten generieren und anordnen
- Ereignisbehandlung entwerfen und implementieren

Objektbaum zur Ampel-Simulation



Programm zur Ampel-Simulation

Im Konstruktor der zentralen Klasse wird der **Objektbaum** hergestellt:

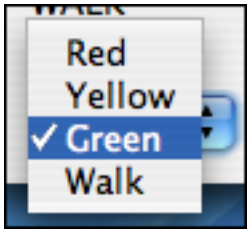
```
class TrafficLight extends JFrame
{                       //  basiert auf: The Traffic Light program by J M Bishop Oct 1997

                                        //  Objektvariable, auf die Listener zugreifen:
   private String[] message =           //  Phasendauer für jede Lampe als Text:
      { "default", "default", "default", "default" };
   private int light = 0;                           //  die ausgewählte Lampe
   private LightsCanvas lights;                     //  Enthält die gezeichnete Ampel

   public TrafficLight (String wt)                  //  Konstruktor der zentralen Klasse
   {  super (wt);                                   //  Aufbau des Objektbaumes:
      Container cont = getContentPane();            //  innere Fensterfläche
      cont.setLayout (new BorderLayout ());  // Layout des Wurzelobjektes

                                                    //  Zentrierter Titel:
      JPanel title = new JPanel (new FlowLayout (FlowLayout.CENTER));
      title.add (new JLabel("Savanna Traffic Light Simulation"));
      cont.add (title, BorderLayout.NORTH);

                        //  Die Ampel wird in einer getrennt definierten Klasse gezeichnet:
      lights = new LightsCanvas (message);
      cont.add (lights, BorderLayout.CENTER);
```

Auswahl-Komponente

Auswahl-Komponenten (**JComboBox**) lösen **ItemEvent**s aus, wenn ein Element ausgewählt wird. Mit der Methode **itemStateChanged** kann ein **ItemListener** darauf reagieren:

```
String[] lightNames = { "Red", "Yellow", "Green", "Walk" };
JComboBox colours = new JComboBox (lightNames);
```


```
colours.addItemListener
   (  new ItemListener ()
      {  public void itemStateChanged (ItemEvent e)
         {  if (e.getStateChange() == ItemEvent.SELECTED)
            {  String s = e.getItem().toString();

               if (s.equals("Red"))            light = 0;
               else if (s.equals("Yellow"))    light = 1;
               else if (s.equals("Green"))     light = 2;
               else if (s.equals("Walk"))      light = 3;
            }
         }
      }
   );
```

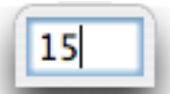


Über den **ItemEvent**-Parameter kann man auf das gewählte Element zugreifen.

Eingabe der Phasenlänge

Eingabe mit einem **JTextField**. Reaktion auf ein **ActionEvent**:

```
JPanel labeledText = new JPanel (new FlowLayout (FlowLayout.LEFT));
                                          //fasst TextField und Beschriftung zusammen
labeledText.add(new JLabel("Duration"));

                                  //  Eingabeelement für die Phasendauer einer Lampe:
JTextField duration = new TextField (3);
duration.setEditable (true);

duration.addActionListener
   (  new ActionListener ()
      {  public void actionPerformed (ActionEvent e)
         {                                    //  Zugriff auf den eingegebenen Text:
         // message[light] = ((JTextField) e.getSource()).getText();
                                                        //  oder einfacher:
            message[light] = e.getActionCommand ();
            lights.repaint();  // Die Zeichenmethode der gezeichneten Ampel
                               // wird erneut ausgeführt,
                               // damit der geänderte Text sichtbar wird.
         } });

labeledText.add (duration);
```



Button-Zeile

Einfügen der **Button-Zeile** in den Objektbaum:

```
   JButton walkButton = new JButton ("Walk");
                                                  // noch keine Reaktion zugeordnet

   JButton closeButton = new JButton ("Close");
   closeButton.addActionListener
      (  new ActionListener ()                          // Programm beenden:
         {  public void actionPerformed(ActionEvent e)
            { setVisible (false); System.exit (0); }
         }
      );
                                                  // Zusammensetzen der Button-Zeile:
   JPanel buttons = new JPanel(new FlowLayout (FlowLayout.CENTER));
   buttons.add (colours);
   buttons.add (labeledText);
   buttons.add (walkButton);
   buttons.add (closeButton);

   cont.add (buttons, BorderLayout.SOUTH);
} // TrafficLight Konstruktor

   public static void main (String[] args) { JFrame f = … }
} // TrafficLight Klasse
```

Ampel zeichnen und beschriften

Eine Unterklasse der allgemeinen Oberklasse **`JComponent`** für Swing-Komponenten stellt die **Zeichenfläche** bereit. Die Methode **`paint`** wird zum Zeichnen und Beschriften überschrieben:

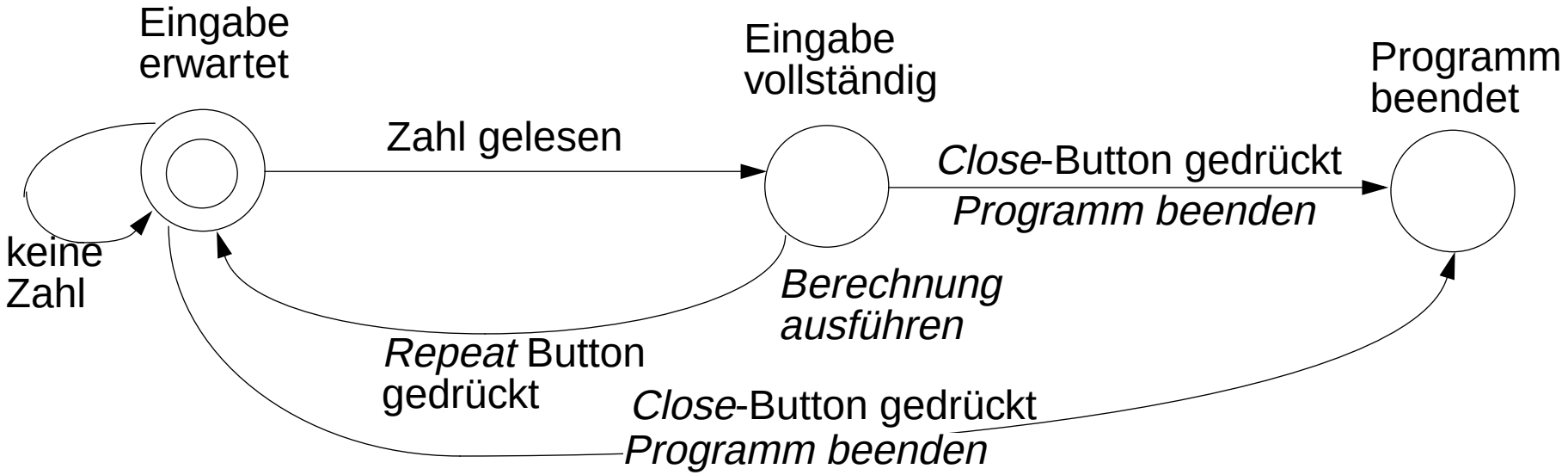
```
class LightsCanvas extends JComponent
{  private String[] msg;
   LightsCanvas (String[] m)                    // Die Array-Elemente enthalten die
   { msg = m; }                                 // Phasendauern der Lampen als Text. Sie
                                                // können durch Eingaben verändert werden.
   public void paint (Graphics g)
   {  super.paint(g);                           // Hintergrund der Komponente zeichnen
      g.drawOval (87, 10, 30, 68);              // darauf: Ampel zeichnen und beschriften
      g.setColor (Color.red);    g.fillOval (95, 15, 15, 15);
      g.setColor (Color.yellow); g.fillOval (95, 35, 15, 15);
      g.setColor (Color.green);  g.fillOval (95, 55, 15, 15);
      g.fillOval (95, 85, 15, 15);                 // walk Lampe ist auch grün

      g.setColor(Color.black);
      g.drawString ("RED", 15 ,28); g.drawString ("YELLOW", 15, 48);
      g.drawString ("GREEN", 15, 68); g.drawString ("WALK", 15, 98);
                                                // eingegebene Phasendauern der Lampen:
      g.drawString (msg[0], 135 ,28); g.drawString (msg[1], 135, 48);
      g.drawString (msg[2], 135, 68); g.drawString (msg[3], 135, 98);
}  }
```

7. Entwurf von Ereignisfolgen

Die zulässigen **Folgen von Bedienereignissen und Reaktionen** darauf müssen für komplexere Benutzungsoberflächen geplant und entworfen werden.

Modellierung durch **endliche Automaten** (auch durch *StateCharts*)

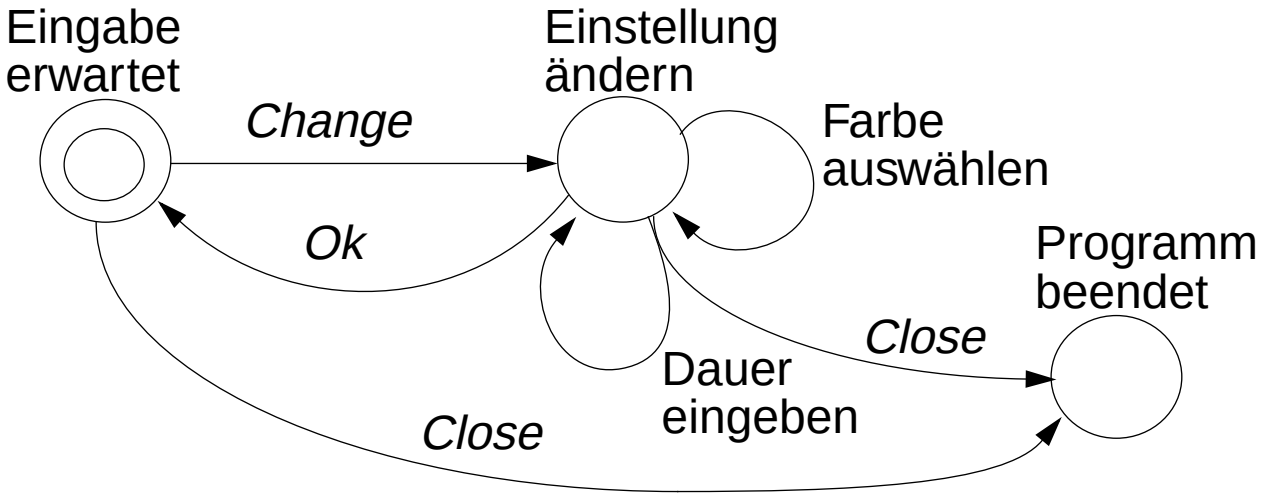
- **Zustände** unterscheiden Bediensituationen (z. B. „Eingabe erwartet“, „Eingabe vollständig“)
- **Übergänge** werden durch Ereignisse ausgelöst.
- **Aktionen** können mit Übergängen verknüpft werden; Reaktion auf ein Ereignis
  z. B. bei Eingabe einer Phasenlänge Ampel neu zeichnen, und
- **Aktionen** können mit dem Erreichen eines Zustandes verknüpft werden,
  z. B. wenn die Eingabe vollständig ist, Berechnung beginnen.



Unzulässige Übergänge

In manchen Zuständen sind **einige Ereignisse nicht als Übergang definiert**.
Sie sind in dem Zustand **unzulässig**, z. B. „Farbe auswählen“ im Zustand „Eingabe erwartet“.

Beispiel: Ampel-Simulation erweitert um zwei Buttons **Change** und **Ok**:

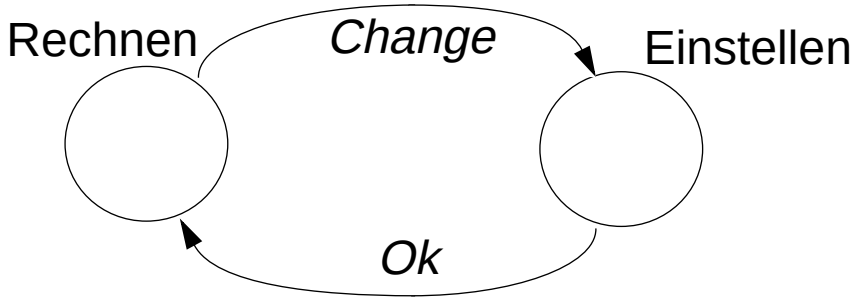


Robuste Programme dürfen auch an unzulässigen Ereignisfolgen nicht scheitern.
Verschiedene Möglichkeiten für **nicht definierte Übergänge**:

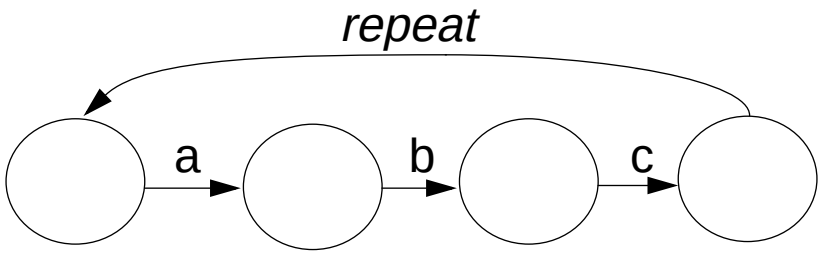
- Sie bleiben **ohne Wirkung**
- Sie bleiben ohne Wirkung und es wird eine **Erklärung** gegeben (Warnungsfenster).
- **Komponenten** werden so **deaktiviert**, dass unzulässige Ereignisse nicht ausgelöst werden können.

Muster für Ereignisfolgen

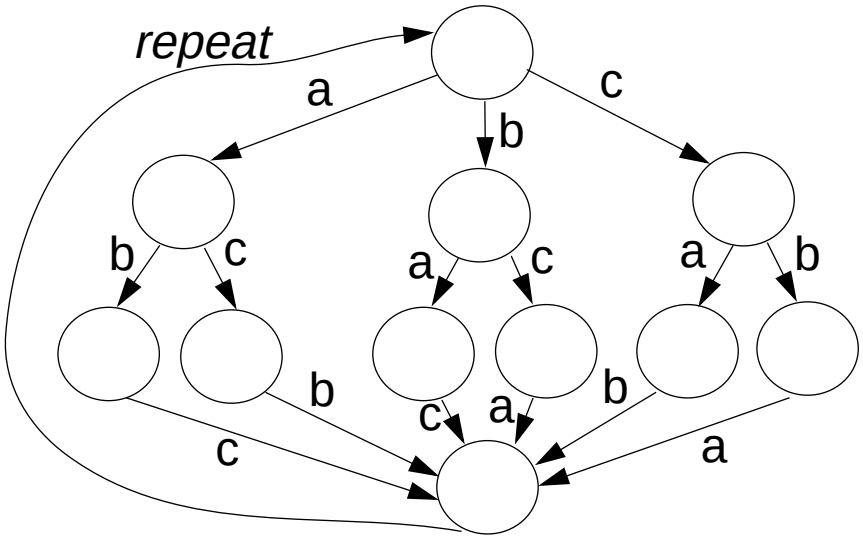
- Verschiedene **Bedienungsarten** (mode): Vorsicht: Nicht unnötig viele Zustände entwerfen. “Don’t mode me in!”



- Festgelegte **sequentielle Reihenfolge**: Vorsicht: Nicht unnötig streng vorschreiben. Bediener nicht gängeln.



- **Beliebige Reihenfolge** von Ereignissen: Modellierung mit endlichem Automaten ist umständlich (Kombinationen der Ereignisse); einfacher mit *StateCharts*.



- **Voreinstellungen** (*defaults*) können Zustände sparen und Reihenfolgen flexibler machen. Vorsicht: Nur sinnvolle Voreinstellungen.

Implementierung des Ereignis-Automaten

Zustände ganzzahlig **codieren**; **Objektvariable** speichert den **augenblicklichen Zustand**:

```
private int currentState = initialState;
private static final int initialState = 0, settingState = 1, …;
```

Einfache **Aktionen der Übergänge** bleiben in den Reaktionsmethoden der **`Listener`**; Methodenaufruf für den Übergang in einen neuen Zustand zufügen:

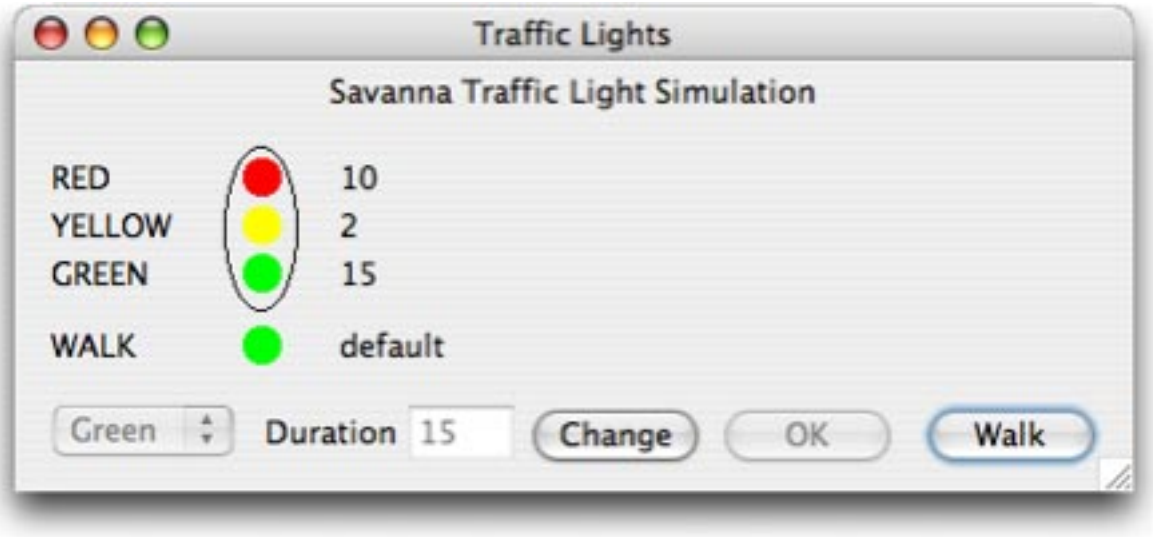
```
okButton.addActionListener
   (  new ActionListener ()
      {  public void actionPerformed(ActionEvent e)
         {  message[light] = duration.getText();
            toState (initialState); } });
```

Aktionen der Zustände in **Übergangsmethode** platzieren, z. B. Komponenten (de)aktivieren:

```
private void toState (int nextState)
{  currentState = nextState;
   switch (nextState)
   {  case initialState:
         lights.repaint();
         okButton.setEnabled(false); changeButton.setEnabled (true);
         colours.setEnabled (false); duration.setEnabled (false);
         break;
      case settingState:
         okButton.setEnabled(true); changeButton.setEnabled (false); …
         break; } }
```

Ampel-Simulation mit kontrollierten Zustandsübergängen

Zwei Knöpfe wurden zugefügt:
Der **Change-Button** aktiviert die Eingabe, der **Ok-Button** schliesst sie ab.



Die Komponenten zur Farbauswahl, Texteingabe und der Ok-Knopf sind im gezeigten Zustand deaktiviert.