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)**:

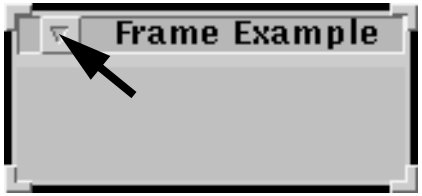
- **Bedien-Operationen lösen Ereignisse aus**, z. B. Knopf drücken, Alternative auswählen, Mauszeiger auf eine graphisches Element schieben.
- **Programmausführung reagiert auf solche Ereignisse** durch Aufruf bestimmter Methoden

Aufgaben:

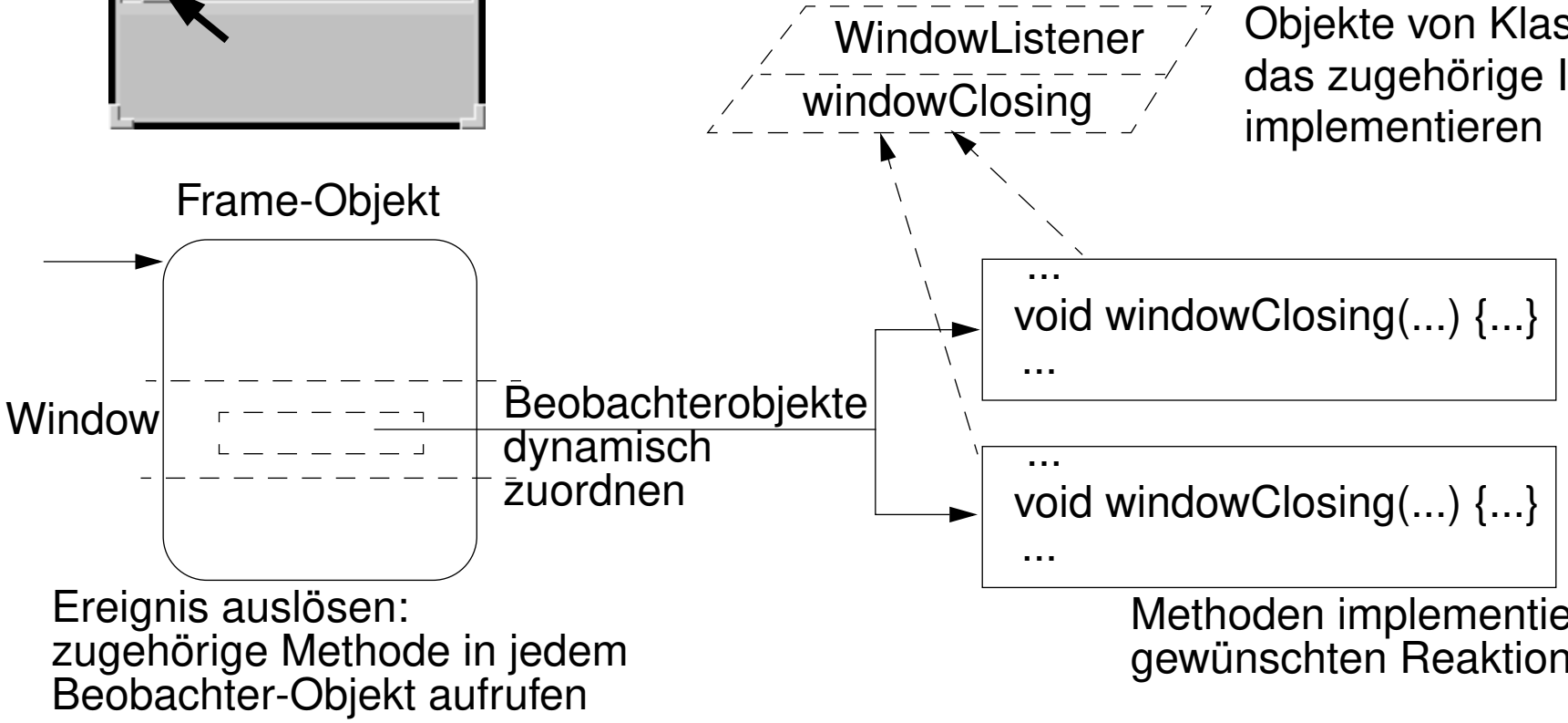
- **Folgen von Ereignissen** und Reaktionen darauf **planen und entwerfen**
  Modellierung z. B. mit endlichen Automaten oder StateCharts
- **Reaktionen auf Ereignisse** systematisch implementieren
  AWT: Listener-Konzept; Entwurfsmuster „Observer“

„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



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:

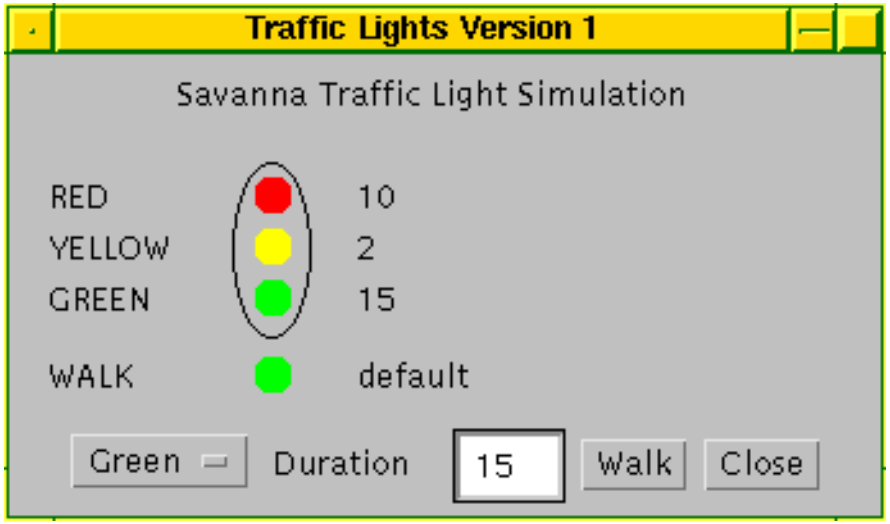
```
try { age = Integer.parseInt (str); }
catch (NumberFormatException e) { ... }
```

6. Beispiel: Ampel-Simulation

Aufgabe: Graphische Benutzungsoberfläche für eine Ampel-Simulation entwerfen

Eigenschaften:

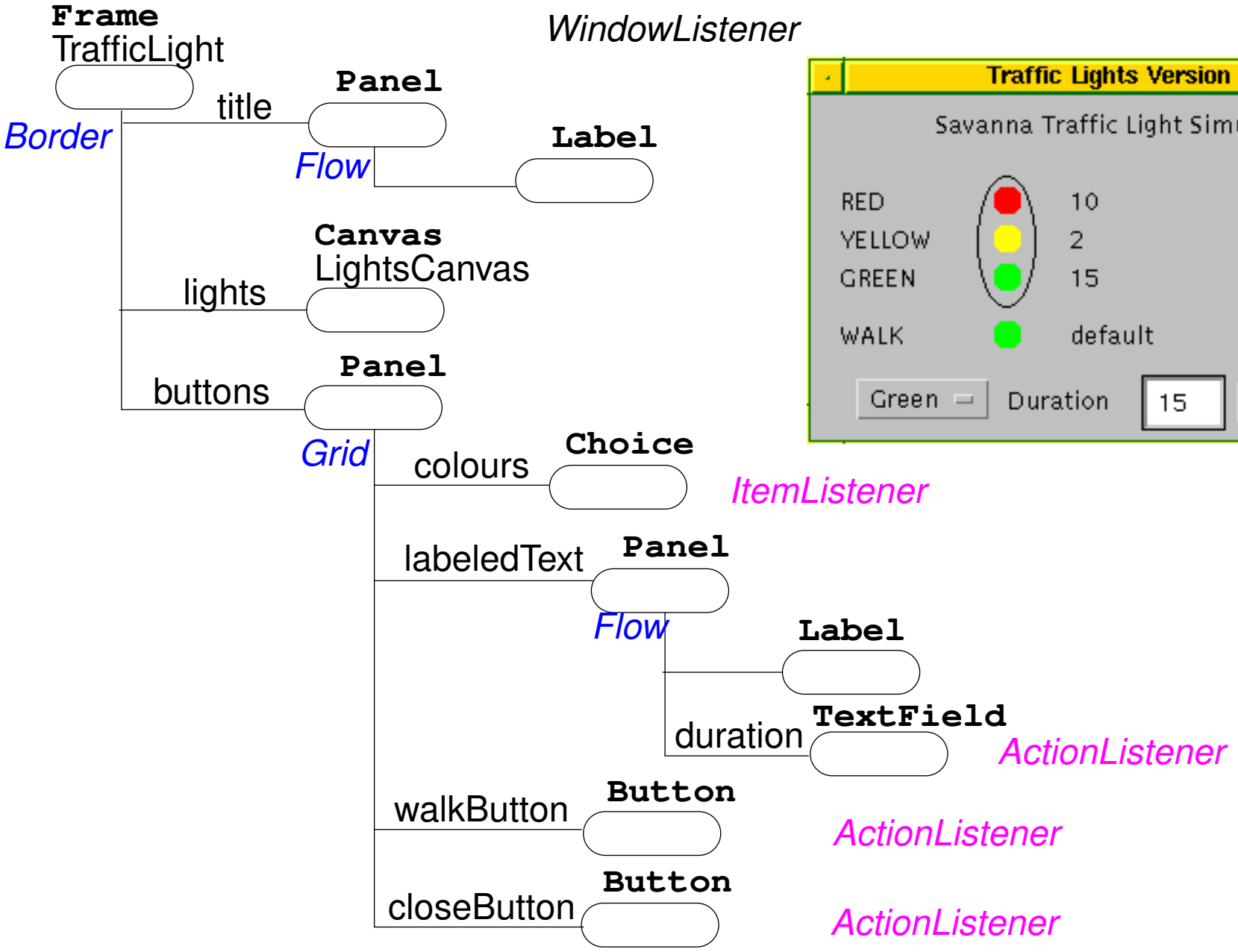
- Ampel visualisieren
  mit Knopf und Licht für Fußgänger
  (später auch animieren)
- Phasenlängen der Lichter
  einzeln einstellbar
- Einstellungen werden angezeigt

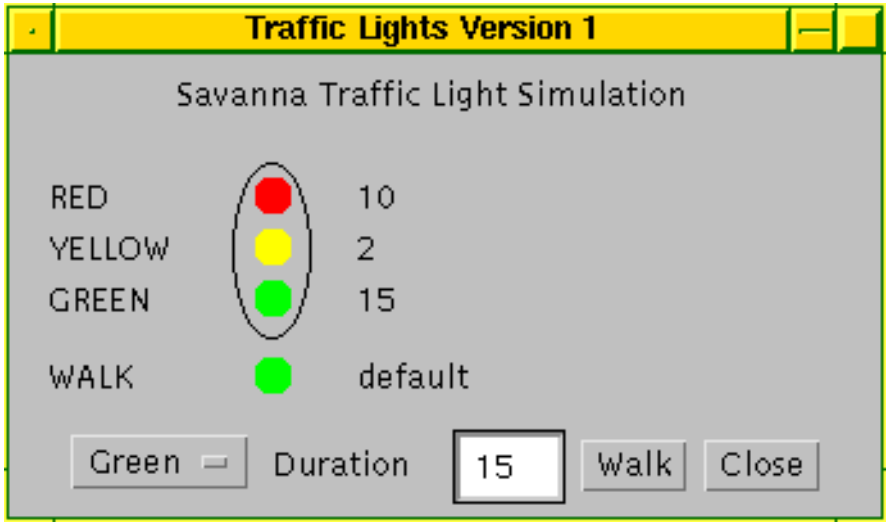


Entwicklungsschritte:

- 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:

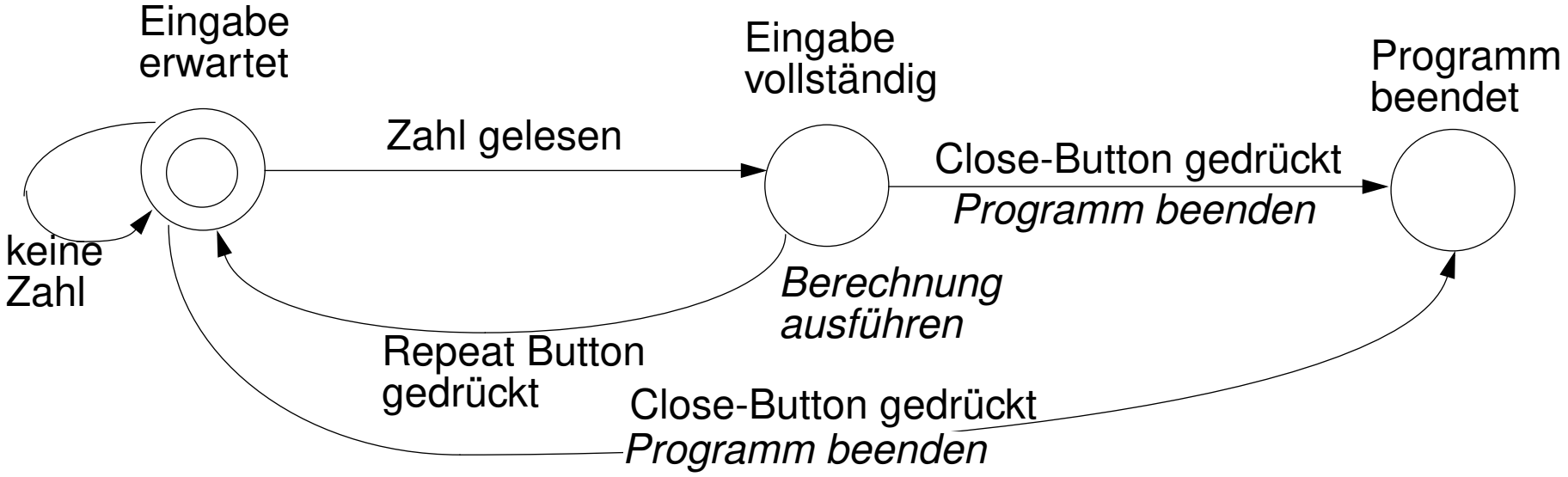
```
class LightsCanvas extends Canvas
{  private String [] msg;

   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)

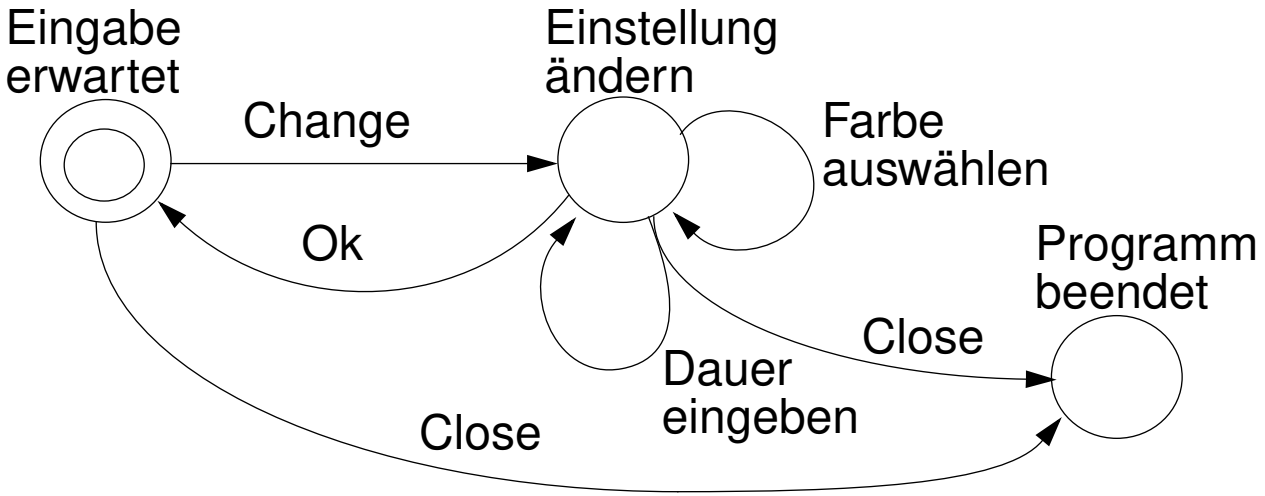
- **Zustände** unterscheiden Bediensituationen (z. B. „Eingabe erwartet“, „Eingabe vollständig“)
- **Übergänge** werden durch Ereignisse ausgelöst.
- **Aktionen** können mit Übergängen verknüpft werden; Reaktion auf ein Ereignis
  z. B. bei Eingabe einer Phasenlänge Ampel neu zeichnen, und
- **Aktionen**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

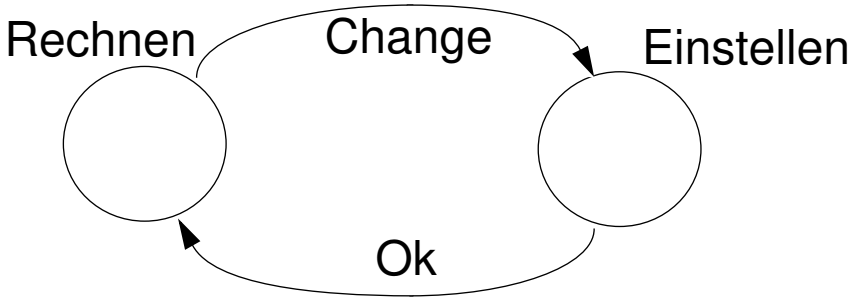


Robuste Programme dürfen auch an unzulässigen Ereignisfolgen nicht scheitern.
Verschiedene Möglichkeiten für **nicht definierte Übergänge**:

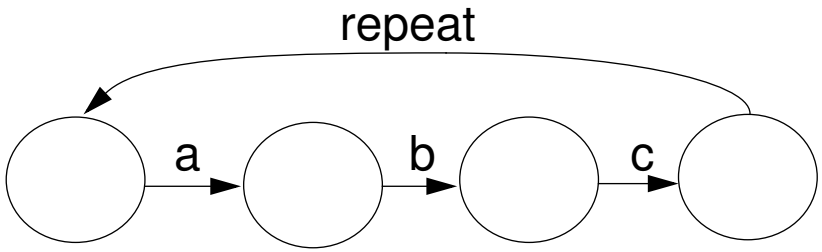
- Sie bleiben **ohne Wirkung**
- Sie bleiben ohne Wirkung und es wird eine **Erklärung** gegeben (Warnungsfenster).
- **Komponenten** werden so **deaktiviert**, dass unzulässige Ereignisse nicht ausgelöst werden können.

Muster für Ereignisfolgen

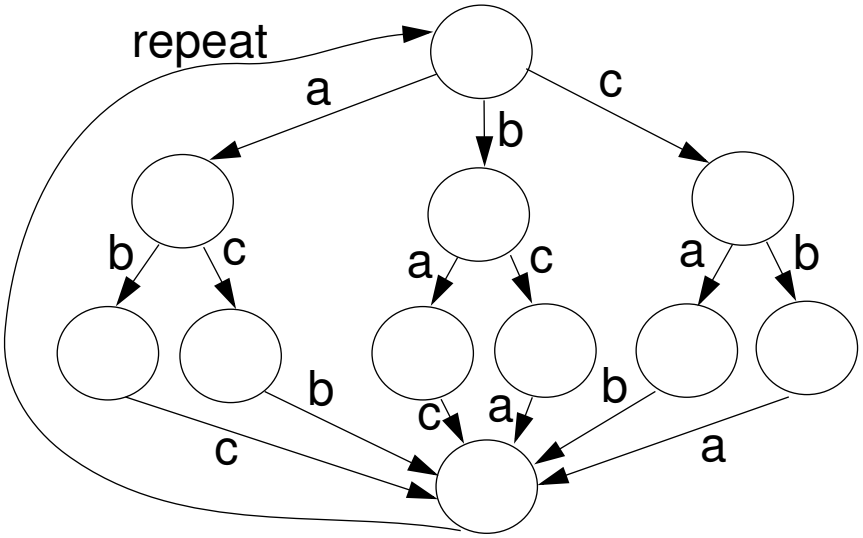
- Verschiedene **Bedienungsarten** (mode):
  Vorsicht: Nicht unnötig viele Zustände entwerfen. „Don‘t mode me in!“



- 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:

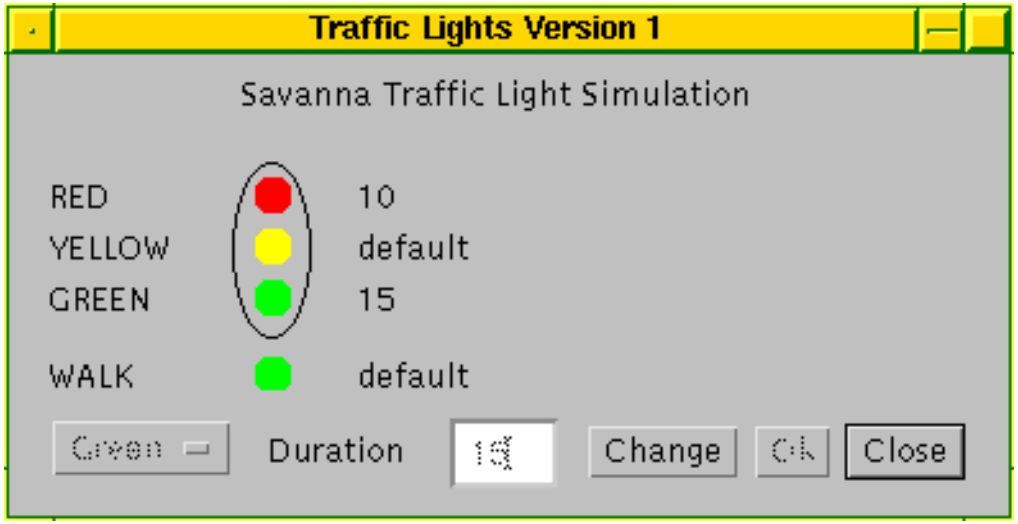
```
changeButton.addActionListener
   (new ActionListener ()
      {  public void actionPerformed(ActionEvent e)
         { toState (settingState); }});
```

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.

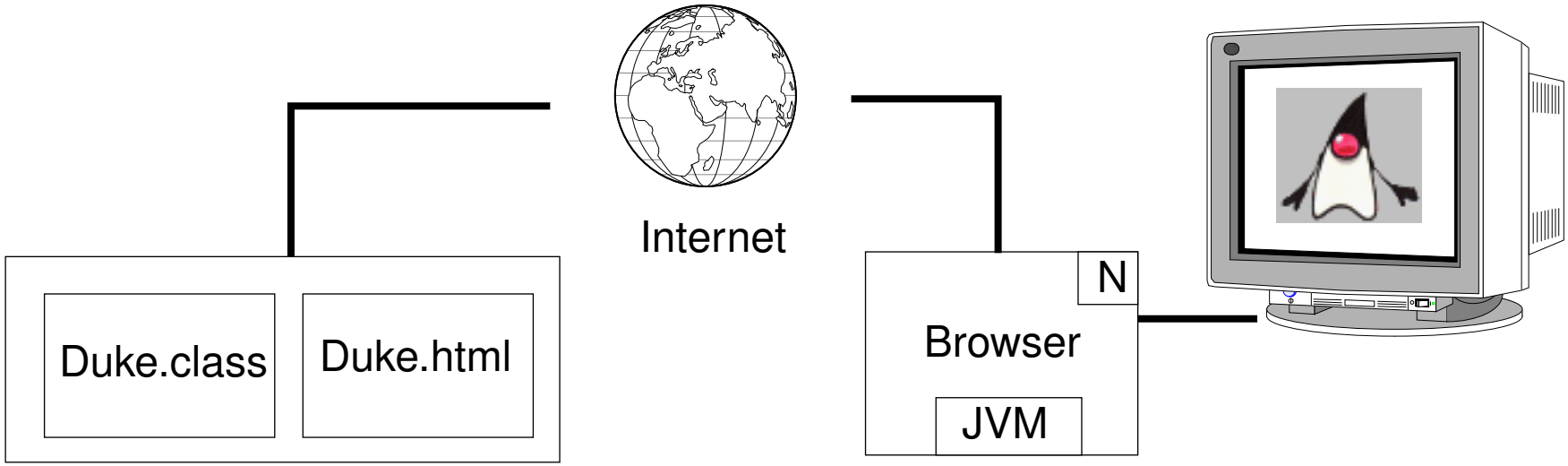


Die Komponenten zur Farbauswahl, Texteingabe und der Ok-Knopf sind im gezeigten Zustand deaktiviert.

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.

© 2004 bei Prof. Dr. Uwe Kastens

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 ... }
   ...
}
```

© 2004 bei Prof. Dr. Uwe Kastens

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`**)

© 2004 bei Prof. Dr. Uwe Kastens

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 **
** und Klammern **
** ... **
** 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.

© 2004 bei Prof. Dr. Uwe Kastens

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.

© 2004 bei Prof. Dr. Uwe Kastens

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"));
```

© 2004 bei Prof. Dr. Uwe Kastens

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.

© 2004 bei Prof. Dr. Uwe Kastens