B1. Ein-/Ausgabebetonte Programmierung

Ereignisgesteuerter Programmablauf

- **Definition** Ein *Ereignis* ist ein Vorgang in der Umwelt des Softwaresystems von vernachlässigbarer Dauer, der für das System von Bedeutung ist.

  Eine wichtige Gruppe von Ereignissen sind Benutzerinteraktionen.

- **Beispiele** für Benutzerinteraktions-Ereignisse:
  - Drücken eines Knopfs
  - Auswahl eines Menüpunkts
  - Verändern von Text
  - Zeigen auf ein Gebiet
  - Schließen eines Fensters
  - Verbergen eines Fensters
  - Drücken einer Taste
  - Mausklick

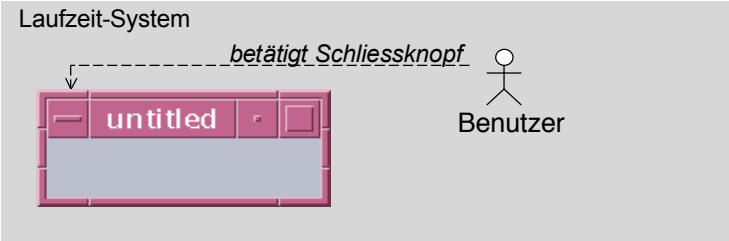
Ereignis-Klassen

- Klassen von Ereignissen in (Java-)Benutzungsoberflächen:
  - WindowEvent
  - ActionEvent
  - MouseEvent
  - KeyEvent, ...
- Bezogen auf Klassen für Oberflächenelemente:
  - Window
  - JFrame
  - JButton
  - JTextField, ...
- Zuordnung (Beispiele):
  - JFrame erzeugt WindowEvent
    - » z.B. Betätigung des Schliessknopfes
  - JButton erzeugt ActionEvent
    - » bei Betätigung des Knopfes

Hauptprogramm für Fensteranzeige

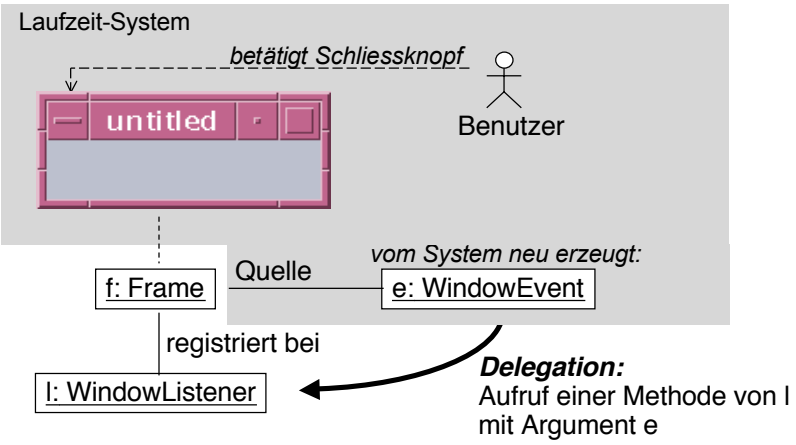
```
import java.awt.*;
Import javax.swing.*;

class ExampleFrame extends JFrame {

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

class GUI1 {
  public static void main (String[] argv) {
    ExampleFrame f = new ExampleFrame();
  }
}
```

Ereignis-Delegation (1)



- Reaktion auf ein Ereignis durch Programm:
  - Ereignis wird vom Laufzeitsystem erkannt
- Programm soll von technischen Details entkoppelt werden
  - Beobachter-Prinzip:
    - » Programmteile registrieren sich für bestimmte Ereignisse
    - » Laufzeitsystem sorgt für Aufruf an passender Stelle
- Objekte, die Ereignisse beobachten, heißen bei Java *Listener*.

Ereignis-Delegation (2)



Registrierung für Listener

- In javax.swing.JFrame (ererbt von java.awt.Window):

```
public class JFrame ... {
  public void addWindowListener
     (WindowListener l)
}
```

- java.awt.event.WindowListener ist eine Schnittstelle:

```
public interface WindowListener {
  ... Methoden zur Ereignisbehandlung
}
```

- Vergleich mit Observer-Muster:
  - Frame bietet einen "Observable"-Mechanismus
  - Window-Listener ist eine "Observer"-Schnittstelle

java.awt.event.WindowListener

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

java.util.EventListener:

Basisinterface für alle "Listener" (keine Operationen)

java.awt.event.WindowEvent

```
public class WindowEvent extends AWTEvent {
  ...
  // Konstruktor, wird vom System aufgerufen
  public WindowEvent (Window source, int id);

  // Abfragemöglichkeiten
  public Window getWindow();
  ...
}
```


java.awt.event.ActionEvent, ActionListener

```
public class ActionEvent extends AWTEvent {
  ...
  // Konstruktor, wird vom System aufgerufen
  public ActionEvent
   (Window source, int id, String command);
  // Abfragemöglichkeiten
  public Object getSource ();
  public String getActionCommand();
  ...
}

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

WindowListener für Ereignis "Schließen"

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

class WindowCloser implements WindowListener {

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

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


Hauptprogramm für schließbares Fenster

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

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

class ExampleFrame extends JFrame {

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

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

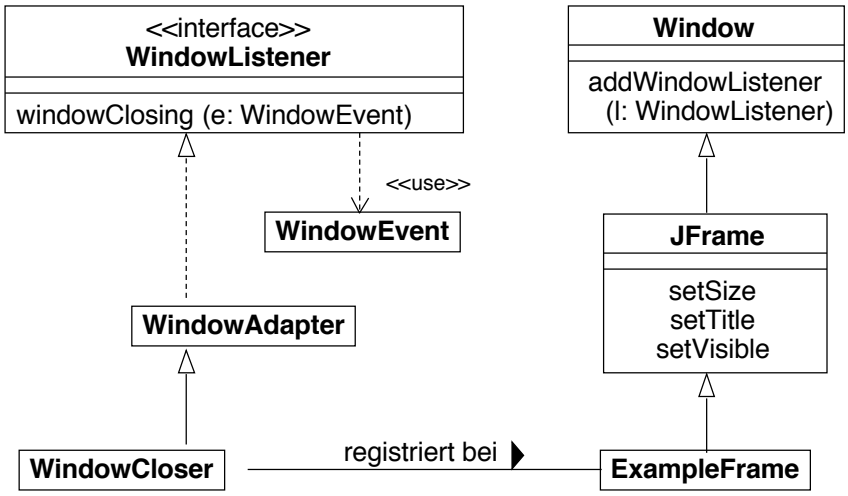
java.awt.event.WindowAdapter

```
public abstract class WindowAdapter
    implements WindowListener {

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

}
```


Vereinfachung 1: WindowAdapter benutzen

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

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

class ExampleFrame extends JFrame {

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

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

Schließbares Fenster: Klassenstruktur



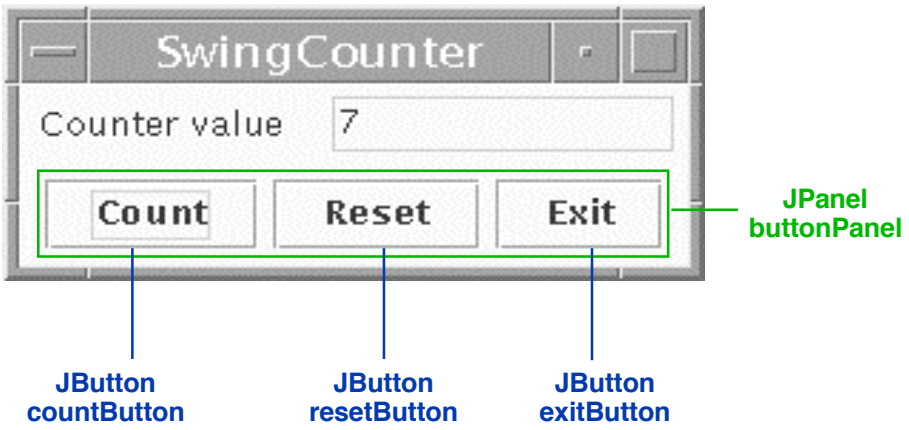
Vereinfachung 2: Innere Klasse benutzen

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

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

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

class GUI4 {
public static void main (String[] argv) {
   ExampleFrame f = new ExampleFrame();}}
```

Vereinfachung 3: Anonyme Klasse benutzen

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

class ExampleFrame extends JFrame {

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

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


Zähler-Beispiel: Entwurf der Bedienelemente



Die Sicht *(View):* Bedienelemente

```
class CounterFrame extends JFrame {
  JPanel valuePanel = new JPanel();
  JTextField valueDisplay = new JTextField(10);
  JPanel buttonPanel = new JPanel();
  JButton countButton = new JButton("Count");
  JButton resetButton = new JButton("Reset");
  JButton exitButton = new JButton("Exit");

  public CounterFrame (Counter c) {
    setTitle("SwingCounter");
    valuePanel.add(new JLabel("Counter value"));
    valuePanel.add(valueDisplay);
    valueDisplay.setEditable(false);
    getContentPane().add(valuePanel);
    buttonPanel.add(countButton);
    buttonPanel.add(resetButton);
    buttonPanel.add(exitButton);
    getContentPane().add(buttonPanel);
    pack();
    setVisible(true);
  }
}
```


Layout-Manager

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

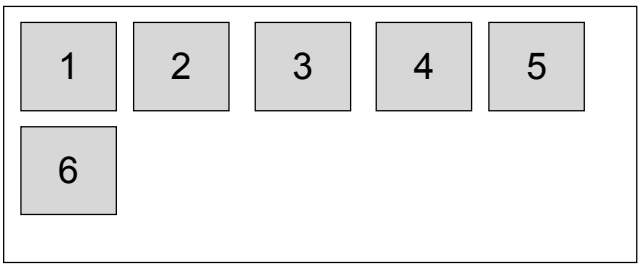
- In awt.Component:

  ```
  public void add (Component comp, Object constraints);
  ```

  erlaubt es, zusätzliche Information (z.B. Orientierung, Zeile/Spalte) an den Layout-Manager zu übergeben

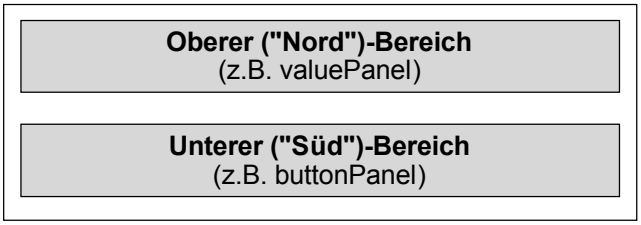
Flow-Layout

- Grundprinzip:
  - Anordnung analog Textfluß:
    von links nach rechts und von oben nach unten
- Default für JPanels
  - z.B. in valuePanel und buttonPanel
    für Hinzufügen von Labels, Buttons etc.
- Parameter bei Konstruktor: Orientierung auf Zeile, Abstände
- Constraints bei **`add`**: keine



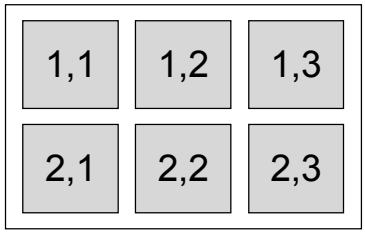
Border-Layout

- Grundprinzip:
  - Orientierung nach den Seiten (N, S, W, O)
    bzw. Mitte (center)
- Default für Window, JFrame
  - z.B. in CounterFrame
    für Hinzufügen von valuePanel, buttonPanel
- Parameter bei Konstruktor: Keine
- Constraints bei **`add:`**
  - **`BorderLayout.NORTH, SOUTH, WEST, EAST, CENTER`**



Grid-Layout

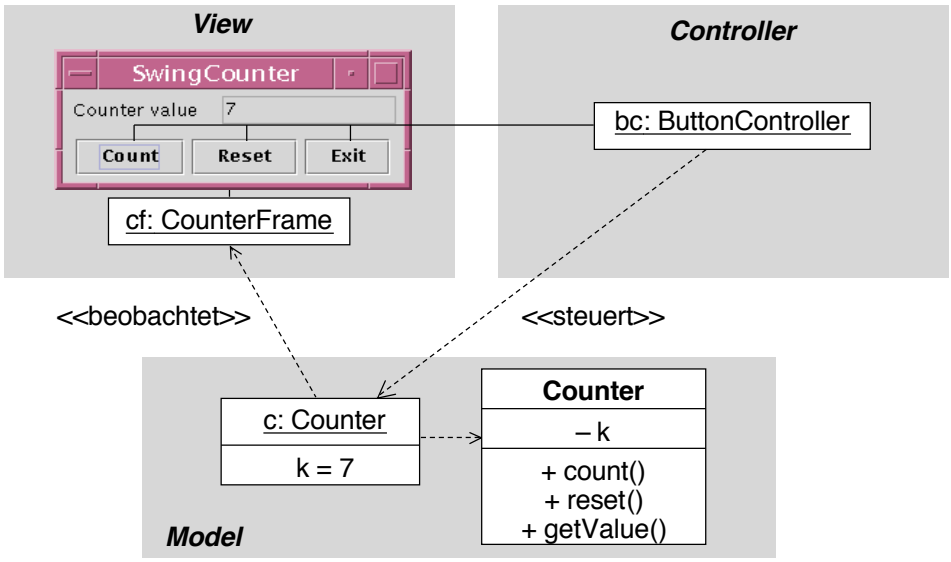
- Grundprinzip:
  - Anordnung nach Zeilen und Spalten
- Parameter bei Konstruktor:
  - Abstände, Anzahl Zeilen, Anzahl Spalten
- Constraints bei **`add:`**
  - Zeilen- und Spaltenindex als int-Zahlen



Die Sicht *(View):* Alle sichtbaren Elemente

```
class CounterFrame extends JFrame {
  JPanel valuePanel = new JPanel();
  JTextField valueDisplay = new JTextField(10);
  JPanel buttonPanel = new JPanel();
  JButton countButton = new JButton("Count");
  JButton resetButton = new JButton("Reset");
  JButton exitButton = new JButton("Exit");

  public CounterFrame (Counter c) {
    setTitle("SwingCounter");
    valuePanel.add(new JLabel("Counter value"));
    valuePanel.add(valueDisplay);
    valueDisplay.setEditable(false);
    getContentPane().add(valuePanel, BorderLayout.NORTH);
    buttonPanel.add(countButton);
    buttonPanel.add(resetButton);
    buttonPanel.add(exitButton);
    getContentPane().add(buttonPanel, BorderLayout.SOUTH);
    pack();
    setVisible(true);
  }
}
```

Model-View-Controller-Architektur



Zähler-Beispiel: Anbindung Model/View

```
class CounterFrame extends JFrame
   implements Observer {
  ...
  JTextField valueDisplay = new JTextField(10);
  ...

  public CounterFrame (Counter c) {
    ...
    valuePanel.add(valueDisplay);
    valueDisplay.setEditable(false);
    valueDisplay.setText(String.valueOf(c.getValue()));
    ...
    c.addObserver(this);
    pack();
    setVisible(true);
  }

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

Grundidee der Implementierung von Observable

- Der Programmierer muß den hier skizzierten Code nicht kennen, sondern nur indirekt anwenden!

```
public class Observable {

  private Collection observed;
  private boolean changed = false;
  ...

  public void addObserver (Observer o) { observed.add(o); }

  public void setChanged() { changed = true; }

  public void notifyObservers (Object arg) {
      Iterator it = observed.iterator();
      if (!changed) return;
      while (it.hasNext()) {
            (it.next()).update(this, arg);
      }
  }
}
```


java.awt.event.ActionEvent, ActionListener

```
public class ActionEvent extends AWTEvent {
  ...
  // Konstruktor wird vom System aufgerufen

  public Object getSource ()
  public String getActionCommand()
  ...
}

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

Wieviele Controller?

- Möglichkeit 1: Ein Controller für mehrere Buttons (sh.nächste Folie)
  - Speicherplatzersparnis
  - Aber: Wie unterscheiden wir, woher die Ereignisse kommen?
  - Z.B. über **`getSource()`** und Abfrage auf Identität mit Button-Objekt
  - Z.B. über **`getActionCommand()`** und Abfrage auf Kommando-String
    - » Default: Kommando-String aus Button-Beschriftung
    - » Kann gesetzt werden mit **`setActionCommand()`**
    - » Standard-Kommando-String gleich Button-Label - nicht ungefährlich…
- Möglichkeit 2:
  - Direkte Angabe von Eventhandlern
    - » am knappsten über anonyme innere Klassen
  - Viele Controller-Objekte
  - Siehe weiter hinten

Die Steuerung *(Controller)*

```
class ButtonController implements ActionListener {

  Counter myCounter;

  public void actionPerformed (ActionEvent event) {
    String cmd = event.getActionCommand();
    if (cmd.equals("Count"))
      myCounter.count();
    if (cmd.equals("Reset"))
      myCounter.reset();
    if (cmd.equals("Exit"))
      System.exit(0);
  }

  public ButtonController (Counter c) {
    myCounter = c;
  }
}
```

Zähler-Beispiel: Anbindung des Controllers

```
class CounterFrame extends JFrame {
  ...
  JPanel buttonPanel = new JPanel();
  JButton countButton = new JButton("Count");
  JButton resetButton = new JButton("Reset");
  JButton exitButton = new JButton("Exit");
  public CounterFrame (Counter c) {
    ...
    ButtonController bc = new ButtonController(c);
    countButton.setActionCommand("Count");
    countButton.addActionListener(bc);
    buttonPanel.add(countButton);
    resetButton.setActionCommand("Reset");
    resetButton.addActionListener(bc);
    buttonPanel.add(resetButton);
    exitButton.setActionCommand("Exit");
    exitButton.addActionListener(bc);
    buttonPanel.add(exitButton);
    ...
  }
}
```


Alles zusammen: CounterFrame (1)

```
class CounterFrame extends JFrame implements Observer {
  JPanel valuePanel = new JPanel();
  JTextField valueDisplay = new JTextField(10);
  JPanel buttonPanel = new JPanel();
  JButton countButton = new JButton("Count");
  JButton resetButton = new JButton("Reset");
  JButton exitButton = new JButton("Exit");
  public CounterFrame (Counter c) {
      setTitle("SwingCounter");
      valuePanel.add(new JLabel("Counter value"));
      valuePanel.add(valueDisplay);
      valueDisplay.setEditable(false);
      valueDisplay.setText(String.valueOf(c.getValue()));
      getContentPane().add(valuePanel,BorderLayout.NORTH);
      ButtonController bc = new ButtonController(c);
      countButton.setActionCommand("Count");
      countButton.addActionListener(bc);
      buttonPanel.add(countButton);
      resetButton.setActionCommand("Reset");
      resetButton.addActionListener(bc);
      buttonPanel.add(resetButton);
      exitButton.setActionCommand("Exit");
      exitButton.addActionListener(bc);
      buttonPanel.add(exitButton);
      getContentPane().add(buttonPanel,BorderLayout.SOUTH);
```

Alles zusammen: CounterFrame (2)

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

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

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


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

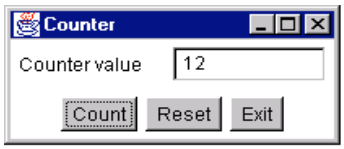

Controller durch anonyme Klassen

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

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

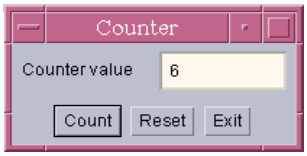
Controller und View bilden eine Einheit: In der Praxis weit verbreitet.

"Look-and-Feel"

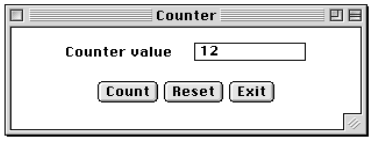
- Jede Plattform hat ihre speziellen Regeln für z.B.:
  - Gestaltung der Elemente von "Frames" (Titelbalken etc.)
  - Standard-Bedienelemente zum Bewegen, Schließen, Vergrößern, von "Frames"
- Dasselbe Java-Programm mit verschiedenen "Look and Feels":



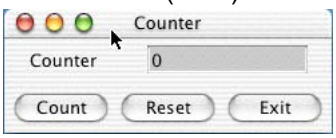
Windows



Solaris (CDE)



Macintosh (Classic)



Macintosh (MacOS X)

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