Software-Entwicklung II
SS 2004

Prof. Dr. Uwe Kastens

Ziele der Vorlesung

Ziele der Vorlesung Software-Entwicklung 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 Software-Entwicklung 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

Literaturhinweise

Elektronisches Skript zur Vorlesung:

- **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 und JDK 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:

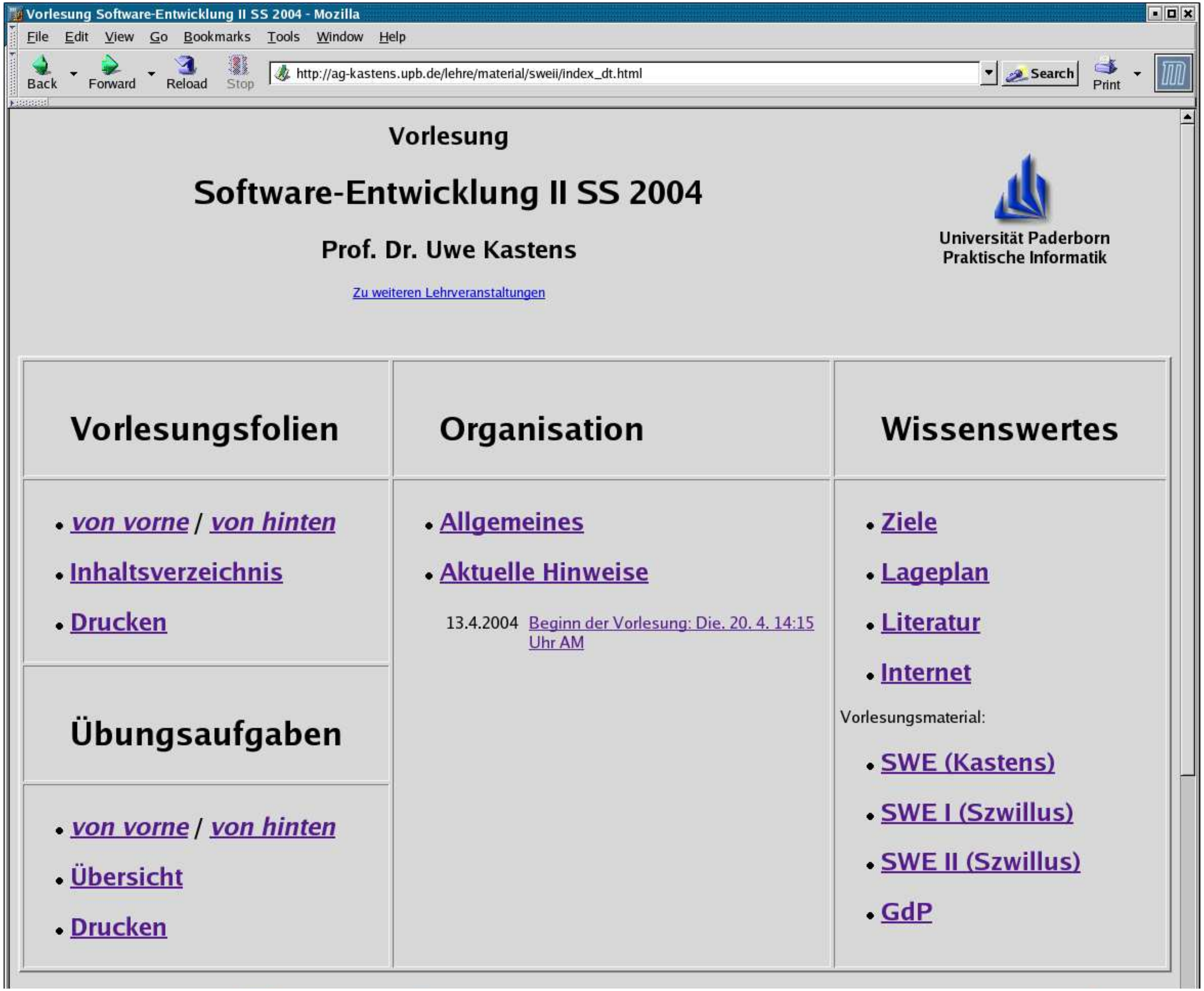
- **U. Kastens: Vorlesung Grundlagen der Programmiersprachen, Skript, 2003 http://www.upb.de/cs/ag-kastens/gdp**
- **D. A. Watt: Programmiersprachen - Konzepte und Paradigmen, Hanser, 1996**

eine Einführung in Java von den Autoren der Sprache:

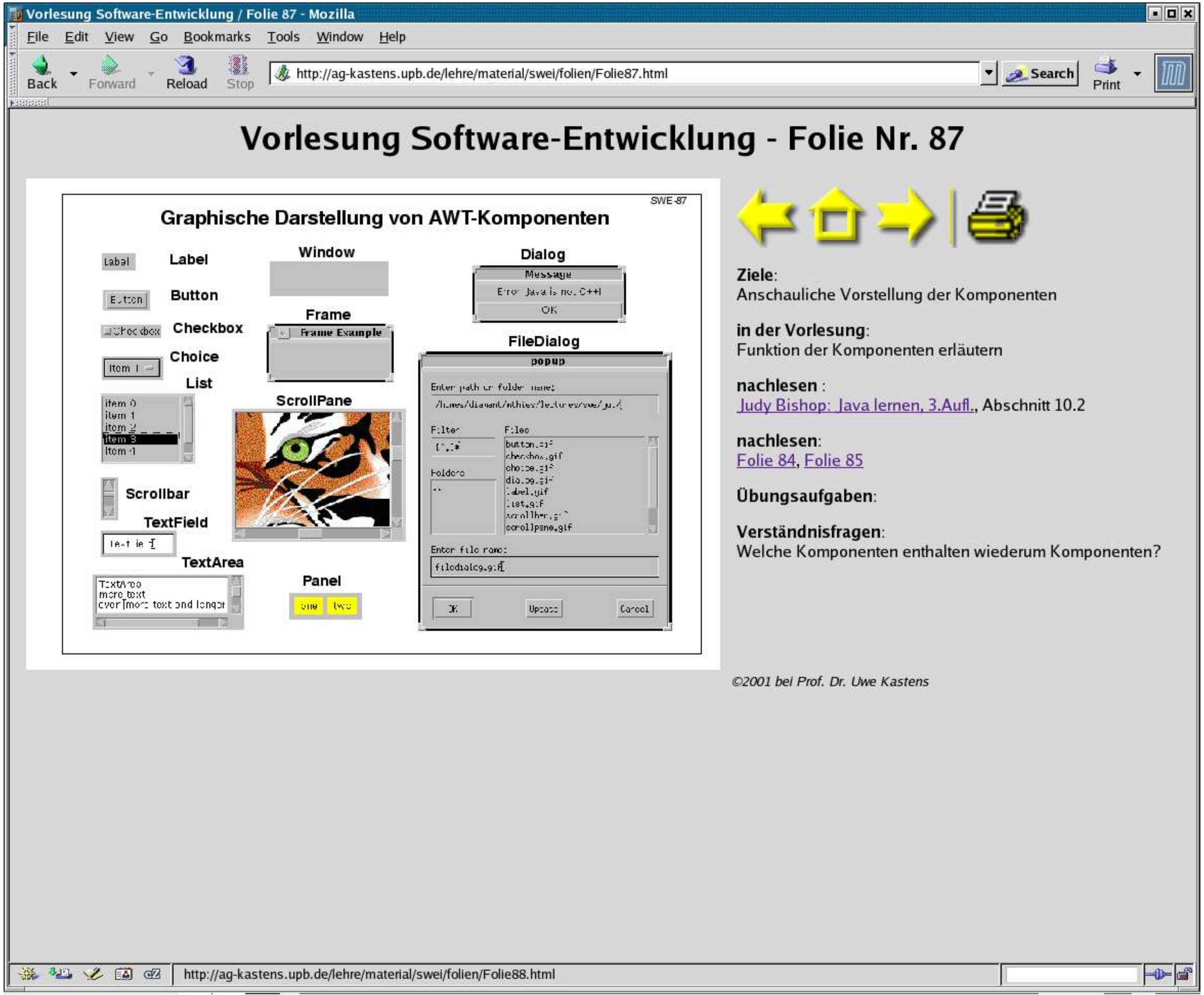
- **Arnold, Ken / Gosling, James: The Java programming language, Addison-Wesley, 1996.**
- **Arnold, Ken / Gosling, James: Die Programmiersprache Java TM, 2. Aufl. Addison-Wesley, 1996**

Hinweise auf weiteres Material im SWE-Skript im WWW

Elektronisches Skript: HomePage



Elektronisches Skript: Folien im Skript



Elektronisches Schript: Organisation der Vorlesung

Vorlesung Software-Entwicklung II - Organisation - Mozilla

File Edit View Go Bookmarks Tools Window Help

Organisation

Sprechstunde Prof. Dr. Uwe Kastens:

- Mo 11.00 - 12:00 F2.308
- Do 16.00 - 17.00 F2.308

Termine

Vorlesung:

- Di 14:15 - 15:45 AM
- Mi 14:15 - 15:45 AM

Beginn: Dienstag, 20.4.2004

Zentralübung:

- Mi 13:15 - 14:00 AM

Beginn: Mittwoch, 5.5.2004

Übungsbetreuer:

Tomasz Mistrzyk

Übungen:

- Gruppe 1 Mo 11 - 13 E4.101 N.N.
- Gruppe 2 Mo 14 - 16 E4.101 N.N.
- Gruppe 3 Mo 11 - 13 E3.143 N.N.
- Gruppe 4 Mo 14 - 16 E3.143 N.N.
- Gruppe 5 Mo 16 - 18 E3.143 N.N.
- Gruppe 6 Mo 11 - 13 D3.344 N.N.
- Gruppe 7 Mo 14 - 16 D3.344 N.N.

Vorlesung Software-Entwicklung II - Organisation - Mozilla

File Edit View Go Bookmarks Tools Window Help

- Gruppe 21 Fr 14 - 16 E3.301 N.N.

Beginn: Montag, 26.4.2004

Sichern Sie sich Ihren Platz bis Dienstag, 25.04.2004: Benutzen Sie Ihren Eintrag aus der Teilnehmer-Datenbank von SWE I (WS 2003/04) (siehe unten). Dort können Sie den Wunschtermin angeben, zu dem Sie eine Übungsgruppe besuchen möchten. Wenn in der Gruppe noch Plätze frei sind, wird Ihr Wunsch erfüllt.

Teilnehmer-Datenbank:

- In Ihrem Teilnehmerkonto können Sie jederzeit die aktuell über Sie gespeicherten Informationen abrufen.

  Hier geht es zu Ihrem Teilnehmerkonto: Mein Konto NEU

  Sie können Ihre persönlichen Informationen, sowie die Zugehörigkeit zu einer Übungsgruppe und Ihre Punkte aus den Hausaufgaben einsehen. Später finden Sie hier die Klausuranmeldung und erfahren Ihr Klausurergebnis.

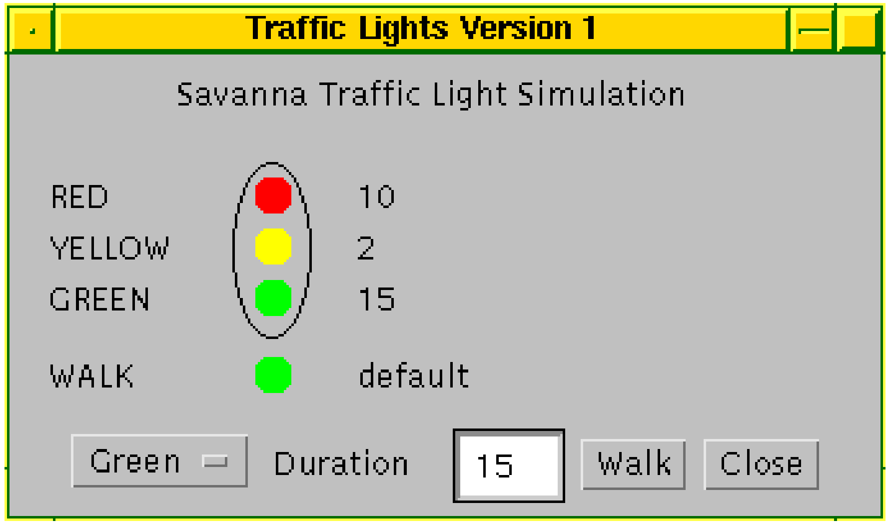
Übungsaufgaben:

- Die Übungsblätter werden einmal pro Woche (meist freitags) im WWW veröffentlicht und können dann bis zur darauffolgenden Woche bearbeitet werden. Abgabetermin: siehe Übungsblatt (meist freitags).
- Insgesamt 6 der Übungsaufgaben werden als **Korrekturaufgabe** gekennzeichnet. Abgaben dieser Aufgaben werden auf einer Punkteskala von 0-2 bewertet (nicht bearbeitet, sinnvoll bearbeitet, gut bearbeitet). Insgesamt können durch die bewerteten Aufgaben also 12 Punkte erreicht werden.
- Die Übungspunkte lassen sich wie folgt verwenden, um die Noten von **bestandenen** Klausuren zu verbessern: Wer mindestens 6 Punkte erreicht, verbessert sein Klausurergebnis um einen Zwischenwert, z. B. von 2.3 auf 2.0.
- **Abgabe der Korrekturaufgaben**
  - Geben Sie **in Gruppen von bis zu drei Teilnehmern** ab.
  - Bitte notieren Sie im Kopf Ihrer Abgabe die Nummer des Übungsblattes und die Namen und Matrikelnummern aller Beteiligten. Die Nummer der Übungsgruppe muss deutlich auf die Abgabe geschrieben werden, damit wir die korrigierten Zettel in die richtige Gruppe mitnehmen können (Beispiel).
  - Geben Sie Ihre Lösungen im entsprechend gekennzeichneten orangenen Kasten auf dem D3-Flur ab.
  - Sie erhalten die korrigierten Lösungen in den Übungen zurück. Nicht abgeholte Lösungen finden Sie auf dem orangenen Schrank im D3-Flur.
  - Bei abgeschriebenen Lösungen werden wir sowohl Abschrift als auch das Original mit 0 Punkten bewerten. Bei wiederholtem Abschreiben werden Sie vom Bonusnotenverfahren ausgeschlossen.

1. Einführung in graphische Benutzungsoberflächen

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

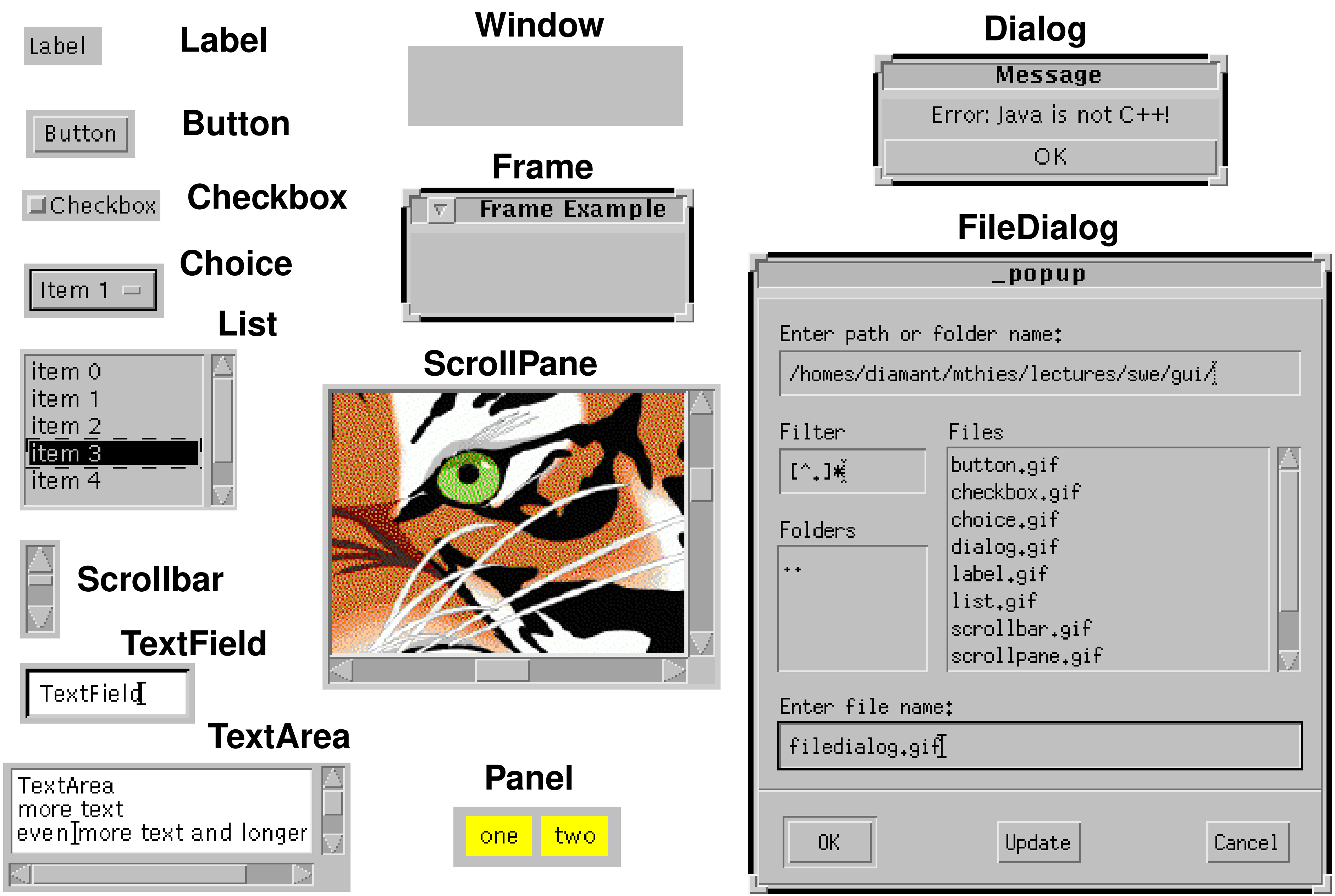
- interaktiven Bedienung von Programmen,
- Ein- und Ausgabe mit graphischen Techniken und visuellen Komponenten



Javas Standardbibliothek `jawa.awt`
(abstract windowing toolkit) enthält wiederverwendbare Klassen zur Implementierung und Benutzung der wichtigsten GUI-Komponenten:

- Graphik
- GUI-Komponenten (siehe SWE-84)
- Plazierung, Layoutmanager
- Ereignisbehandlung (**`java.awt.event`**)
- Bildmanipulation

Graphische Darstellung von AWT-Komponenten



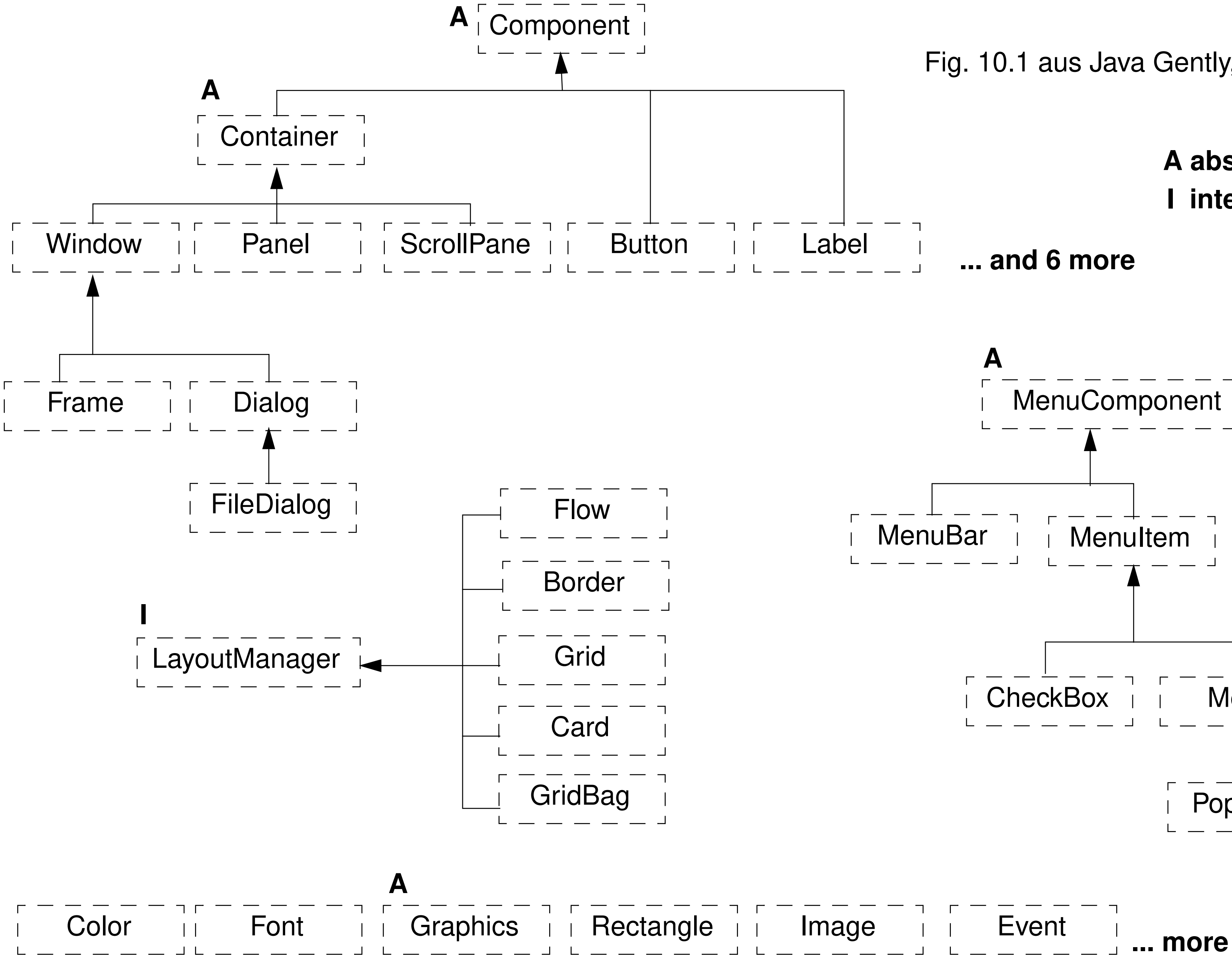
Klassenhierarchie für Komponenten von Benutzungsoberflächen

Teil der Standardbibliothek **`java.awt`** (abstract windowing toolkit)

Klasse in der Hierarchie	Kurzbeschreibung
Component (abstract)	darstellbare Komponenten von Benutzungsoberflächen
Container (abstract)	Behälter für Komponenten
Panel	konkrete Klasse zu Container, Behälter für Komponenten
ScrollPane	Sicht auf große Komponente, 2 Rollbalken
Window	Fenster (ohne Rand, Titel, usw.); Wurzel der Objektbäume
Dialog	Fenster; fordert interaktive Bearbeitung
FileDialog	Fenster zur interaktiven Dateiauswahl
Frame	Fenster mit Rand, Titel, Menüleiste; veränderbare Größe
Button	Schaltknopf
Canvas	frei verwendbare rechteckige Zeichenfläche
CheckBox	An/Aus-Schalter
Choice	Auswahl aus einer Liste von Texten (ausklappbar)
List	Auswahl aus einer Liste von Texten (mit Rollbalken)
Scrollbar	Rollbalken zur Einstellung eines ganzzahligen Wertes
Label	Textzeile
TextComponent	edierbarer Text
TextField	einzelne Textzeile
TextArea	mehrzeiliger Text mit Rollbalken

Klassenhierarchie der AWT-Bibliothek

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



A abstract class
I interface

Wiederverwendung von Klassen aus Bibliotheken

Die Klasse **`java.awt.Frame`** implementiert **gerahmte Fenster** in graphischen Benutzungsoberflächen (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

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 `Frame`. Sie erben die Funktionalität der Oberklassen.
Die Methode **`paint`** **wird überschrieben** mit einer Methode, die das Gewünschte zeichnet:

```
public class Rings extends Frame
{  public Rings () { setTitle ("Olympic Rings"); }
   public void paint (Graphics g) { // draw olympic rings ... }
   public static void main (String [] args)
   {  Frame f = new Rings(); ... }
}
```

Einige Eigenschaften auf den Ebenen der `Frame`-Hierarchie

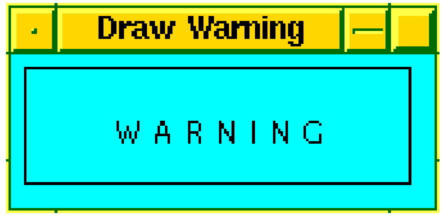
Klasse	Datenelemente	Ereignisse	Methoden
Component	Location, Size, Bounds, Visible	Key, Mouse, Focus, Component	paint
Container	Layout	Container	add, getComponents, paint
Window	Locale	Window	show, pack, toBack, toFront
Frame	Title, MenuBar, Resizable, IconImage		

Namenskonventionen:

zum Datenelement XXX
gibt es die Methoden
getXXX und ggf. setXXX

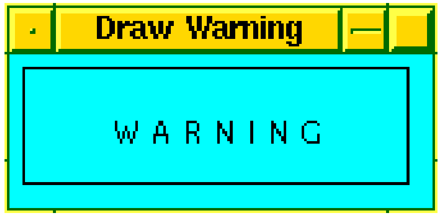
Ereignisse sind Objekte
der Klassen YYYEvent

2. Zeichenflächen benutzen, Programmschema



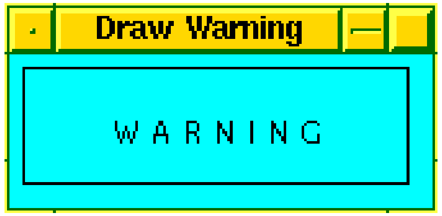

```
import java.awt.*; import java.awt.event.*;
        // Hauptklasse als Unterklasse von Frame:
public class GraphicWarning extends Frame
{  GraphicWarning (String title)                         // Konstruktor
   {  super (title);                   // Aufruf des Konstruktors von Frame








   }

   public void paint (Graphics g)      // überschreibt paint in einer Oberklasse
   {

   }

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

Programmschema: Eigenschaften und Ereignisbehandlung



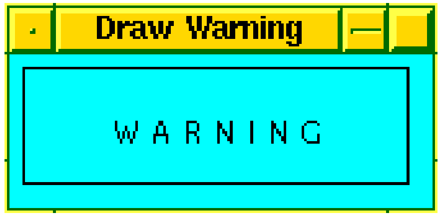

```
import java.awt.*; import java.awt.event.*;
          // Hauptklasse als Unterklasse von Frame:
public class GraphicWarning extends Frame
{  GraphicWarning (String title)                        // Konstruktor
   {  super (title);                    // Aufruf des Konstruktors von Frame
      setBackground (Color.cyan);
      setSize (35*letter,6*line);                        // Größe festlegen
      addWindowListener            // Ereignisbehandlung: Beim Drücken des
         (new WindowAdapter ()         // Schließknopfes Programm beenden.
            { public void windowClosing(WindowEvent e)
               { System.exit(0); } });
      setVisible (true);                  // führt zu erstem Aufruf von paint
   }
   static private final int line = 15, letter = 5;    // zur Positionierung
   public void paint (Graphics g)     // überschreibt paint in einer Oberklasse
   {
   }

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

Programmschema: `paint`-Methode ausfüllen




```
import java.awt.*; import java.awt.event.*;
          // Hauptklasse als Unterklasse von Frame:
public class GraphicWarning extends Frame
{  GraphicWarning (String title)                              // Konstruktor
   {  super (title);                            // Aufruf des Konstruktors von Frame
      setBackground (Color.cyan);
      setSize (35*letter,6*line);                             // Größe festlegen
      addWindowListener                     // Ereignisbehandlung: Beim Drücken des
         (new WindowAdapter ()                // Schließknopfes Programm beenden.
            { public void windowClosing(WindowEvent e)
               { System.exit(0); } });
      setVisible (true);                           // führt zu erstem Aufruf von paint
   }
   static private final int line = 15, letter = 5;     // zur Positionierung
   public void paint (Graphics g)       // überschreibt paint in einer Oberklasse
   {  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)
   {  Frame f = new GraphicWarning ("Draw Warning");   // Objekt erzeugen
   }
}
```

Ablauf des Zeichen-Programms



1. **`main`** aufrufen:
    1.1. **`GraphicWarning`**-Objekt erzeugen, Konstruktor aufrufen:
        1.1.1 **`Frame`**-Konstruktor aufrufen
        1.1.2 Eigenschaften setzen, z. B. Farben,
        1.1.3 ggf. weitere Initialisierungen des Fensters
        1.1.4 Größe festlegen, **`setSize (...)`**,
        1.1.5 Reaktion auf Drücken des Schließknopfes vorbereiten **`addWindowListener`**,
        1.1.6 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:

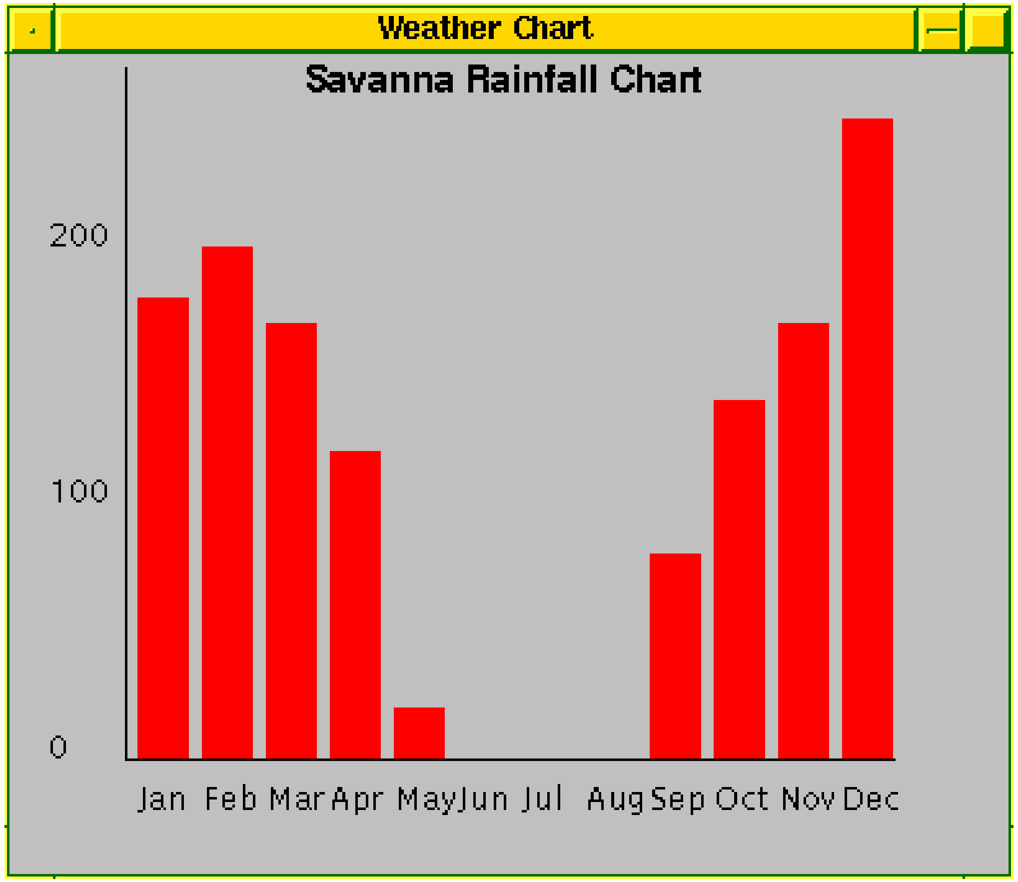
2. **`Graphics`** Objekt erzeugen
    2.1 damit erstmals **`paint`** aufrufen (immer wieder, wenn nötig):
    weiter in **`paint`** von **`GraphicWarning`**
        2.1.1 auf der Zeichenfläche des Parameters **`g`** schreiben und zeichnen

3. Schließknopf Drücken

in Methoden der Oberklassen:

    3.1 **`windowClosing`** des **`WindowListener`** aufrufen
        3.1.1 Programm beenden, **`exit(0)`**

Beispiel: Balkendiagramm zeichnen



Beispiel: Balkendiagramm zeichnen

```
public void paint (Graphics g)
{  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);
   }
}
static double[] rainTable = new double[12];
static String Months [] =
   {"Jan","Feb","Mar","Apr","May","Jun","Jul","Aug","Sep","Oct","Nov","Dec"};
```

3. AWT-Komponenten erzeugen und platzieren

Ein einfaches Beispiel für Text und Schaltknöpfe:

Aufgaben zur Herstellung:

Aussehen:

- Label- und Button-Objekte generieren
- Anordnung der Komponenten festlegen

Ereignisse:

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

Komponenten platzieren

Die Klasse **`Container`** (Oberklasse von **`Frame`**) 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 **`Frame`**:

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

Er 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.

```
add (new Label ("W A R N I N G")); ...

Button waitButton = new Button ("Wait"); add (waitButton); ...
```

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

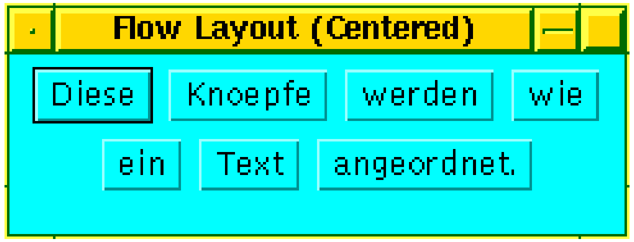
Wird die Gestalt des **`Container`**s verändert, so ordnet der **`LayoutManager`** die Komponenten ggf. neu an.

Programmschema zum Platzieren von Komponenten

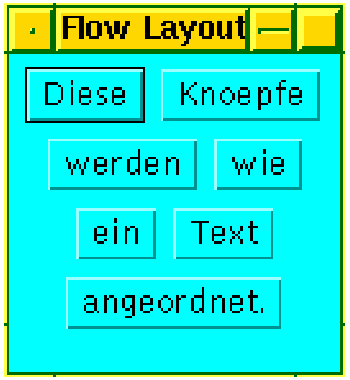
```
import java.awt.*; import java.awt.event.*;

class FlowTest extends Frame                    // Definition der Fenster-Klasse
{  FlowTest ()                                  // Konstruktor
   {  setBackground(Color.cyan);                // Eigenschaften des Fensters setzen

      setLayout(new FlowLayout(FlowLayout.CENTER));     // LayoutManager

      add(new Button("Diese")); ...                     // Komponenten zufügen

      setTitle("Flow Layout (Centered)"); setSize(250,100);
      setVisible(true);
   }
}

public class LayoutTry
{  public static void main(String[] args)
   { Frame f = new FlowTest();                  // Ein FlowTest-Objekt erzeugen
      WindowListener winListen =                // Ereignisbehandlung
         new WindowAdapter ()
         {  public void windowClosing(WindowEvent e)
            {  System.exit(0); }
         };
      f.addWindowListener (winListen);
   }
}
```

LayoutManager FlowLayout

```
class FlowTest extends Frame
{  FlowTest ()
   {  setBackground(Color.cyan); setForeground(Color.black);

      setLayout(new FlowLayout(FlowLayout.CENTER));

      add(new Button("Diese")); add(new Button("Knoepfe"));
      add(new Button("werden")); add(new Button("wie"));
      add(new Button("ein")); add(new Button("Text"));
      add(new Button("angeordnet."));

      setTitle("Flow Layout (Centered)"); setSize(250,100);
      setVisible(true);
   }
}
```

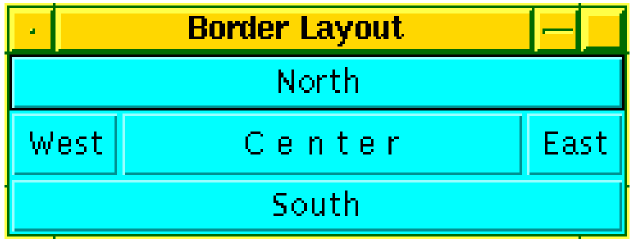


nach
Ändern
der Form:

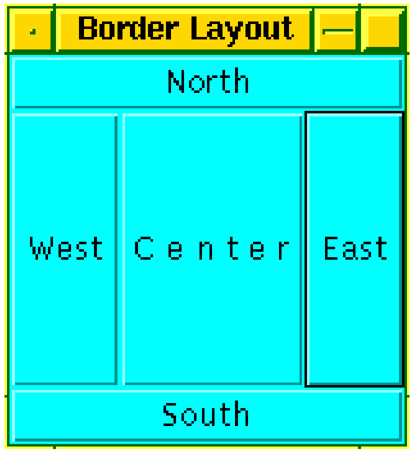


LayoutManager BorderLayout

```
class BorderTest extends Frame
{  BorderTest ()
   {  setBackground(Color.cyan); setForeground(Color.black);

      setLayout(new BorderLayout());

      add(new Button("North"), BorderLayout.NORTH);
      add(new Button("East"), BorderLayout.EAST);
      add(new Button("South"), BorderLayout.SOUTH);
      add(new Button("West"), BorderLayout.WEST);
      add(new Button("C e n t e r"), BorderLayout.CENTER);

      setTitle("Border Layout"); setSize(250,100); setVisible(true);
   }
}
```

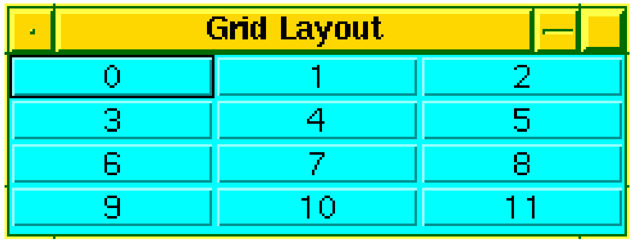


nach
Ändern
der Form:

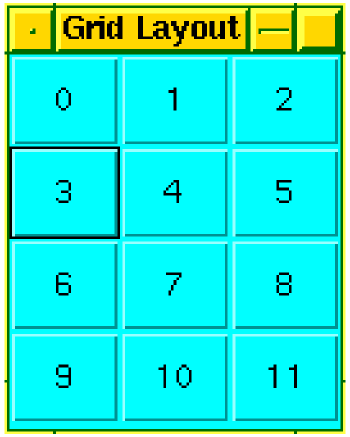


LayoutManager GridLayout

```
class GridTest extends Frame
{  GridTest ()
   {  setBackground(Color.cyan); setForeground(Color.black);

      setLayout(new GridLayout(4, 3));

      for (int i = 0; i < 12; i++)
         add(new Button((new Integer(i)).toString()));

      setTitle("Grid Layout"); setSize(250,100); setVisible(true);
   }
}
```

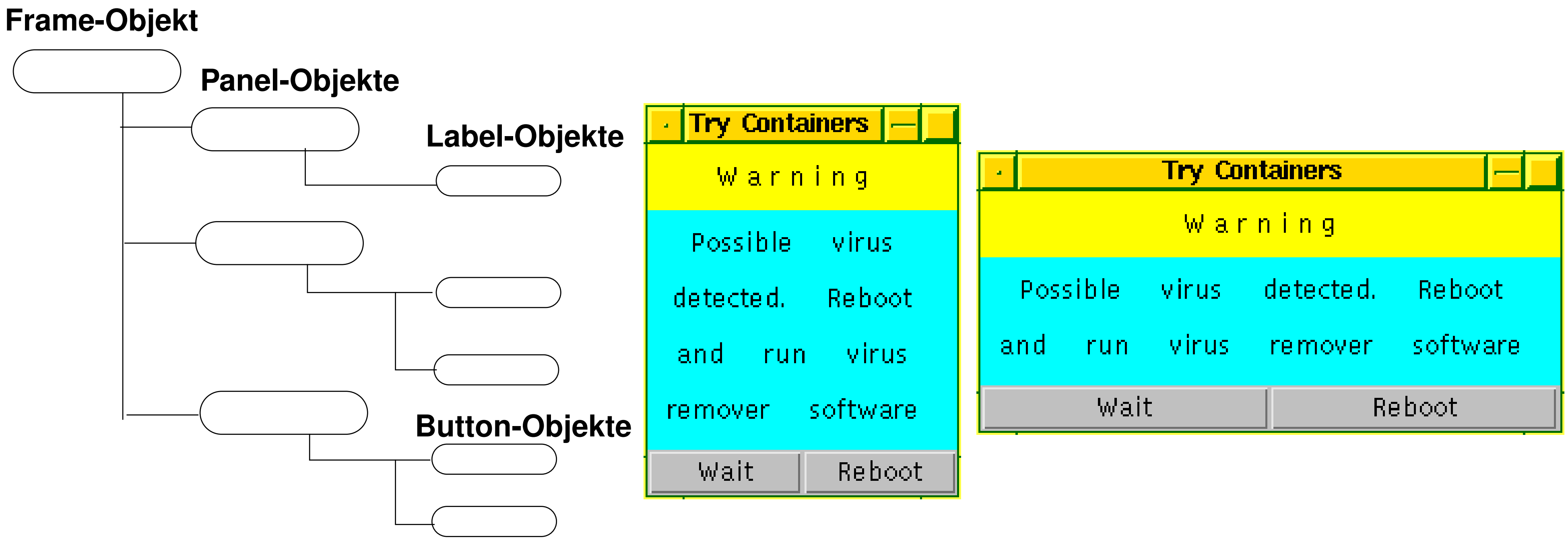


nach Ändern der Form:



4. Hierarchisch strukturierte Fensterinhalte

- **Zusammengehörige Komponenten** in einem Objekt einer `Container`-Unterklasse unterbringen (`Panel`, `Window`, `Frame` oder selbstdefinierte Unterklasse).
- Eigenschaften der Objekte im `Container` können dann gemeinsam bestimmt werden, z. B. Farbe, `LayoutManager`, usw. zuordnen.
- Mit `Container`-Objekten werden beliebig tiefe **Baumstrukturen** von AWT-Komponenten erzeugt. In der visuellen Darstellung sind sie **ineinander geschachtelt**.



Programm zu hierarchisch strukturiertem Fenster

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

class LabelContainer extends Container          // Klasse zur Herstellung von
{  LabelContainer (String[] words)              // Fließtext aus String-Array
   {  setLayout(new FlowLayout(FlowLayout.CENTER));
      for (int i = 0; i < words.length; i++)
         add (new Label (words[i]));
   }
}

class LayoutComp                         // Hauptklasse erzeugt den Komponentenbaum
{  public static void main(String[] args)
   {  String [] message = {"Possible", "virus", "detected.", "Reboot",
                           "and", "run", "virus", "remover", "software"};
      Container warningText = new LabelContainer (message);

      ... Erzeugung des Komponentenbaumes einfügen ...

      WindowListener winListen = new WindowAdapter ()
      { public void windowClosing(WindowEvent e) { System.exit(0); }};
      rootFrame.addWindowListener (winListen);
      rootFrame.setLocation (100, 100); rootFrame.setSize (160, 200);
      rootFrame.setVisible(true);
   }
}
```

Komponentenbaum erzeugen

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

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

                                   // Knöpfe im Panel mit GridLayout:
Panel twoButtons = new Panel ();
twoButtons.setBackground(Color.lightGray);
twoButtons.setLayout (new GridLayout(1, 2));
twoButtons.add (new Button ("Wait"));
twoButtons.add (new Button ("Reboot"));

                             // im Frame mit BorderLayout zusammenfassen:
Frame rootFrame = new Frame ("Try Containers");
rootFrame.setBackground(Color.cyan);
rootFrame.setLayout(new BorderLayout());
rootFrame.add(header, BorderLayout.NORTH);
rootFrame.add(warningText, BorderLayout.CENTER);
rootFrame.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, Alternative auswählen, Mauszeiger auf eine graphisches Element schieben.
- **Programmausführung reagiert auf solche Ereignisse** durch Aufruf bestimmter Methoden

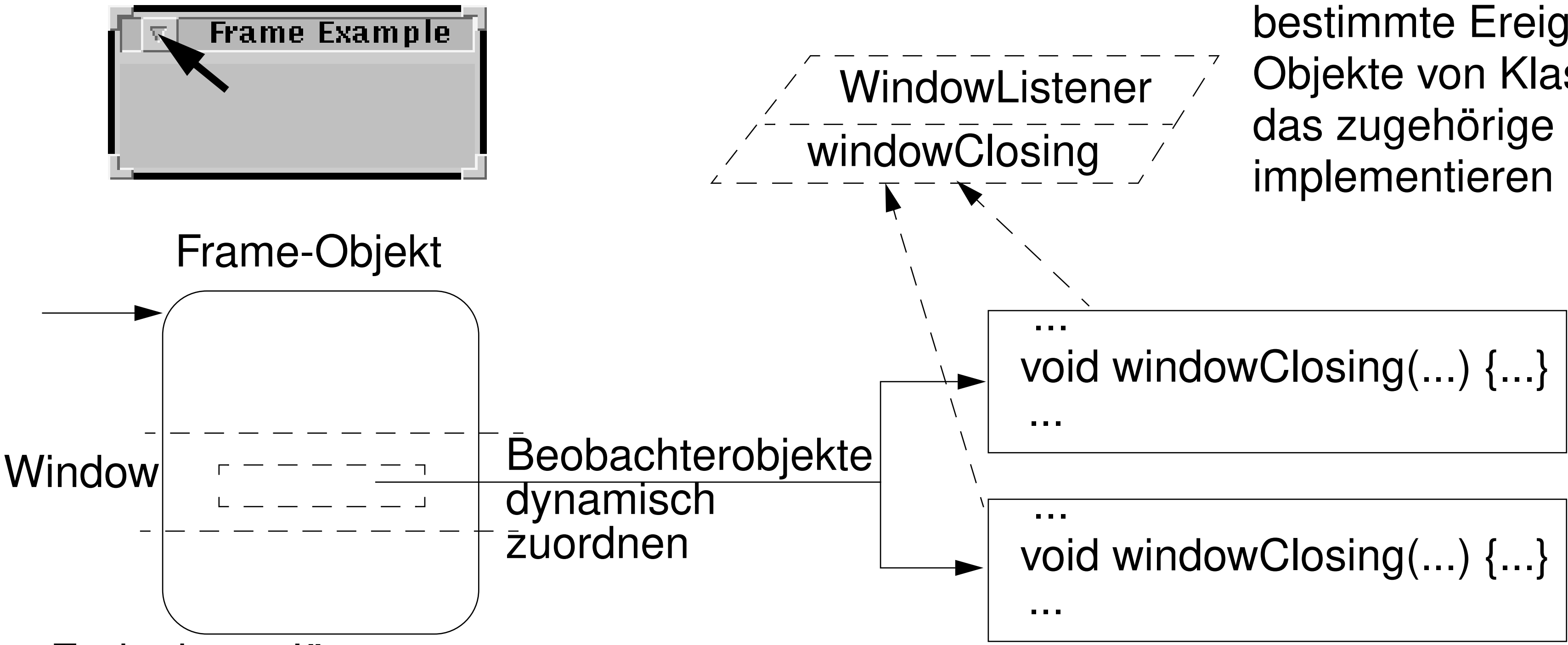
Aufgaben:

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

„Observer"-Prinzip in der Ereignisbehandlung

An AWT-Komponenten werden Ereignisse ausgelöst, z. B. ein WindowEvent an einem Frame-Objekt:

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.

Programmiertechnik für `Listener`

Im **`java.awt`** 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 WClose extends WindowAdapter
{  public void windowClosing (WindowEvent e)
   { System.exit (0); }
}
```

Zufügen eines Listener-Objektes zu einer AWT-Komponente:

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

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

Zufügen eines `Listener`-Objektes zu einer AWT-Komponente:

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

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

AWT-Komponenten **`Button, List, MenuItem, TextField`** lösen **`ActionEvents`** aus.
Sie werden von **`ActionListener`**-Objekten beobachtet, mit einer einzigen Methode:

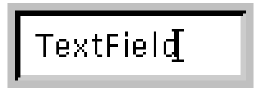
```
public void actionPerformed (ActionEvent e) {...}
```

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

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

class ButtonTest extends Frame
{  private Button waitButton, rebootButton; int state = 0;
   ButtonTest (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 **`ButtonTest`**-Objekt - nicht auf das unmittelbar umgebende **`ActionListener`**-Objekt.

Eingabe von Texten



Komponente `TextField`: einzeiliger, editierbarer Text

Ereignisse: **`ActionEvent`** (wie bei **`Button`**) ausgelöst bei der Eingabe von **`return`**

einige Methoden (aus der Oberklasse TextComponent):

`String getText ()`	Textinhalt liefern
`void setText (String v)`	Textinhalt setzen
`void setEditable (boolean e)`	Edierbarkeit festlegen
`boolean isEditable ()`	Edierbarkeit
`void selectText (int start, int end)`	Zeichenbereich anzeigen

Typischer **`ActionListener`**:

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

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

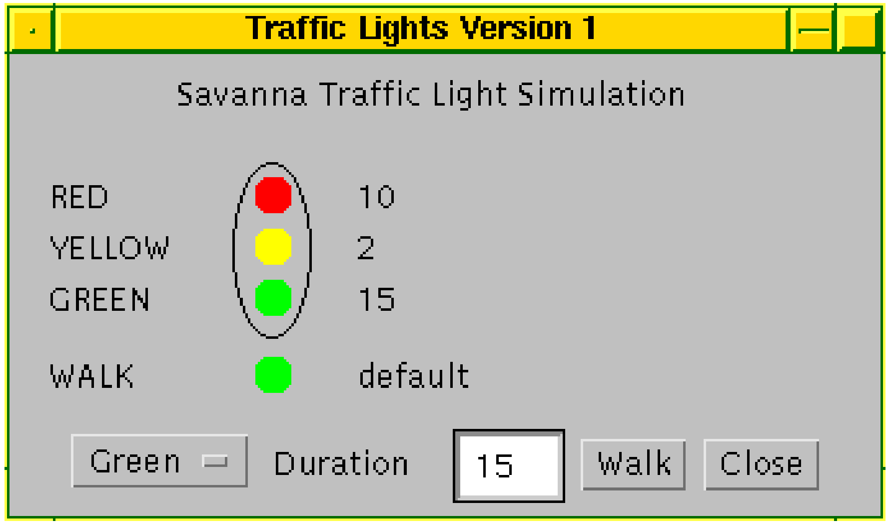
```
try { age = Integer.parseInt (str); }
catch (NumberFormatException e) { ... }
```

6. Beispiel: Ampel-Simulation

Aufgabe: Graphische Benutzungsoberfläche für eine Ampel-Simulation entwerfen

Eigenschaften:

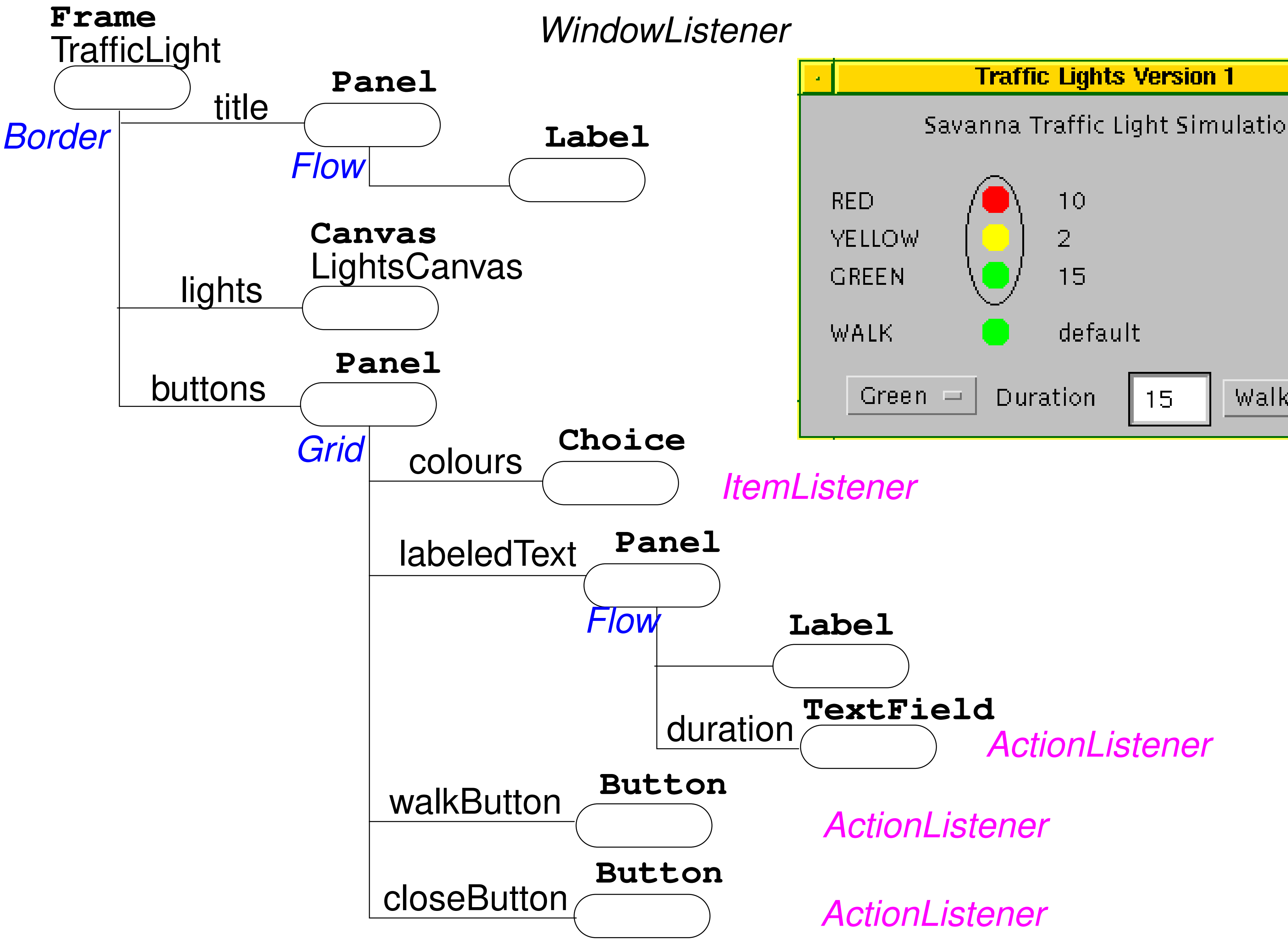
- Ampel visualisieren
  mit Knopf und Licht für Fußgänger
  (später auch animieren)
- Phasenlängen der Lichter
  einzeln einstellbar
- Einstellungen werden angezeigt



Entwicklungsschritte:

- Komponenten strukturieren
- zeichnen der Ampel (**`paint`** auf **`Canvas`**)
- 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 Frame
{                                  // The Traffic Light program by J M Bishop Oct 1997
                                                  // revised by U Kastens Jan 1999

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

  public TrafficLight ()                            // Konstruktor der zentralen Klasse
  {                                                      // Aufbau des Objektbaumes:
    setLayout (new BorderLayout ());              // Layout des Wurzelobjektes

    Panel title = new Panel ();                                  // Zentrierter Titel:
    title.setLayout (new FlowLayout (FlowLayout.CENTER));
    title.add (new Label("Savanna Traffic Light Simulation"));
    add (title, BorderLayout.NORTH);

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

Auswahl-Komponente

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

```
Choice colours = new Choice (); // Choice of a light

colours.addItem ("Red"); colours.addItem ("Yellow");
colours.addItem ("Green"); colours.addItem ("Walk");

colours.addItemListener
   (new ItemListener ()
      {  public void itemStateChanged (ItemEvent e)
         {  String s = (String) e.getItem ();

            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 **`Event`**-Parameter kann man die Zeichenreihe des gewählten Elementes zugreifen.

Eingabe der Phasenlänge

Eingabe mit einem **`TextField`**. Reaktion auf ein **`ActionEvent`**:

```
Panel labeledText = new Panel ();//fasst TextField und Beschriftung zusammen

labeledText.setLayout (new FlowLayout (FlowLayout.LEFT));
labeledText.add(new Label("Duration"));

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

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

labeledText.add (duration);
```

Button-Zeile

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

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

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

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

}// TrafficLight Klasse
```

Ampel zeichnen und beschriften

Eine Unterklasse der AWT-Klasse **`Canvas`** stellt eine **Zeichenfläche** bereit.
Die Methode **`paint`** wird zum Zeichnen und Beschriften überschrieben:

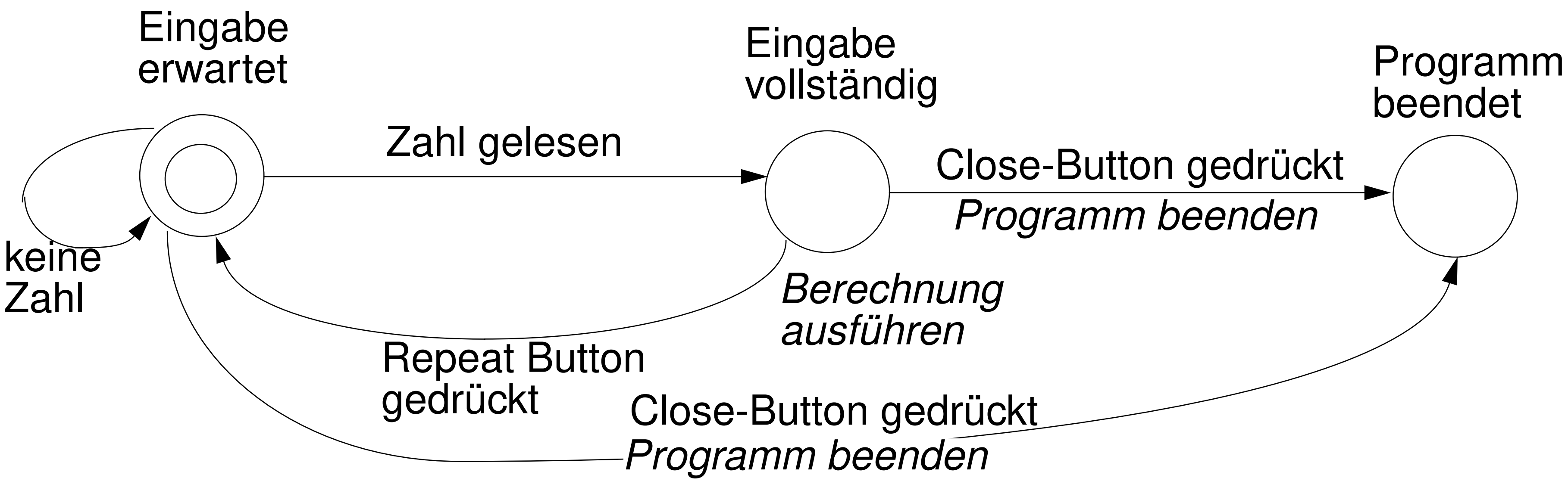
```
class LightsCanvas extends Canvas
{  private String [] msg;

   LightsCanvas (String [] m)                    // Die Array-Elemente enthalten die
   { msg = m; }                                  // Phasendauern der Lampen. Sie können
                                                 // durch Eingaben verändert werden.
   public void paint (Graphics g)
   {  g.drawOval (87, 10, 30, 68);               // 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);
                                   // eingegebenen 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)

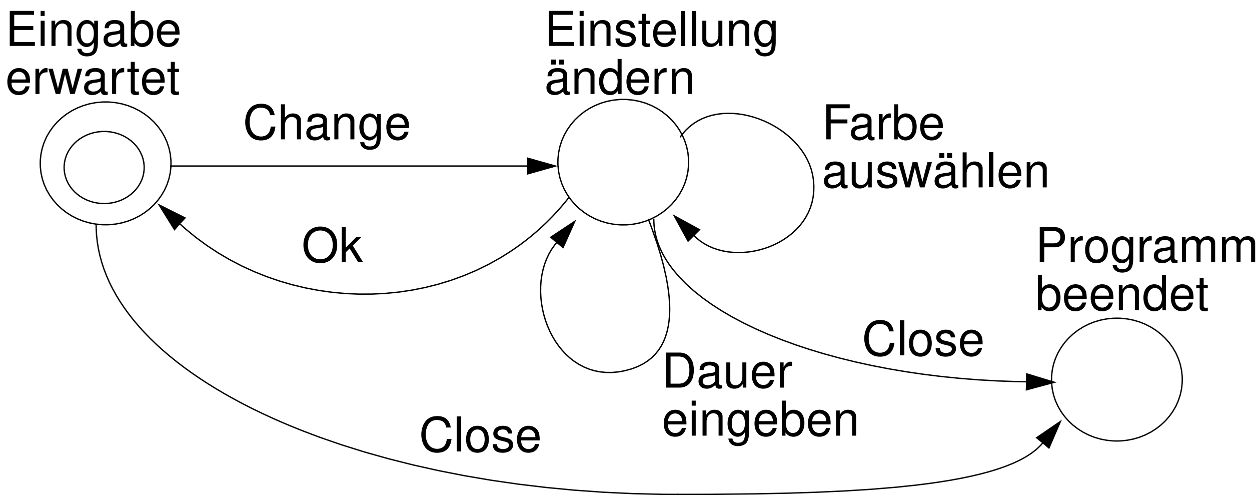
- **Zustände** unterscheiden Bediensituationen (z. B. „Eingabe erwartet“, „Eingabe vollständig“)
- **Übergänge** werden durch Ereignisse ausgelöst.
- **Aktionen** können mit Übergängen verknüpft werden; Reaktion auf ein Ereignis z. B. bei Eingabe einer Phasenlänge Ampel neu zeichnen, und
- **Aktionen** 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`**:

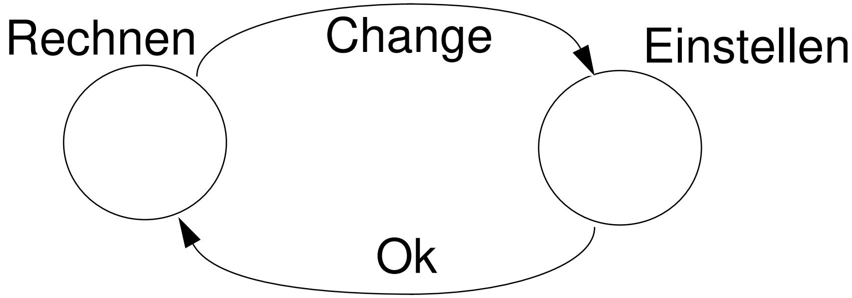


Robuste Programme dürfen auch an unzulässigen Ereignisfolgen nicht scheitern.
Verschiedene Möglichkeiten für **nicht definierte Übergänge**:

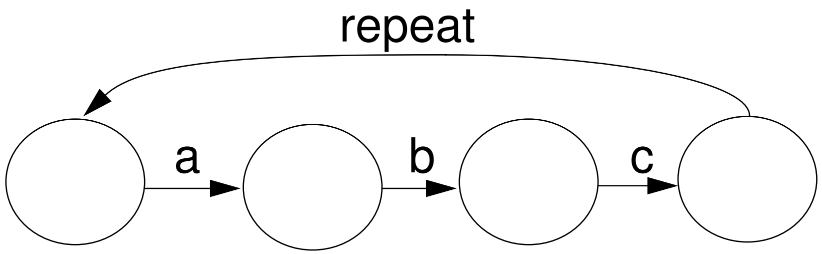
- Sie bleiben **ohne Wirkung**
- Sie bleiben ohne Wirkung und es wird eine **Erklärung** gegeben (Warnungsfenster).
- **Komponenten** werden so **deaktiviert**, dass unzulässige Ereignisse nicht ausgelöst werden können.

Muster für Ereignisfolgen

- Verschiedene **Bedienungsarten** (mode):
  Vorsicht: Nicht unnötig viele Zustände entwerfen. „Don‘t mode me in!“

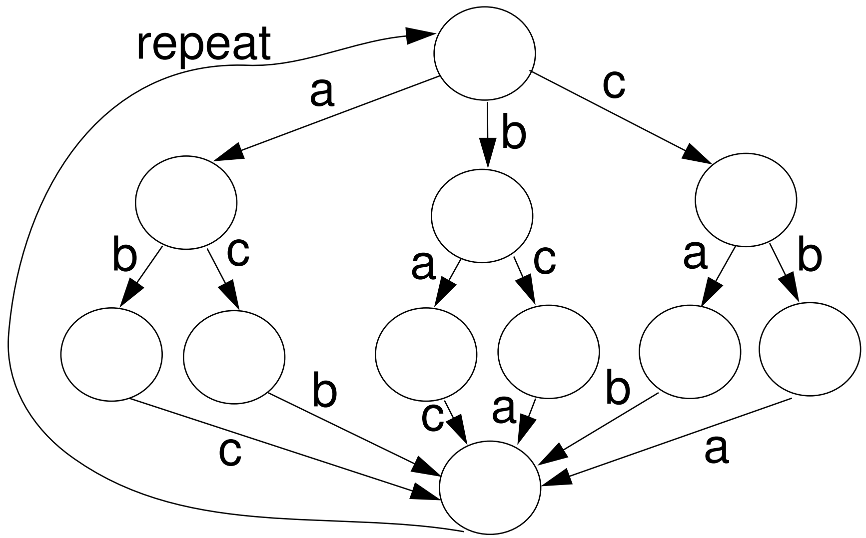


- Festgelegte **sequentielle Reihenfolge**:
  Vorsicht: Nicht unnötig streng vorschreiben. Bediener nicht gängeln.



- **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 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:

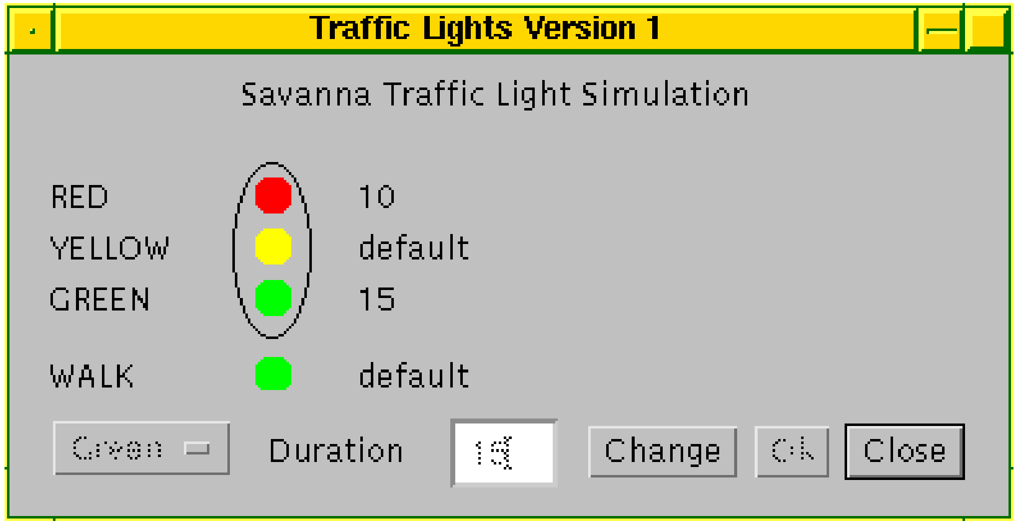
```
changeButton.addActionListener
   (new ActionListener ()
      {  public void actionPerformed(ActionEvent e)
         { toState (settingState); }});
```

Aktionen der Zustände in **Übergangsmethode** plazieren, 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: ...
         break;
      default: ...
}  }
```

Ampel-Simulation mit kontrollierten Zustandsübergängen

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

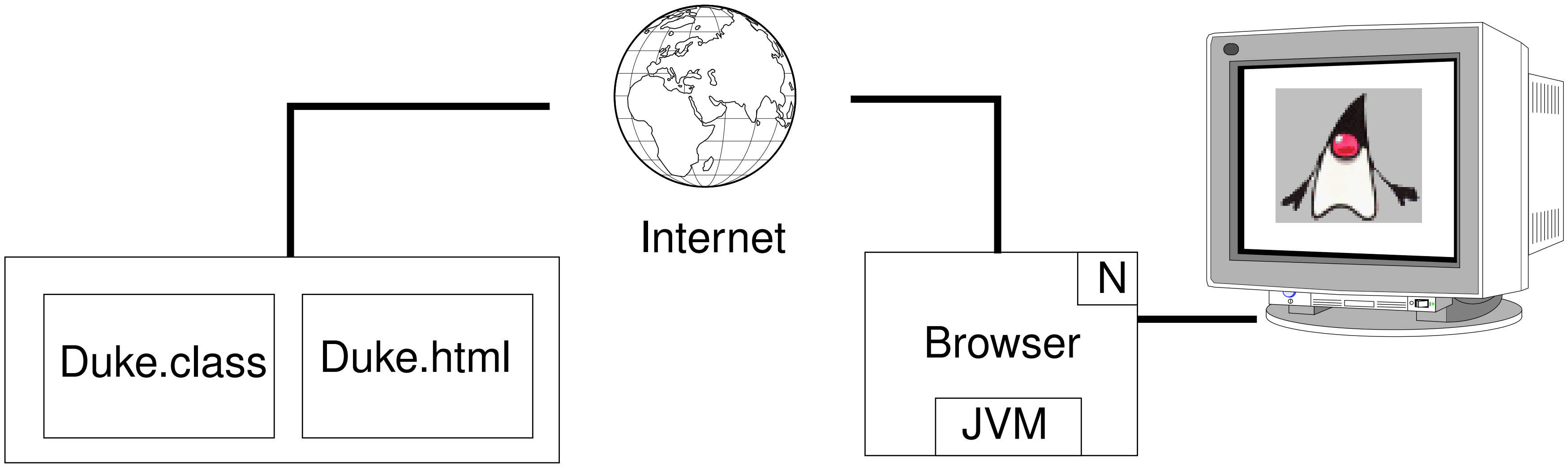


Die Komponenten zur Farbauswahl, Texteingabe und der Ok-Knopf sind im gezeigten Zustand deaktiviert.

8. Applets

Applet (small application):

- kleines Anwendungsprogramm in Java für eine spezielle Aufgabe,
- an eine WWW-Seite (world wide web) gekoppelt;
- das Applet wird mit der WWW-Seite über das Internet übertragen;
- der Internet Browser (Werkzeug zum Navigieren und Anzeigen von Informationen) enthält eine Java Virtual Machine (JVM); sie führt eintreffende Applets aus.



Programm (Java-Applet) wird übertragen, läuft beim Empfänger, bewirkt dort Effekte.

Applets benutzen Bibliotheken des Empfängers.

Programmierung von Applets

Applets werden wie Programme mit AWT-Benutzungsoberfläche geschrieben, aber:

- Die Hauptklasse ist Unterklasse von **`Applet`** statt von **`Frame`**.
- Es gibt die Methode **`main`** nicht.
- **`exit`** darf nicht aufgerufen werden.
- Die Methode **`public void init ()`** tritt an die Stelle des Konstruktors.
- Ein- und Ausgabe mit AWT-Komponenten statt mit Dateien.

Programmschema: Fenster mit Zeichenfläche (SWE-88):

```
class WarningApplet extends Applet
{ ...
   public void paint (Graphics g)
   { ... auf g schreiben und zeichnen ...}
}
```

Programmschema: Bedienung mit AWT-Komponenten (SWE-103):

```
class TrafficLight extends Applet
{  public void init ()
   { ... Objektbaum mit AWT-Komponenten aufbauen ... }
   ...
}
```

Applet-Methoden

Die Klasse **`Applet`** liegt in der Hierarchie der AWT-Komponenten:

```
Component
   Container
      Panel
         Applet
```

`Applet` definiert Methoden ohne Effekt zum Überschreiben in Unterklassen:

	wird aufgerufen, wenn das Applet ...
`void init ()`	geladen wird
`void start ()`	seine Ausführung (wieder-)beginnen soll
`void stop ()`	seine Ausführung unterbrechen soll
`void destroy ()`	das Applet beendet wird

Methoden zum Aufruf in Unterklassen von **`Applet`**, z. B.

`void showStatus (String)`	Text in der Statuszeile des Browsers anzeigen
`String getParameter (String)`	Wert aus HTML-Seite lesen

Weitere Methoden aus Oberklassen, z. B

`void paint (Graphics g)`	auf **`g`** schreiben und zeichnen (aus **`Container`**)

Applets zu HTML-Seiten

HTML (Hypertext Markup Language):

- Sprache zur Beschreibung formatierter, strukturierter Texte und deren Gestaltung
- Standardsprache zur Beschreibung von Internet-Seiten
- Einfache Sprachstruktur: Marken `<li>` und Klammern `<ul>` ... `</ul>` mit bestimmter Bedeutung, z. B.

```
Wir unterscheiden
<ul>
   <li> diesen Fall,
   <li> jenen Fall
</ul>
und viele andere Fälle.
```

Wir unterscheiden

- diesen Fall,
- jenen Fall

und viele andere Fälle.

Ein Applet wird auf einer HTML-Seite aufgerufen, z. B.

```
<title>Catch the Duke!</title>
<hr>
<applet code=CatchM.class width=300 height=300>
</applet>
```

Beim Anzeigen der HTML-Seite im Browser oder Appletviewer wird das Applet auf einer Fläche der angegebenen Größe ausgeführt.

Java-Programm in Applet umsetzen

Ein Java-Programm kann man in folgenden Schritten in ein Applet transformieren:

1. Alle Datei-E/A in Benutzung von AWT-Komponenten umsetzen, z. B. **`System.out.println(...)`** in **`g.drawstring(...)`** in der Methode **`paint`**.
2. Programm nicht anhalten, **`exit`**-Aufrufe und **`close`**-Button entfernen.
3. Layoutmanager explizit wählen, Default ist **`FlowLayout`**.
4. **`java.applet.*`** importieren,
   Hauptklasse als Unterklasse von **`Applet`** definieren
5. Konstruktor durch **`init`**-Methode ersetzen; darin wird der Objektbaum der Komponenten aufgebaut.
6. **`main`**-Methode entfernen;
   die Hauptklasse des Java-Programms entfällt einfach, wenn sie nur die **`main`**-Methode enthält und darin nur das **`Frame`**-Objekt erzeugt und plaziert wird (wie im Beispiel **`Traffic`**).
7. HTML-Datei herstellen mit Aufruf der **`.class`**-Datei.
8. Testen des Applets; erst mit dem **`appletviewer`** dann mit dem Browser.

siehe Beispiel Ampel-Simulation SWE-104 bis 108.

Parameter zum Start des Applet

Dem Applet können zum Aufruf von der HTML-Seite Daten mitgegeben werden.

Notation: Paare von Zeichenreihen für Parametername und Parameterwert

```
<PARAM NAME="Parametername" VALUE="Parameterwert">
```

eingesetzt im Applet-Aufruf:

```
<APPLET CODE="COffeeShop.class" WIDTH=600 HEIGHT=200>
<PARAM NAME="Columbian" VALUE="12">
<PARAM NAME="Java" VALUE="15">
<PARAM NAME="Kenyan" VALUE="9">
</APPLET>
```

Im Applet auf die Parameterwerte zugreifen:

```
preis = Integer.parseInt (getParameter ("Java"));
```

Sicherheit und Applets

Beim Internet-Surfen kann man nicht verhindern, daß fremde Applets auf dem eigenen Rechner ausgeführt werden. Deshalb sind ihre **Rechte i. a. stark eingeschränkt**:

Operation	Java Programm	Applet im applet viewer	lokales Applet im Browser	fremdes Applet im Browser
lokale Datei zugreifen	X	X		
lokale Datei löschen	X			
anderes Programm starten	X	X		
Benutzernamen ermitteln	X	X	X	
zum Sender des Applet verbinden	X	X	X	X
zu anderem Rechner verbinden	X	X	X	
Java Bibliothek laden	X	X	X	
exit aufrufen	X	X		
Pop-up Fenster erzeugen	X	X	X	X

Diese Einstellungen der Rechte können im Browser geändert werden.

Man kann auch **signierten Applets** bestimmter Autoren weitergehende Rechte geben.

Außerdem wird der **Bytecode** jeder Klasse auf Konsistenz **geprüft**, wenn er in die JVM geladen wird.

Wiederholung: Objektorientierte Vererbung

Grundlegende Aspekte objektorientierter Vererbung werden wiederholt an Hand von Folien aus

U. Kastens: Vorlesung SWE, 1998/99 (aktualisiert), http://ag-kastens.upb.de/lehre/material/swei

- 18 - 20: Klassen und Objekte
- 78 - 82b: Vererbung

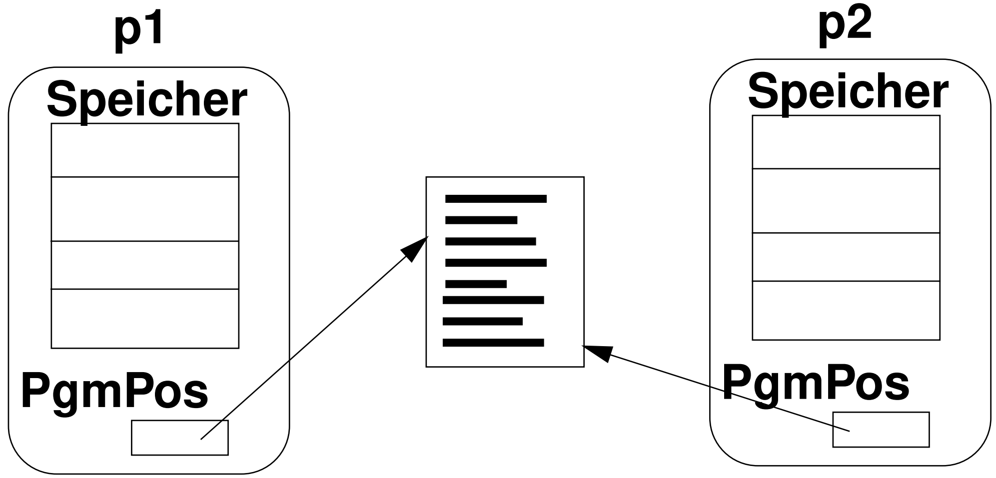
Folgender Abschnitt wird ebenso zur Wiederholung angeraten:

- 73 - 77b: Objektorientierte Abstaktion, Schnittstellen

9. Parallele Prozesse
Grundbegriffe (1)

Prozess:
Ausführung eines sequentiellen Programmstückes in dem zugeordneten Speicher (Adressraum).
Zustand des Prozesses:
Speicherinhalt und Programmposition

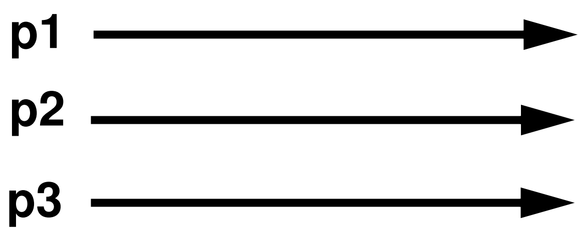
Zustände zweier Prozesse zu demselben Programm



sequentieller Prozess: Die Operationen werden eine nach der anderen ausgeführt.

Parallele Prozesse: mehrere Prozesse, die **gleichzeitig auf mehreren Prozessoren** ausgeführt werden;

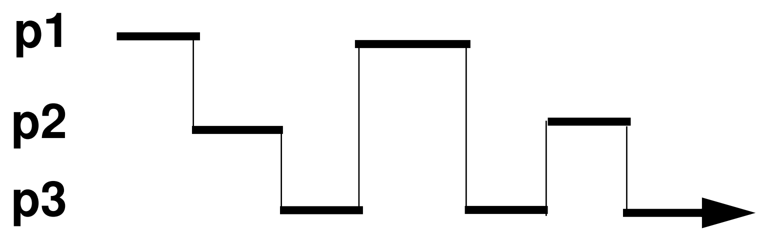
- keine Annahme über relative Geschwindigkeit
- **unabhängige** Prozesse oder
- **kommunizierende** Prozesse:
  p1 liefert Daten für p2 oder
  p1 wartet bis eine Bedingung erfüllt ist



Grundbegriffe (2)

verzahnte Prozesse: mehrere Prozesse, die stückweise **abwechselnd auf einem Prozessor** ausgeführt werden

Prozessumschaltung durch den Scheduler des Betriebssystems, der JVM oder durch die Prozesse selbst.



Ziele:

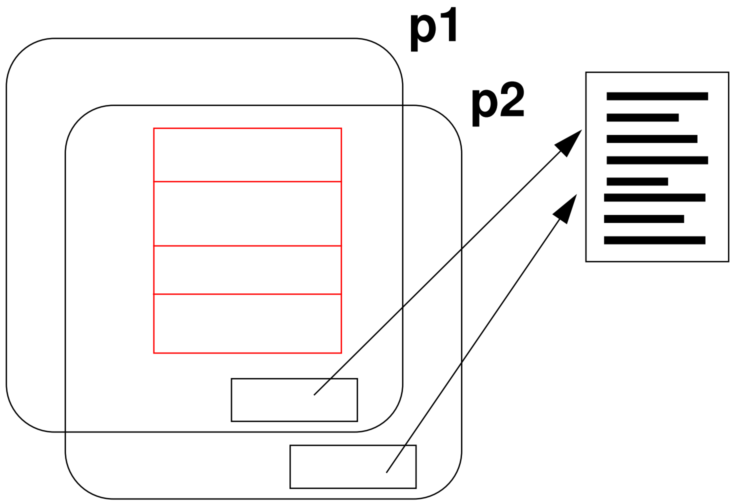
- Prozessor auslasten,
  blockierte Prozesse nicht bearbeiten
- Gleichzeitigkeit simulieren

nebenläufige Prozesse:
Prozesse, die parallel oder verzahnt ausgeführt werden können

Grundbegriffe (3)

Threads („Fäden“ der Programmausführung):
Prozesse, die parallel oder verzahnt in **gemeinsamem Speicher** ablaufen.

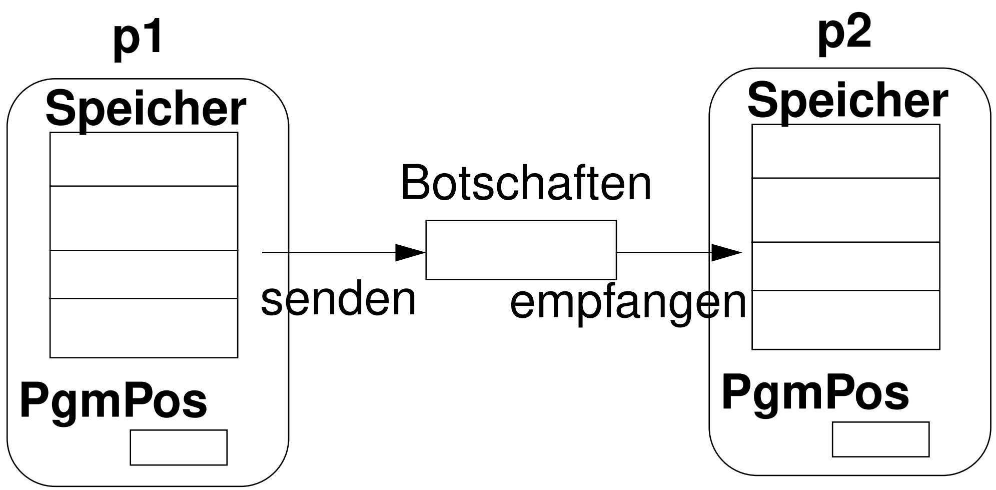
Die **Prozessumschaltung** ist besonders einfach und **schnell,**
daher auch **leichtgewichtige Prozesse**



Zugriff auf Variable im gemeinsamen Speicher

Im Gegensatz zu
Prozessen im verteilten Speicher auf verschiedenen Rechnern,

kommunizieren über Botschaften, statt über Variable im Speicher



Anwendungen Paralleler Prozesse

- **Benutzungsoberflächen**:
  Die Ereignisse werden von einem speziellen Systemprozess weitergegeben.
  Aufwendige Berechnungen sollten nebenläufig programmiert werden, damit die Bedienung der Oberfläche nicht blockiert wird.
- **Simulation** realer Abläufe:
  z. B. Produktion in einer Fabrik
- **Animation**:
  Veranschaulichung von Abläufen, Algorithmen; Spiele
- **Steuerung** von Geräten:
  Prozesse im Rechner überwachen und steuern externe Geräte, z. B. Montage-Roboter
- **Leistungssteigerung** durch Parallelrechner:
  mehrere Prozesse bearbeiten gemeinsam die gleiche Aufgabe,
  z. B. paralleles Sortieren großer Datenmengen.

Java Threads erzeugen (1)

Prozesse, die nebenläufig, gemeinsam im Speicher des Programms bzw. Applets ablaufen.

Es gibt 2 Techniken Threads zu erzeugen. Erste Technik:

Eine Benutzerklasse implementiert das Interface **`Runnable`**:

```
class Aufgabe implements Runnable
{ ...
   public void run ()               vom Interface geforderte Methode run
   {...}                            das als Prozess auszuführende Programmstück
   Aufgabe (...) {...}              Konstruktor
}
```

Der Prozess wird als Objekt der vordefinierten Klasse Thread erzeugt:

```
Thread auftrag = new Thread (new Aufgabe (...));
```

Erst folgender Aufruf startet dann den Prozess:

`auftrag.start();` Der neue Prozess beginnt, neben dem hier aktiven zu laufen.

Diese Technik (das Interface **`Runnable`** implementieren) sollte man anwenden, wenn der **abgespaltene Prozess nicht weiter beeinflußt** werden muss; also einen Auftrag erledigt (Methode **`run`**) und dann terminiert.

Java Threads erzeugen (2)

Zweite Technik:

Die Benutzerklasse wird als Unterklasse der vordefinierten Klasse **`Thread`** definiert:

```
class DigiClock extends Thread
{ ...
   public void run ()              überschreibt die Thread-Methode run
   {...}                           das als Prozess auszuführende Programmstück
   DigiClock (...) {...}           Konstruktor
}
```

Der Prozess wird als Objekt der Benutzerklasse erzeugt (es ist auch ein **`Thread`**-Objekt):

```
Thread clock = new DigiClock (...);
```

Erst folgender Aufruf startet dann den Prozess:

```
clock.start();          der neue Prozess beginnt neben dem hier aktiven zu laufen
```

Diese Technik (Unterklasse von Thread) sollte man anwenden, wenn
der abgespaltene Prozess weiter beeinflußt werden soll;
also weitere Methoden der Benutzerklasse definiert und von aussen aufgerufen werden,
z. B. zum vorübergehenden Anhalten oder endgültigem Terminieren.

10. Unabhängige parallele Prozesse

Beispiel: Digitale Uhr als Prozess im Applet (1)

Der Prozess soll in jeder Sekunde Datum und Uhrzeit als Text aktualisiert anzeigen.

```
class DigiClock extends Thread
{  public void run ()
   {  while (running)                          iterieren bis von außen terminiert wird
      {  line.setText(new Date().toString());                      Datum schreiben
         try { sleep (1000); } catch (Exception ex) {}                       Pause
      }
   }
                                       Methode, die den Prozeß von außen terminiert:
   public void stopIt () { running = false; }
   private boolean running = true;                               Zustandsvariable

   public DigiClock (Label t) {line = t;}   Label zum Beschreiben übergeben
   private Label line;
}
```

Prozess **als Unterklasse** von **`Thread`**, weil er

- durch Aufruf von **`stopIt`** terminiert und
- durch weitere **`Thread`**-Methoden unterbrochen werden soll.

Beispiel: Digitale Uhr als Prozess im Applet (2)

Der Prozess wird in der **init**-Methode der **Applet**-Unterklasse erzeugt:

```
public class DigiApp extends Applet
{  public void init ()
   {  Label clockText = new Label ("--------------------------------");
      add (clockText);

      clock = new DigiClock (clockText);                  Prozess erzeugen
      clock.start();                                      Prozess starten
   }

   public void start ()   { clock.resume(); }            Prozess fortsetzen
   public void stop ()    { clock.suspend(); }           Prozess anhalten
   public void destroy () { clock.stopIt(); }            Prozess terminieren

   private DigiClock clock;
}
```

Prozesse, die in einem Applet gestartet werden,

- sollen angehalten werden (**suspend**, **resume**), solange das Applet nicht sichtbar ist (**stop**, **start**),
- müssen terminiert werden (**stopIt**), wenn das Applet entladen wird (**destroy**).

Andernfalls belasten Sie den Rechner, obwohl sie nicht mehr sichtbar sind.

Wichtige Methoden der Klasse `Thread`

`public void run ();`

wird überschrieben mit der Methode, die die auszuführenden Anweisungen enthält

`public void start ();`

startet die Ausführung des Prozesses

`public void suspend ();`

hält den angegebenen Prozess an: **`clock.suspend();`**

`public void resume ();`

setzt den angegebenen Prozess fort: **`clock.resume();`**

`public void join () throws InterruptedException;`

der aufrufende Prozess wartet bis der angegebene Prozess terminiert ist:

```
try { auftrag.join(); } catch (Exception e){}
```

`public static void sleep (long millisec) throws InterruptedException;`

der aufrufende Prozess wartet mindestens die in Millisekunden angegebene Zeit:

```
try { Thread.sleep (1000); } catch (Exception e){}
```

`public final void stop () throws SecurityException;`

nicht benutzen! Terminiert den Prozess u. U. in einem inkonsistenten Zustand

Beispiel: Ampelsimulation (1)

Aufgabe:

- Auf einer Zeichenfläche mehrere Ampeln darstellen.
- Der Prozess jeder Ampel durchläuft die Ampelphasen und zeichnet die Lichter entsprechend.
- Vereinfachungen:
  1 Ampel; keine Einstellung der Phasendauer; keine Grünanforderung für Fußgänger.

Lösung:

- Ampel-Objekte aus einer Unterklasse (**`SetOfLights`**) von **`Thread`** erzeugen (Methode 2).
- Ampel-Objekte zeichnen auf gemeinsamer Zeichenfläche (**`Canvas`**-Objekt).
- **`run`**-Methode wechselt die Ampelphasen und
  ruft eine Methode zum Zeichnen der Lichter auf (**`drawLights`**).
- Eine weitere Methode (**`draw`**) zeichnet die unveränderlichen Teile des Ampel-Bildes.
- Die **`paint`**-Methode der Zeichenfläche ruft beide Zeichenmethoden der Ampel-Objekte auf
  (Delegation der Zeichenaufgabe).

Prozessklasse der Ampelsimulation

```
class SetOfLights extends Thread
{  /* Jedes Objekt dieser Klasse zeichnet eine Ampel an vorgegebener
      Position x auf ein gemeinsames Canvas-Objekt area.  */

   private Canvas area;                          // gemeinsame Zeichenfläche
   private int x;                                // X-Koordinate dieser Ampel
   private int light = 0;                        // aktuelle Ampelphase

   public SetOfLights (Canvas area, int x)
   { this.area = area; this.x = x;}

   public void run ()
   {  while (running)                            // drei Ampelphasen wiederholen
      {  for (light = 0; light < 3; light++)
         {  drawLights ();                       // neuen Zustand zeichnen
            try { sleep(500); }         // nach fester Dauer Zustand wechseln
            catch (InterruptedException e) { }
   }  }  }

   public void stopIt () { running = false; }// Prozess von außen anhalten
   private boolean running = true;

   public void draw (Graphics g) {...}   // unveränderliche Bildteile zeichnen
   public void drawLights ()     {...}      // veränderliche Bildteile zeichnen
}
```

Zeichenmethoden der Ampelsimulation

```
class SetOfLights extends Thread
{  ...
   public void draw (Graphics g)
   {              // unveränderliche Teile des Bildes relativ zu x zeichnen und schreiben:
      g.drawOval(x-8, 10, 30, 68);                          // der Ampelumriss
   }

   public void drawLights ()
   {                                    // veränderliche Teile des Bildes zeichnen:
      Graphics g = area.getGraphics();//das Graphics-Objekt der Zeichenfläche

      if (light == 0) g.setColor(Color.red);                   // die 4 Lichter
      else g.setColor(Color.lightGray);
      g.fillOval(x, 15, 15, 15);

      if (light == 1) g.setColor(Color.yellow);
      else g.setColor(Color.lightGray);
      g.fillOval(x, 35, 15, 15);

      if (light == 2) g.setColor(Color.green);
      else g.setColor(Color.lightGray);
      g.fillOval(x, 55, 15, 15);

      g.setColor(Color.green); g.fillOval(x, 85, 15, 15);
}  }
```

Applet-Klasse der Ampelsimulation

```
public class TrafficLight extends Applet
{  private Canvas area;                          // gemeinsame Zeichenfläche
   private int lightsPosition = 105;      // X-Koordinate der nächsten Ampel
   private SetOfLights lights;                     // zunächst nur eine Ampel

   public void init ()
   {  setLayout (new BorderLayout ());
      area =                        // das Canvas-Objekt zum Zeichnen der Ampeln
       new Canvas()   // anonyme Unterklasse mit 1 Objekt:
        { public void paint (Graphics g) // Zeichnen an Ampel-Objekt deligieren
          { lights.draw (g);                         // zeichne statischen Teil
            lights.drawLights ();                 // zeichne veränderlichen Teil
        }};
      add (area, BorderLayout.CENTER);

      lights = new SetOfLights (area, lightsPosition); // 1 Ampel-Objekt
      area.repaint ();                              // erstes Zeichnen anstoßen
      lights.start ();                                       // Prozess starten
   }
                            // Prozesse des Applets anhalten, fortsetzen, beenden:
   public void stop ()    { lights.suspend (); }
   public void start ()   { lights.resume (); }
   public void destroy () { lights.stopIt (); }
}
```

11. Monitore, Synchronisation: Gegenseitiger Ausschluss

Wenn mehrere Prozesse die **Werte gemeinsamer Variablen verändern**, kann eine ungünstige Verzahnung (oder echte Parallelausführung) zu inkonsistenten Daten führen.

Z. B. zwei Prozesse benutzen lokale Variable **`tmp`** und eine gemeinsame Variable **`konto`**:

```
p    tmp = konto;                                      konto = tmp+10;

q                    tmp = konto; konto = tmp+10;
```

Kritische Abschnitte: zusammengesetzte Operationen, die gemeinsame Variablen lesen und/oder verändern.

Prozesse müssen kritische Abschnitte unter **gegenseitigem Ausschluss (mutual exclusion)** ausführen:
D. h. **zu jedem Zeitpunkt darf höchstens ein Prozess** solch einen kritischen Abschnitt für bestimmte Variablen ausführen. Andere Prozesse, die für die gleichen Variablen mit der Ausführung eines kritischen Abschnitts beginnen wollen, müssen warten.

Gegenseitiger Ausschluss durch Monitore

Monitor:

Ein Modul, der Daten und Operationen darauf kapselt.
Die **kritischen Abschnitte** auf den Daten werden als **Monitor-Operationen** formuliert.
Prozesse rufen Monitor-Operationen auf, um auf die Daten zuzugreifen.
Die Monitor-Operationen werden unter gegenseitigem Ausschluss ausgeführt.

Monitore in Java:

Methoden einer Klasse, die kritische Abschnitte auf Objektvariablen implementieren, können als `synchronized` gekennzeichnet werden:

```
class Bank
{  synchronized public void abbuchen (...)   {...}
   synchronized public void überweisen (...) {...}
   ...
   private int[] konten;
}
```

Jedes Objekt der Klasse wirkt dann als **Monitor** für seine Objektvariablen:
Aufrufe von `synchronized` Methoden von mehreren Prozessen für dasselbe Objekt werden unter **gegenseitigem Ausschluss** ausgeführt.
Zu jedem Zeitpunkt kann höchstens 1 Prozess mit `synchronized` Methoden auf Objektvariable zugreifen.

Beispiel: gegenseitiger Ausschluss durch Monitor

```
class Bank
{  synchronized public void abbuchen (String name)  // kritischer Abschnitt
   {  int vorher = konto;
      System.out.println (name + " Konto vor : " + vorher);
                                   // kritische Verwendung der Objektvariablen konto:
      konto = vorher - 10;
      System.out.println (name + " Konto nach: " + konto);
   }
   private int konto = 100;
}

class Kunde extends Thread
{  Kunde (String str, Bank b) { name = str; meineBank = b; }
   public void run ()
   {  for (int i = 1; i <= 2; i++) meineBank.abbuchen(name); }
   private Bank meineBank; private String name;
}

class BankPgm
{  public static void main (String [] args)
   {  Bank b = new Bank();          // Objekt als Monitor für seine Objektvariable
      new Kunde("ich", b).start(); new Kunde("du ", b).start();
}  }
```

12. Bedingungssynchronisation im Monitor

Bedingungssynchronisation:

Ein Prozess wartet, bis eine Bedingung erfüllt ist - verursacht durch einen anderen Prozess; z. B. erst dann in einen Puffer schreiben, wenn er nicht mehr voll ist.

Bedingungssynchronisation im Monitor:

Ein Prozess, der in einer kritischen Monitor-Operation auf eine Bedingung wartet, muss den Monitor freigeben, damit andere Prozesse den Zustand des Monitors ändern können.

Bedingungssynchronisation in Java:

Vordefinierte Methoden der Klasse **`Object`** zur Bedingungssynchronisation:
(Sie müssen aus **`synchronized`** Methoden heraus aufgerufen werden.)

- **`wait()`** **blockiert den aufrufenden Prozess und gibt den Monitor frei**
  - das Objekt, dessen **`synchronized`** Methode er gerade ausführt.
- **`notifyAll()`** **weckt alle in diesem Monitor blockierten Prozesse**;
  sie können weiterlaufen, sobald der Monitor frei ist.
- **`notify()`** weckt einen blockierten Prozess;
  ist für unser Monitorschema nicht brauchbar

Nachdem ein blockierter Prozess geweckt wurde, muß er die **Bedingung**, auf die er wartet, **erneut prüfen** - sie könnte schon durch schnellere Prozesse wieder invalidiert sein:

```
while (avail<n) try { wait(); } catch (InterruptedException e) {}
```

Beispiel: Monitor zur Vergabe von Ressourcen

Aufgabe: Eine Begrenzte Anzahl gleichartiger Ressourcen wird von einem Monitor verwaltet. Prozesse fordern einige Ressourcen an und geben sie später zurück.

```
class Monitor
{  private int avail;                                    // Anzahl verfügbarer Ressourcen
   Monitor (int a) { avail = a; }

   synchronized void getElem (int n, int who)            // n Elemente abgeben
   {  System.out.println("Client"+who+" needs "+n+",available "+avail);

      while (avail < n)                                  // Bedingung in Warteschleife prüfen
      { try { wait(); } catch (InterruptedException e) {}
      }                                 // try ... catch nötig wegen wait()
      avail -= n;
      System.out.println("Client"+who+" got "+n+", available "+avail);
   }

   synchronized void putElem (int n, int who)    // n Elemente zurücknehmen
   {  avail += n;
      System.out.println("Client"+who+" put "+n+",available "+avail);
      notifyAll();                  // Alle Wartenden können die Bedingung erneut prüfen
   }
}
```

Beispiel: Prozesse und Hauptprogramm zu SWE-146

```
import java.util.Random;

class Client extends Thread
{  private Monitor mon; private Random rand;
   private int ident, rounds, max;
   Client (Monitor m, int id, int rd, int avail)
   {  ident = id; rounds = rd; mon = m; max = avail;
      rand = new Random();            // Neuer Zufallszahlengenerator
   }

   public void run ()
   {  while (rounds > 0)
      {  int m = Math.abs(rand.nextInt()) % max + 1;
         mon.getElem (m, ident);    // m Elemente anfordern
         try { sleep (Math.abs(rand.nextInt()) % 1000 + 1);}
            catch (InterruptedException e) {}
         mon.putElem (m, ident);    // m Elemente zurückgeben
         rounds--;
      }
   }
}
```

```
class TestMonitor
{  public static void main (String[] args) {
      int avail = 20;
      Monitor mon = new Monitor (avail);
      for (int i = 0; i < 5; i++)
         new Client(mon,i, 4,avail).start();
}  }
```

Schema: Gleichartige Ressourcen vergeben

Ein **Monitor** verwaltet eine endliche Menge von k >= 1 **gleichartigen Ressourcen**.

Prozesse fordern jeweils unterschiedlich viele (n) Ressourcen an, 1<= n <= k, und geben sie nach Gebrauch wieder zurück.

Die **Wartebedingung** für das Anfordern ist „Sind n Ressourcen verfügbar?“

Zu jedem Zeitpunkt zwischen Aufrufen der Monitor-Methoden gilt:
„Die Summe der freien und der vergebenen Ressourcen ist k.“

Die Monitor-Klasse auf SWE-146 implementiert dieses Schema.

Beispiele:

- Walkman-Vergabe an Besuchergruppen im Museum (siehe [Java lernen, 13.5])
- Taxi-Unternehmen; n = 1

auch **abstrakte Ressourcen**, z. B.

- das Recht, eine Brücke begrenzter Kapazität (Anzahl Fahrzeuge oder Gewicht) zu befahren,

auch **gleichartige Ressourcen mit Identität**:
Der Monitor muß dann die Identitäten der Ressourcen in einer Datenstruktur verwalten.

- Nummern der vom Taxi-Unternehmen vergebenen Taxis
- Adressen der Speicherblöcke, die eine Speicherverwaltung vergibt

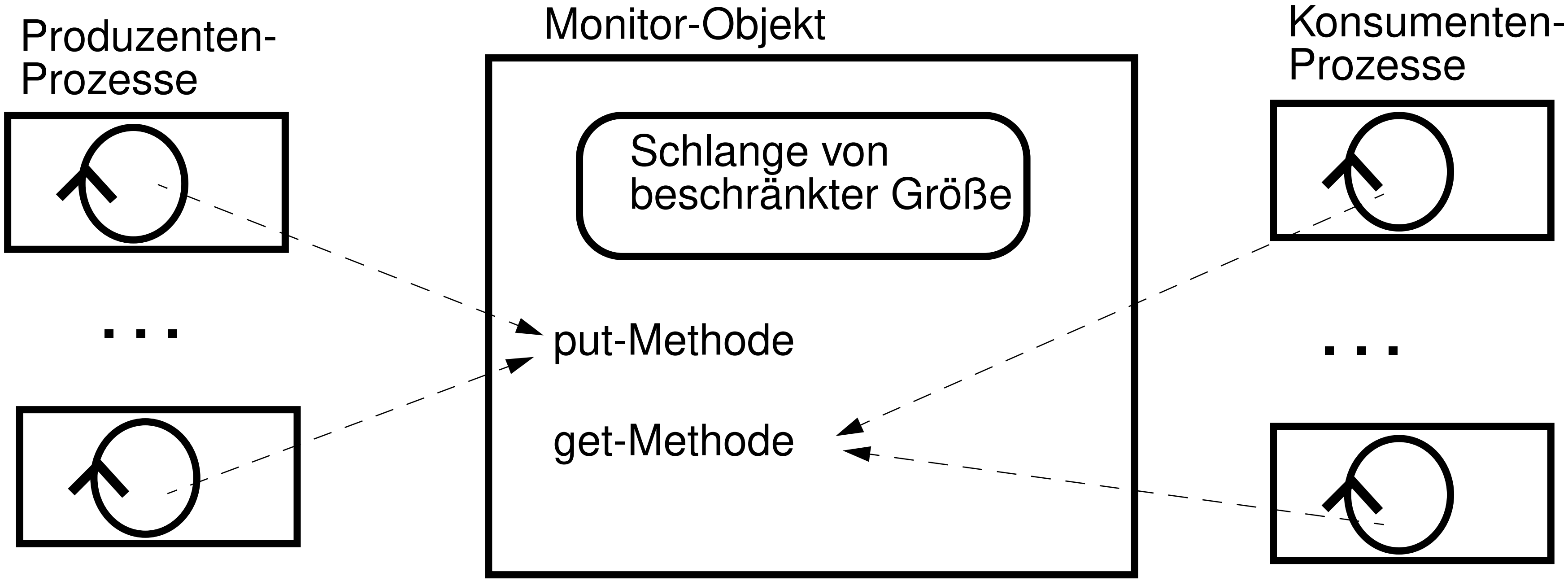
Schema: Beschränkter Puffer

Ein Monitor speichert Elemente in einem **Puffer von beschränkter Größe**.

Produzenten-Prozesse liefern einzelne Elemente,
Konsumenten-Prozesse entnehmen einzelne Elemente.

Der Monitor stellt sicher, daß die Elemente in der gleichen **Reihenfolge** geliefert und entnommen werden. (Datenstruktur: Schlange)

Die **Wartebedingungen** lauten:
- in den Puffer **schreiben, nur wenn er nicht voll** ist,
- **aus dem Puffer nehmen, nur wenn er nicht leer** ist.



Monitor-Klasse für Beschränkte Puffer

```
class Buffer
{  private Queue buf;      // Schlange der Länge n zur Aufnahme der Elemente
   public Buffer (int n) {buf = new Queue(n); }

   synchronized public void put (Object elem)
   {                     // ein Produzenten-Prozess versucht, ein Element zu liefern
      while (buf.isFull())                  // warten bis der Puffer nicht voll ist
         try {wait();} catch (InterruptedException e) {}
      buf.enqueue (elem);// Wartebedingung der get-Methode hat sich verändert
      notifyAll();          // jeder blockierte Prozess prüft seine Wartebedingung
   }
   synchronized public Object get ()
   {                    // ein Konsumenten-Prozess versucht, ein Element zu nehmen
      while (buf.isEmpty())                 // warten bis der Puffer nicht leer ist
         try {wait();} catch (InterruptedException e) {}
      Object elem = buf.first();
      buf.dequeue();        // Wartebedingung der put-Methode hat sich verändert
      notifyAll();          // jeder blockierte Prozess prüft seine Wartebedingung
      return elem;
   }
}
```

13. Verklemmungen

Ein **einzelner Prozess ist verklemmt** (in einer **Deadlock**-Situation), wenn er auf eine **Bedingung wartet, die nicht mehr wahr werden kann**.

Mehrere Prozesse sind untereinander verklemmt (in einer Deadlock-Situation), wenn sie **zyklisch aufeinander warten**;
d. h. die Wartebedingung eines Prozesses kann nur von einem anderen, auch wartenden Prozess erfüllt werden.

Verklemmung bei Ressourcenvergabe können eintreten,
wenn folgende Bedingungen gelten**:**

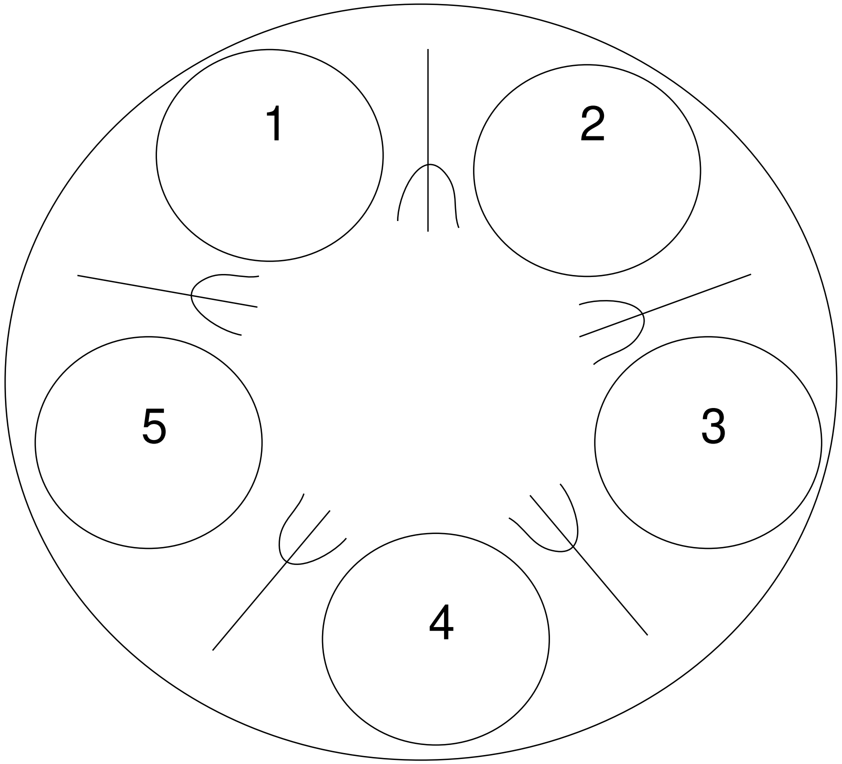
1. Jeder Prozess **fordert mehrmals nacheinander Ressourcen an**.
2. Jeder Prozess benötigt bestimmte Ressourcen **exklusiv** für sich allein.
3. Die **Relation** „Prozess p benötigt eine Ressource die Prozess q“ hat ist **zyklisch**.

Beispiel: Dining Philosophers (SWE-152)

Synchronisationsaufgabe: Dining Philosophers

Abstraktion von Prozesssystemen, in denen die Prozesse **mehrere Ressourcen exklusiv** benötigen. (Erstmals formuliert von E. Dijkstra 1968.)

5 Philosophen sitzen an einem gedeckten Tisch. Jeder beschäftigt sich abwechselnd mit denken und essen. Wenn ein Philosoph Hunger verspürt, versucht er die beiden Gabeln neben seinem Teller zu bekommen, um damit zu essen. Nachdem er gesättigt ist, legt er die Gabeln ab und verfällt wieder in tiefgründig philosophisches Nachdenken.



Abstraktes Programm für die Philosophen:

wiederhole
    denken
    nimm rechte Gabel
    nimm linke Gabel
    essen
    gib rechte Gabel
    gib linke Gabel

Es gibt einen Verklemmungszustand:

Für alle Prozesse i aus {1,..,5} gilt:
    Prozess i hat seine rechte Gabel
    und wartet auf seine linke Gabel.

Dining Philosophers, Lösung (1)

Lösungsprinzip: Ressourcen (Gabeln) **zugleich anfordern** - nicht nacheinander.

Ein **Monitor** verwaltet die Gabeln als **Ressourcen mit Identität**.
Monitor-Operation: Dem Prozess i die linke und rechte Gabel geben.
Wartebedingung dafür: Die beiden Gabeln für den i-ten Prozess sind frei.

```
class ResourcePairs
{                         // nur Paare aus „benachbarten“ Ressourcen werden vergeben
   private boolean[] avail = {true, true, true, true, true};

   synchronized void getPair (int i)
   {  while (!(avail[i % 5] & avail[(i+1) % 5]))
         try {wait();} catch (InterruptedException e) {}

      avail[i % 5] = false; avail[(i+1) % 5] = false;
   }

   synchronized void putPair (int i)
   {  avail[i % 5] = true; avail[(i+1) % 5] = true;
      notifyAll();
   }
}
```

Der Verklemmungszustand „Jeder Prozess hat **eine** Gabel.“ kann nicht eintreten.

Dining Philosophers, Lösung (2)

Lösungsprinzip: Die Aussage die den Verklemmungszustand charakterisiert negieren.

Für alle Prozesse i aus {1,..,5} gilt: Prozess i hat seine rechte Gabel und wartet.

Negation liefert Lösungsidee:

Es gibt einen Prozess i aus {1,..,5}: Prozess i hat seine rechte Gabel nicht oder er wartet nicht.

Lösung: Einer der Prozesse, z. B. 1, nimmt erst die linke dann die rechte Gabel.

Der Verklemmungszustand „Jeder Prozess hat **eine** Gabel.“ kann nicht eintreten.

Die Bedingung (3) von SWE-151 „zyklisches Warten“ wird invalidiert.

Ein **Monitor** verwaltet die Gabeln als **Ressourcen mit Identität**.
Monitor-Operation: Dem Prozess i die i-te Gabel (**einzeln**!) geben.
Wartebedingung dafür: Die i-te Gabel ist frei.

Zusammenfassung von SWE II

allgemeine Begriffe und Konzepte	**Java Programmierung**

Graphische Bedienungsoberflächen

Komponenten-Bibliothek	AWT, Swing
hierarchische Objektstrukturen	LayoutManager
Eingabekomponenten	Button, TextField, ...
Observer-Muster	
Reaktion auf Ereignisse, Folgen	Applets

Parallele Prozesse

Grundbegriffe	Java-Threads, Prozess-Objekte
parallel, verzahnt, nebenläufig	(1) implements Runnable, (2) extends Thread
	Thread-Methoden, Threads in Applets
Monitore	Monitor-Objekte in Java
gegenseitiger Ausschluss	synchronized Mehtoden
Bedingungssynchronisation im Monitor	Methoden wait, notifyAll

Schema: Vergabe gleichartiger Ressourcen
Schema: Beschränkter Puffer
Schema: Ressourcenvergabe

Verklemmung