Swing

- Vorbemerkungen
- Einfaches Beispiel
- Container, Komponenten, Ereignisbehandlung
- Erstellung eines einfachen Formulars
- Graphische Animation
- Komplexe Anwendung

Unterschiede AWT und Swing

- In der AWT erfolgen alle graphischen Darstellungen über die Widgets des Windowsytems. Die AWT stellt praktisch (nur) ein neutrales Frontend zu diesem System dar. (Die Komponenten sind „schwergewichtig“)

- Swing bietet „leichtgewichtige“ Komponenten an, die in Java realisiert sind und nur auf einer relativ elementaren Ebene an das Fenstersystem gekoppelt sind.

- Ein Vorteil von Swing ist das einstellbare (plugable) look & feel. Ein anderer Vorteil ist die größere Freiheit in der Gestaltung der Komponenten (z.B. runde Schaltflächen).

- Swinganwendungen benutzen Teile der AWT (Layoutmanager, Eventhandling). Sie sollten aber keine AWT-Komponenten benutzen!

- Im Unterschied zur AWT sind die Swingkomponenten nicht threadsicher. Konsequenz:: Änderungen an der Darstellung sollten nur im Eventthread erfolgen!

- Swing bietet auch relativ komplexe Komponenten an. Hier wird ein einheitliches Schema nach dem Muster MVC angeboten.

Vorbemerkungen

- Grundsätzlich kann man mit Java (wie entsprechend auch mit anderen Programmiersprachen) unterschiedliche GUI-Bibliotheken verwenden, die nach Java portiert wurden. Weit verbreitet ist inzwischen die SWT.

- Die ursprüngliche GUI-Bibliothekt findet sich in java.awt.* Paketen. AWT = abstract windowing toolkit basiert auf dem „native“ Look & Feel des jeweiligen Hostsystems.

- Swing ist eine Bibliothek aus „leichtgewichtigen“ in Java implementierten Komponenten, die über einstellbares Look & Feel realisiert werden können.

- In Swing werden mehrere Entwurfsmuster explizit verwendet.

- Sowohl für AWT als auch für Swing gibt es weitere spezialisierte Pakete (z.B. java2d und java3d).

- Hier soll nur ein grober Überblick gegeben werden.

Einfachstes Swing Beispiel

```
 1 import javax.swing.*;
 2 public class HelloWorldSwing {
 3    public static void main(String[] args) {
 4        JFrame frame = new JFrame("HelloWorldSwing");
 5        final JLabel label = new JLabel("Hello World");
 6        frame.getContentPane().add(label);
 7        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
 8        frame.pack();
 9        frame.setVisible(true);
10    }
11 }
```

Zeile 1: importiert das Swing-Paket **`javax.swing`**.
Zeile 4: ein Frame ist ein **Top-Level Window**.
Zeile 4: `getContentPane()` gibt den Behälter (**container**) des Frame zurück.
Zeile 5: ein `JLabel` ist eine primitive Komponente (**component**).
Zeile 7: hier wird die Aktion des **Endeknopfes** definiert.
Zeile 8: hiermit wird der **Layoutmanager** des Behälters zum automatischen packen der Komponenten aktiviert.
Zeile 9: erst hierdurch wird die graphische **Darstellung** angestoßen und der Eventthread gestartet.

Top-Level-Container

Top-Level-Container stellen ein vom Windowsystem verwaltetes Fenster dar. Sie enthalten eine Zeichenfläche (Behälter für Komponenten) und verarbeiten die Ereignisse der Benutzerinteraktion.

Es gibt 3 verschiedene Top-Level-Container: `JFrame, JDialog und JApplet`.

Auf Wunsch kann ein besonderes Look & Feel eingstellt werden.

Randbemerkungen:

- Die Top-Level-Container sind zugleich Komponenten und Behälter. Als Komponenten verfügen Sie über das allgemeine Verhalten (z.B. Model-View-Control).
- Zu vielen AWT-Klassen gibt es entsprechende Swing-Klassen (zu erkennen am J)
- Man muss bei Swing manchmal die AWT verwenden. Man sollte aber dabei keine AWT-Komponenten einsetzen.
- Neben den Top-Level-Containern gibt es als wichtigsten internen Container die Klasse `JPanel`.

Top-Level-Container

Top-Level-Container stellen ein vom Windowsystem verwaltetes Fenster dar. Sie enthalten eine Zeichenfläche (Behälter für Komponenten) und verarbeiten die Ereignisse der Benutzerinteraktion.

Es gibt 3 verschiedene Top-Level-Container: `JFrame, JDialog und JApplet`.

Auf Wunsch kann ein besonderes Look & Feel eingstellt werden.

Randbemerkungen:

- Die Top-Level-Container sind zugleich Komponenten und Behälter. Als Komponenten verfügen Sie über das allgemeine Verhalten (z.B. Model-View-Control).
- Zu vielen AWT-Klassen gibt es entsprechende Swing-Klassen (zu erkennen am J)
- Man muss bei Swing manchmal die AWT verwenden. Man sollte aber dabei keine AWT-Komponenten einsetzen.
- Neben den Top-Level-Containern gibt es als wichtigsten internen Container die Klasse `JPanel`.

Swing Komponenten.

- Komponenten sind in einem **Container** enthalten.
- Komponenten haben ein bestimmte graphisches **Aussehen**.
- Komponenten wie `JButton` oder `JLabel` enthalten einen **Text**.
- Komponenten können aber auch **Graphiken** enthalten.
- Viele Komponenten sind selbst wieder Container.
- Komponenten sind in der Regel **undurchsichtig** (opaque). Dies bedeutet, dass das gesamte Flächenrechtekt „bemalt“ ist.
- Swing-Komponenten können aber auch (teilweise) **durchsichtig** sein.
- Mit einigen Komponenten sind **Aktionsobjekte** verknüpft, die auf Ergeignisse (event) reagieren.
- Komponenten für Texteingabe sind nur dann aktiv, wenn Sie den **Fokus** besitzen (per Maus oder durch Einstellung des Programms)
- Neben den einfachen Komponenten (`JButton`) gibt es auch sehr **komplexe Komponenten** (`JTable`, `JEditorPane`).

Ereignisbehandlung

Swing kennt unterschiedliche Arten von Ereignissen. Neben den einfachen GUI-Ereignissen gibt es nämlich weitere Ereignisse innerhalb des MVC-Modells.

GUI-Ereignisse werden wie in der AWT behandelt:

ActionListener	der Benutzer drückt return oder betätigt einen Button
WindowListener	der Benutzer schließt ein Fenster
MouseListener	der Benutzer drückt einen Mausknopf während der Mauszeiger über einer Komponente ist
MouseMotionListener	der Benutzer bewegt die Maus über eine Komponente
ComponentListener	die Komponente wird sichtbar
FocusListener	
ListSelectionListener	Tabellen- oder Listenauswahl ändert sich

Grundarten von Swing-Anwendungen.

Im folgenden sollen unterschiedliche Anwendungsarten beispielhaft vorgestellt werden:

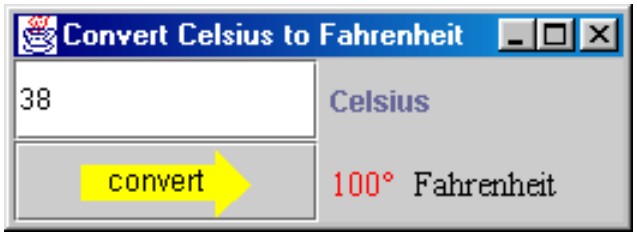
Einfache (in der Regel **formularbasierte Anwendungen)**, bei denen vorgefertigte Widgets benutzt werden. Die Widgets erhalten sowohl das jeweilige Modell als auch die UI (User Interface = View+Controll).

Graphische Anwendungen (als Beispiel eine graphische Animation). Hier müssen Modell und Sicht programmiert werden (dabei ist man aber auch weniger an spezielle Muster gebunden).

Die Verwendung **komplexer Widgets** (`JTree`, `JEditorPane`, `JTable`, `JList`). Bei diesen Anwendungen wird in der Regel eine vorgefertigte UI verwendet. D.h. es ist nur noch ein passendes Modell zu erzeugen.

Einfache GUI-Klasse

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

public class CelsiusConverter implements ActionListener {
    // Klasse dient gleichzeitig als ActionListener

    // Fenster:
    JFrame converterFrame;

    // Teilbereich
    JPanel converterPanel;

    // Eingabefeld:
    JTextField tempCelsius;

    // Beschriftung und Ausgabe:
    JLabel celsiusLabel, fahrenheitLabel;

    // Aktionsknopf:
    JButton convertTemp;

    // Methoden:


}
```



Konstruktor

```
public CelsiusConverter() {
    // Fenster und Titel
    converterFrame = new JFrame("Convert Celsius to Fahrenheit");

    // Fenstergroesse in Pixel
    converterFrame.setSize(40, 40);

    // JPanel ist ein nackter Behälter
    converterPanel = new JPanel();

    // Der Layoutmanager wird definiert (2x2 Gitter)
    converterPanel.setLayout(new GridLayout(2, 2));

    // Komponenten einfuegen
    addWidgets();

    // Panel im Fenster plazieren usw.
    converterFrame.getContentPane().add(converterPanel, BorderLayout.CENTER);
    converterFrame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    converterFrame.pack();
    converterFrame.setVisible(true);
}
```

Erzeugung der Komponenten

```
private void addWidgets() {

    // Bild (fuer Button) laden
    ImageIcon icon =
        new ImageIcon("images/convert.gif", "Convert temperature");

    // zweistelliges Eingabefeld
    tempCelsius = new JTextField(2);
    // Beschriftung des Eingabefeldes
    celsiusLabel = new JLabel("Celsius", SwingConstants.LEFT);
    // Aktionsknopf (mit Bild)
    convertTemp = new JButton(icon);
    // Ausgabefeld
    fahrenheitLabel = new JLabel("Fahrenheit", SwingConstants.LEFT);

    // Rahmen um Labels
    celsiusLabel.setBorder(BorderFactory.createEmptyBorder(5,5,5,5));
    fahrenheitLabel.setBorder(BorderFactory.createEmptyBorder(5,5,5,5));

    // Ereignisbehandlung und plazieren der Komponenten
    convertTemp.addActionListener(this);
    converterPanel.add(tempCelsius);
    converterPanel.add(celsiusLabel);
    converterPanel.add(convertTemp);
    converterPanel.add(fahrenheitLabel);
}
```

Aktionsbehandlung

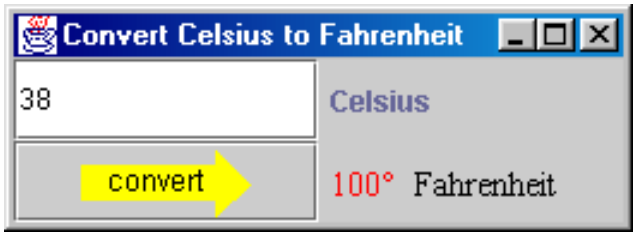
```
public void actionPerformed(ActionEvent event) {
    // Eingabe, Zahlenumwandlung und Umrechnung
    int tempFahr = (int)((Double.parseDouble(tempCelsius.getText()))
                   * 1.8 + 32);

    // Darstellung vom Ergebnis im fahrenheitLabel mit unterschiedlicher
    // Farbe (durch html)

    if (tempFahr <= 32) {
        fahrenheitLabel.setText("<html><Font Color=blue>" + tempFahr +
            "&#176 </Font><Font Color=black> Fahrenheit</font></html>");
    } else if (tempFahr <= 80) {
         fahrenheitLabel.setText("<html><Font Color=green>" + tempFahr +
             "&#176 </Font><Font Color=black> Fahrenheit </Font></html>");
    } else {
        fahrenheitLabel.setText("<html><Font Color=red>" + tempFahr +
            "&#176 </Font><Font Color=black> Fahrenheit</Font></html>");
    }
}
```

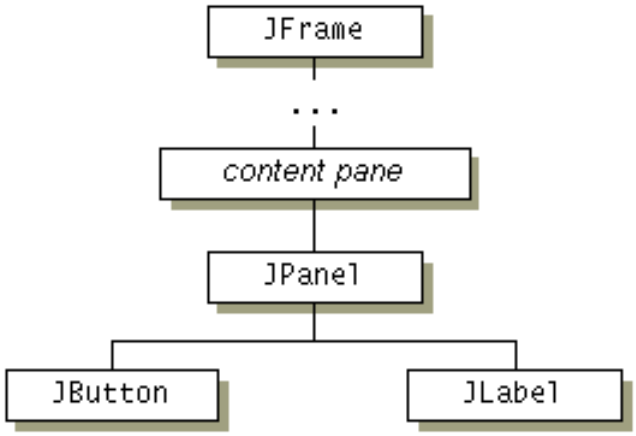
In Swing sollen Komponenten nur (wie hier) in dem Ereignisthread verändert werden!

Rule: Once a Swing component has been realized, all code that might affect or depend on the state of that component should be executed in the event-dispatching thread.

Main

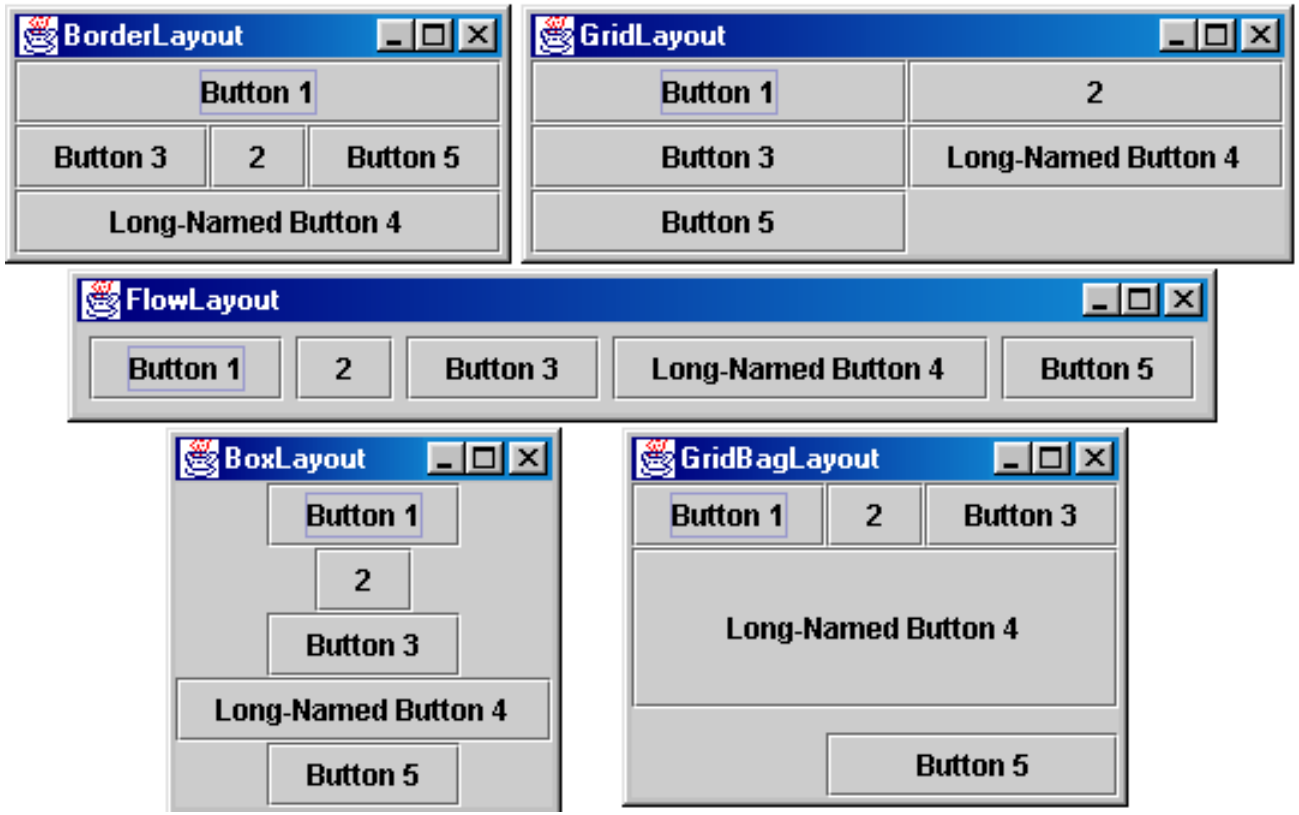
```
public static void main(String[] args) throws Exception {

    // setze Look & Feel zum Platform-Default
    UIManager.setLookAndFeel(
        UIManager.getCrossPlatformLookAndFeelClassName());

    // Start der Anwendung
    CelsiusConverter converter = new CelsiusConverter();
}
```





Es gibt unterschiedliche Layoutmanager

Layoutmanager übernehmen die Plazierung von Komponenten auf der Basis einfacher Regeln.



Anmerkung zu Bildschirmmasken

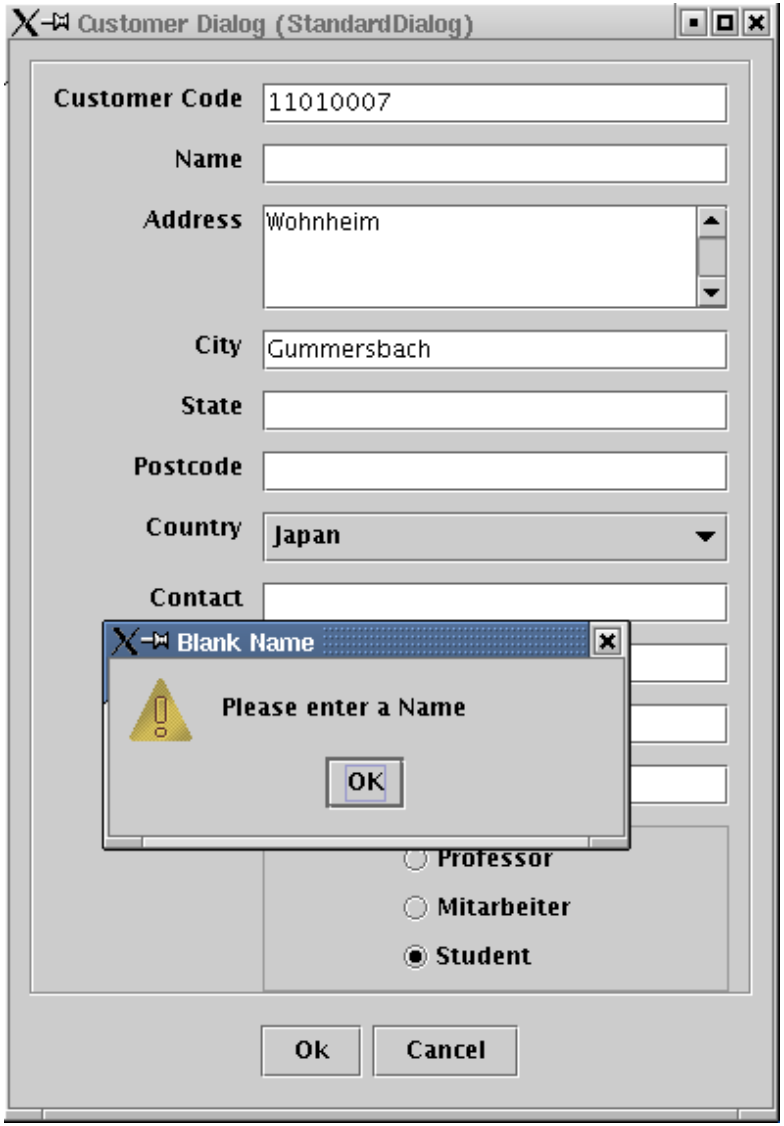
Graphische Entwicklungsumgebungen erlauben oft die interaktive Erzeugung des Layouts der Oberfläche. Dies ist sinnvoll für einfache bis mittlere Anwendungen, bei denen die Oberfläche sich langfristig nicht mehr ändert.

Bei größeren Projekten ist damit jedoch ein unverhältnismäßig großer Aufwand verbunden.

Andere Lösungen bauen auf standardisierten Komponenten und Strukturen auf. Anstelle eines visuellen Layouts wird hierbei oft ein logisch strukturiertes hierarchisches Darstellungsschema eingesetzt.

Anmerkung zu Swing: auf der Swing-Ebene ergibt sich aus der Schachtelung von Behältern (insbesondere `JPanel`) mit evtl. jeweils angepassten Layoutmanagern eine hierarchische Struktur.

Bildschirmdialog mit Fehlerdialog



Typisch für viele formularbasierte Anwendungen ist:

- eine an Reihung unterschiedlicher **Eingabeelemente**
- eine Standardisierung im **Aussehen** und **Verhalten** der verwendeten Komponenten
- eine Standardisierung der **Interaktion** der Komponenten mit der Anwendung

Eine typische Lösung ist:

- **modaler Dialog** für ein Bündel von Eingaben
- **Prüfung der Eingabe** auf Korrektheit
- nach Beendigung der Eingabe **Übernahme aller Werte** in das Anwendungsmodel.

Die Standardisierung der Interaktion erlaubt (und erfordert) eine entsprechende Softwareschicht:

- **vereinfachte Aufrufe**
- **Einschränkung** der Möglichkeiten
- wo es geht und sinnvoll ist: **Orientierung am Stil von Swing**
- explizite **Modelle** können sinnvoll sein, sind oft aber nicht nötig

Swing-Komponenten implementieren **allgemeine Entwurfsmuster**

Anwendungskomponenten sollen spezialisiert sein durch **vereinfachte Muster**

Graphische Darstellung und Animation

Der Neuaufbau der graphischen Ausgabe einer Komponente geschieht ausschließlich über die Methode **`paintComponent().`**

Das Zeichnen kann man als Modifikation eines Framebuffers auffasen. Swing verwendet als Default **doppeltes Puffern**, d.h. es wird zunächst nur im Hauptspeicher gezeichnet und anschließend die fertige Graphik in den aktiven Zeichenbereich übertragen.

Da die Methode `paintComponent()` die graphische Sicht realisiert, ist es wichtig, dass sie sich an den **Swingkonventionen** orientiert.

`paintComponent()` wird auch von Swing **automatisch aufgerufen**, wenn z.B. ein Teil der Zeichenfläche verdeckt war und deshalb neu erstellt werden muss. Direkt darf man es nie aufrufen, sondern nur über **`repaint()`** den Aufruf indirekt veranlassen.

Es ist so gewährleistet, dass `paintComponent()` immer **im Event-Thread** ausgeführt wird.

Per Voreinstellung ist `JPanel` **opaque**. Dies bedeutet, dass die gesamte Fläche neu gezeichnet werden muss.

Graphics = Schnittstelle zum Framebuffer

Ausgabe von eigenen Graphiken geschieht dadurch, dass man zunächst eine eigene Klasse von der Klasse einer Basiskomponente ableitet (z.B. `JPanel`) und dann deren Methode `paintComponent()` überschreibt.

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

Die Variable `g` enthält eine Referenz auf die Zeichenfläche und die „graphische Konfiguration“. Alle Zeichenausgaben laufen über `g`.

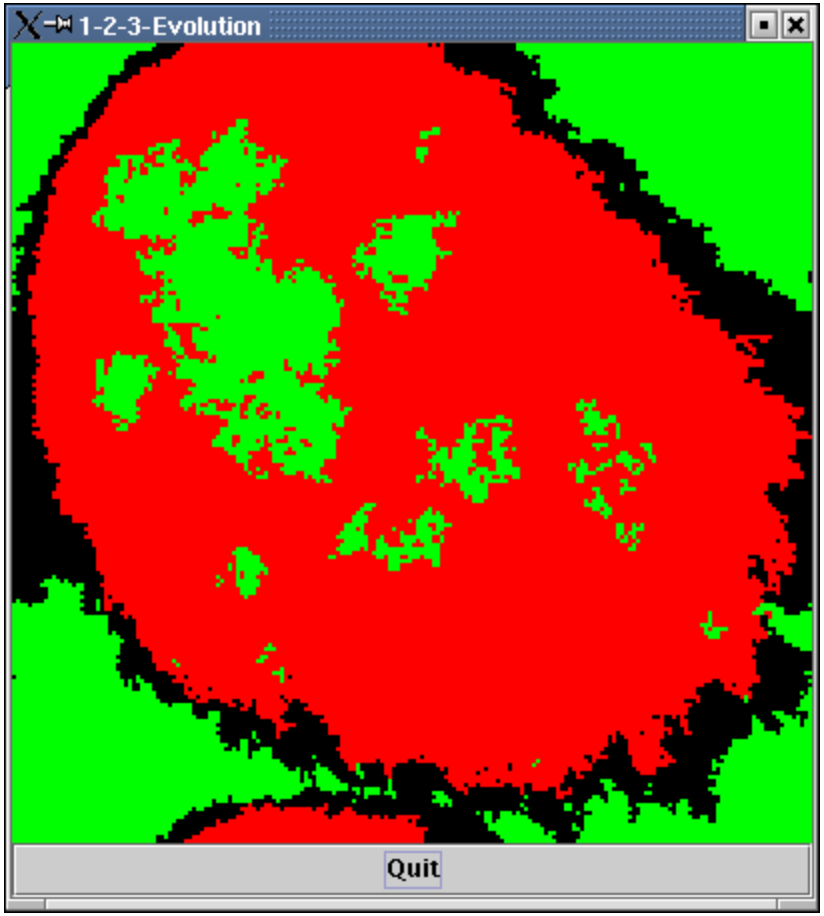
Beispiele für Methoden von `Graphics` sind:

```
public void setColor(Color c);
public void drawLine(int x0, int y0, int x1, int y1);
public void fillRectangle(
    int x0, int y0, int deltaX, int deltaY);
```

Durch Casten eines Graphicsobjekts nach `Graphics2D` stehen weitere Methoden zur Verfügung, die sich an einem zweidimensioneln Graphikkonzept orientieren.

```
Graphics2D g2 = (Graphics2D) g;
```

Diese Animation soll realisiert werden



Beispiel und Fallstudie für effiziente Animation in Swing (1)

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

public class Beispiel extends JFrame {
    // Aktionen beeinhalten die Eigenschaften von Controls
    private static class QuitAction extends AbstractAction {
        QuitAction() { super("Quit"); }
        public void actionPerformed(ActionEvent e) {
            System.exit(0);
        }
    }
    public static void main(String[] argv) throws Exception {
        // definiere das Look & Feel
        UIManager.setLookAndFeel(
          "com.sun.java.swing.plaf.windows.WindowsLookAndFeel");
        new Beispiel(Float.parseFloat(argv[0]));
    }
    ... // hier kommt die eigentliche Initialisierung
}
```

Beispiel und Fallstudie für effiziente Animation in Swing (2)

```
public Beispiel(float fract) {
    // Titel der Superklasse JFrame definieren.
    super("1-2-3-Evolution");
    // setze allgemeine Eigenschaften
    setResizable(false);
    setDefaultCloseOperation(EXIT_ON_CLOSE);
    // definiere einen Quit-Button
    JButton b = new JButton(new QuitAction());
    // erzeuge Animationsobjekt als Komponente
    PlayGround g = new PlayGround(200,200,fract);
    // Bildschirm aufbauen
    // getContentPane(): Behälter von JFrame
    Container contentPane = getContentPane();
    contentPane.add(b, "South"); // Quit-Button ist unten
    contentPane.add(g, "North"); // Graphik ist oben
    pack();
    setVisible(true);
    // Animation starten
    new Thread(g).start();
}
```

Beispiel und Fallstudie für effiziente Animation in Swing (3)

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

public class Playground extends JPanel implements Runnable {
    private final int xSize, ySize; // Größe des Datenfeldes
    private int[][] game;           // Daten
    private boolean ready = false;  // Bedingungsvariable

    /** Legt die Zeichenfläche fest.
        SCALE = Größe eines Punktes. */
    public Dimension getPreferredSize() {
        return new Dimension(SCALE*xSize, SCALE*ySize);
    }
```

Hier sind nur die interessanten Teile aufgeführt.
Wichtig dabei ist, dass als Zeichenfläche mit `JPanel` eine ganz primitive Komponente gewählt wurde (JPanel ist gleichzeitig ein Container).
Da `JPanel` so primitiv ist, muss hier auch die View implementiert werden.

Beispiel und Fallstudie für effiziente Animation in Swing (4)

Die Daten der Klasse `Playground` und die Methode `run()` realisieren das Model. Bei jeder relevanten Änderung wird `repaint()` aufgerufen. Damit nicht während des Zeichnens die Daten verändert werden, ist hier eine explizite Synchronisation eingebaut. Oft verwendet man eine einfache Zeitverzögerung.

```
/** Berechnet neue Datenwerte */
public void run() {
    while(true) {
        /* neu Zeichnen */
        ready =  false;
        repaint();
        /* warten bis komplett gezeichnet wurde */
        synchronized (this) {
            while (!ready) wait();} // Exception !
        }
        /* berechne neue Datenwerte */
        ...
    }
}
```

Beispiel und Fallstudie für effiziente Animation in Swing (5)

```
/**
 * Wird in dem Event Thread aufgerufen
 * U.a. auf Veranlassung repaint()).
 * Stellt die Sicht der Komponente dar.
 * @param g Graphikkonfiguration
 */
public void paintComponent(Graphics g) {
    // Zeichnen des Hintergrunds, wenn nötig.
    //super.paintComponent(g);
    // Zeichnen des Vordergunds
    for (int i=0; i<xSize;i++)
        for(int j=0; j<ySize;j++) {
            g.setColor(colors[game[i][j]]);
            g.fillRect(SCALE*i, SCALE*j, SCALE, SCALE);
         }
        /* Benachrichtigung für run() */
        synchronized(this) {
            ready = true;
            notify();
        }
    }
}
```

Wie kann man die Effizienz verbessern?

- Das pixelweise Zeichnen in Swing ist nicht besonders schnell. (Genau genommen die Erzeugung von Pixelgraphik aus graphischen Grundanweisungen).
- Bei der Animation wird in jedem Schritt nur ein kleiner Teil der Graphik verändert.
- Es sollte also ausreichen, nur diesen Teil neu zu zeichnen.
- Swing erwartet aber, dass stets alles neu gezeichnet wird (Teile der Graphik waren verdeckt, es gibt weitere Graphiken im Hintergrund, Swing hat den Speicher temporär benutzt).
- Ein Ausweg ergibt sich dadurch, dass man ein eigenes **Imageobjekt** verwaltet, dass immer die vollständige Graphik enthält. Dann braucht man da nur noch Änderungen vorzunehmen. Die Kopie des Imageobjekts auf den Bildschirm ist effizient.
- Imageobjekte für den Framebuffer kann man sich aus einer aktuell dargestellten Komponente (hier also `JPanel`) durch Kopie erzeugen.
- Zu jedem `Image` gehört auch ein Graphikbereich
- Bei dieser Methode kann man das doppelte Puffern abschalten.

Beispiel und Fallstudie für effiziente Animation in Swing (5)

```
public void paintComponent(Graphics g) {
    if (img == null) { // first time: initialize
        img = createImage(SCALE*xSize, SCALE*ySize);
        graph = img.getGraphics();
        for (int i=0; i<xSize; i++) {
            for (int j=0; j<ySize; j++) {
                graph.setColor(colors[game[i][j]]);
                graph.fillRect(SCALE*i,SCALE*j,SCALE,SCALE);
            }
        }
    }
    // Änderungen an graph werden in run() durchgeführt.
    g.drawImage(img,0,0,SCALE*xSize,SCALE*ySize,null);
    synchronized(this) {
        ready = true;
        notify();
    }
}
```

Komplexere Modelle

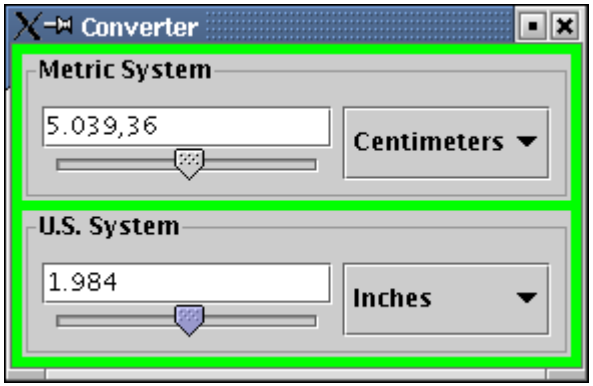
Es gibt eine Reihe Anweisungen, bei denen nicht die Visualisierung beeinflusst wird, sondern nur der Inhalt. In einigen solchen Fällen muss man dann für das Widget ein eigenes Modell zu Verfügung stellen.

Am Beispiel von `JList` wird zunächst eine einfache Anwendung und dann die Verwendung eines Modells gezeigt.

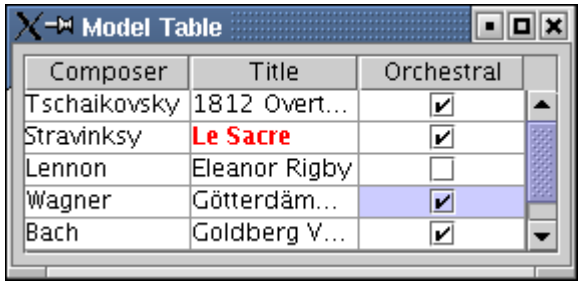
Anmerkung: `JList` und ähnliche Komponenten (`JTable`), erlauben die Modifikation der Sicht, indem man den einzelnen Zellen `CellRenderer`-Objekte zuweist.

Komponenten arbeiten zusammen und brauchen Modelle

Liste, Label, Text



Slider, Combobox



Tabelle, Label, Checkbutton

Jede Swing-Komponente greift auf ein Modell zurück, in dem der Zustand gespeichert ist.

Eigene Modelle braucht man dann, wenn das Defaultmodell nicht ausreicht.

Einfache Liste mit Auswahlmöglichkeit

```
private String[] dlist = {
        "Anchovies", "Bananas", "Cilantro","Doughnuts",
        "Escarrole", "Figues"
    };

final JList list = new JList(dlist);
final JTextField text = new JTextField("");

list.addListSelectionListener( new ListSelectionListener() {
    public void valueChanged(ListSelectionEvent e) {
        text.setText((String)list.getSelectedValue());
    }
} );
// list und text in einem Fenster plazieren...
```

In diesem Fall werden für das Modell und für die UI Defaultklassen verwendet. Die `JList`-Schnittstelle erlaubt nur relativ grobkörnige Änderungen an dem Inhalt der Liste. (Das in `JList` verwendete `DefaultListModel` kann mehr)

Komplexere Modelle

In dem Model-View-Control Muster kennt jedes Modell eine (meist) oder mehrere Views. Jede View verfügt dabei über ein Listener Objekt, das eventuell stattfindete Modelländerungen mitbekommt und die nötigen Aktionen auslöst. Das Model muss über Methoden verfügen, die die angemeldeten Listener speichern.

Dies ist gefordert durch eine Schnittstelle `...Model`. Zusätzlich beschreibt die Modell-Schnittstelle Methoden, mit denen die View den Zustand des Models erfragen kann.

Die Implementierung der View-Registrierung geschieht in der Regel durch die Verwendung einer Klasse `Abstract...Model`.

Für dem einfachen Normalfall gibt es für jedes Widget eine Klasse `Default...Model`.

Wenn man ein eigenes Modell schreibt, dann werden die Abfragemethoden überschrieben. Bei Veränderungen des Models wird erwartet, dass das Modell entsprechende Methoden aufruft (`fire...`).

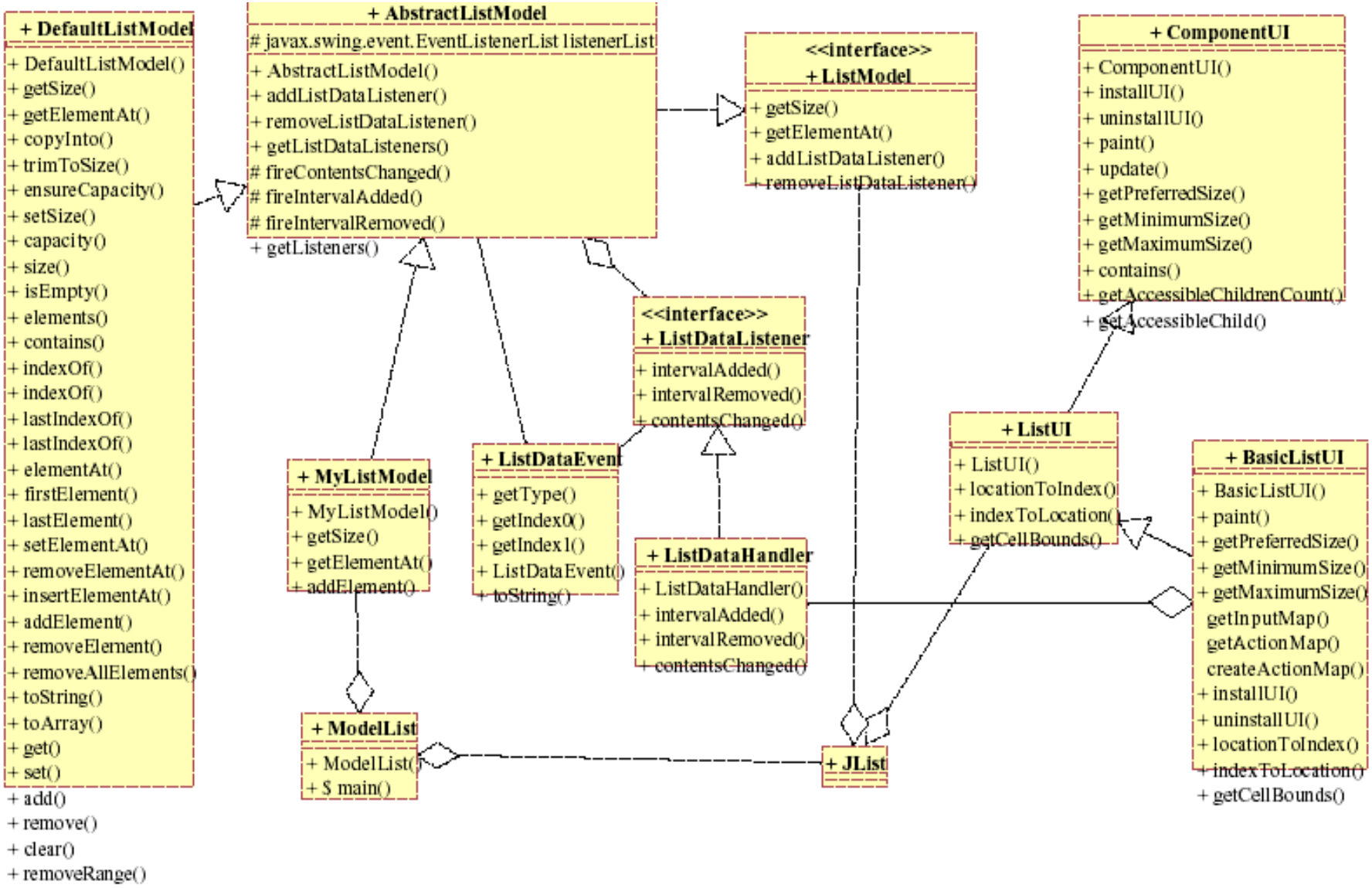
Ein ganz primitives List Modell

```
public class MyListModel extends AbstractListModel {
    private ArrayList data = new ArrayList();

    public MyListModel(String[] initialData) {
        for (int i=0; i<initialData.length; i++)
            data.add(initialData[i]);
    }

    public int getSize() { return data.size(); }

    public Object getElementAt(int index) {
        return data.get(index);
    }

    public void addElement(String s) {
        data.add(s);
        int index = data.size()-1;
        fireIntervalAdded(this, index, index);
    }
}
```

Anwendung mit Veränderung des Models

```
private String[] dlist = {
    "Anchovies", "Bananas", "Cilantro",
    "Doughnuts", "Escarole", "Figues"
    };

final MyListModel model= new MyListModel(dlist);
final JList list = new JList(model);
final JTextField text = new JTextField("");
text.setEditable(true);

text.addActionListener(
    new ActionListener() {
        public void actionPerformed(ActionEvent e) {
            model.addElement(text.getText());
            text.setText("");
        }
    } );
```

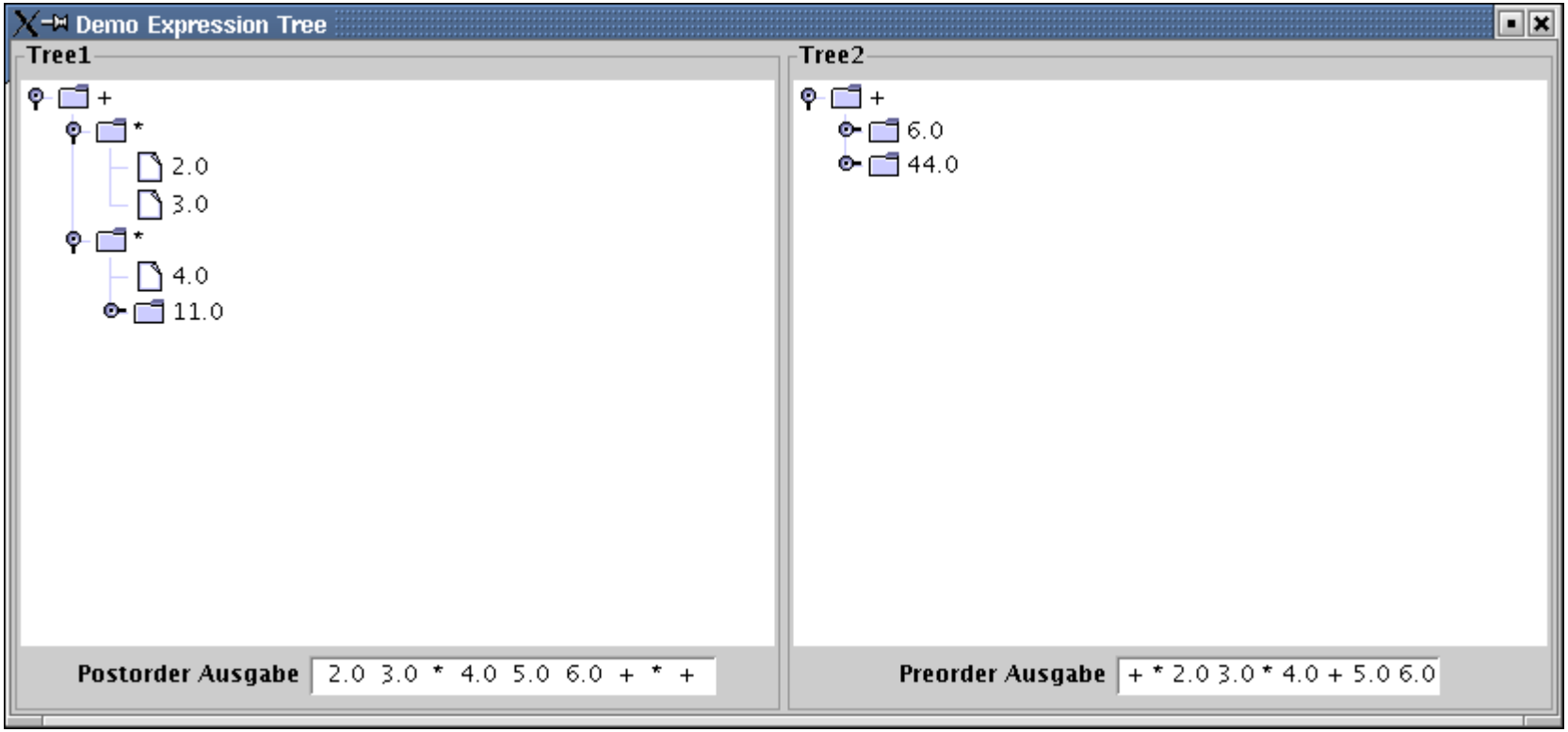
Klassendiagram (eigene Klassen: `MyListModel, ModelList`)



Weitere Bemerkungen zu Modellen

- Außer `JList` sind z.B. auch `JTable` (2 dimensionale Anordnung von Daten) und `JTree` typische Komponenten, bei denen man explizit ein Modell verwendet.
- In vielen Fällen kann man das entsprechende `Default..Model` verwenden (oder davon eine eigene Klasse ableiten).
- Die Modelle sind relativ eng gekoppelt mit der jeweiligen Darstellung.
- Man hat häufig ein internes Objektmodell, dass nochmals separat mit den GUI-Modellen verknüpft ist. Hierfür bietet sich das Observer Muster an.

Es kommt manchmal vor, dass Daten doppelt dargestellt werden



Man muss dabei evtl. unterscheiden zwischen dem Modell der Darstellung und dem abstrakten Datenmodell.

In dem Beispiel bedeutet dies, dass wir 2 TreeModel-Objekte brauchen!

MVC, Observer und Publish/Subscribe

Alle drei Muster sind Architekturmuster, die dafür sorgen, dass Veränderungen an abhängige Klassen gemeldet werden.

- **MVC** realisiert die Abhängigkeit der Sicht vom Modell und die Abhängigkeit von beiden von der Benutzerinteraktion.

- **Observer** löst das Problem, dass ein Objekt bei Veränderungen eines anderen Objekts benachrichtig werden muss. Vergleiche die `Observer` und `Observable` Interfaces in java.util.

- **Publish/Subscrib**e unterscheidet sich von Observer dadurch, dass es ein zusätzliches Vermittlungsobjekt gibt. Das Publish Objekt, meldet Änderungen an das Vermittlungsobjekt, dieses gibt die Meldungen an, die bei ihm registrierten Subscriber weiter.

Ähnlich ist bei allen drei Architekturen, dass die Modell-Objekte über eine Liste der Listener/Observer verfügen, die bei Änderungen zu benachrichtigen sind.

Literatur

James W. Cooper: ***The Design Patterns, Java Companion***
http://www.patterndepot.com/put/8/JavaPatterns.htm
Gibt einen guten Schnelleinstieg in die Grundkonzepte von Swing. Dabei steht die Verwendung von Mustern im Vordergrund.

Eckel, Bruce: ***Thinking in Java, Second Edition***
Prentice Hall, http://www.BruceEckel.com
Sehr gutes fortgeschrittenes Lehrbuch zu Java. Enthält auch eine sehr gute Diskussion über die praktische Verwendung von Swing

Eck, David: ***Introduction to Programming Using Java, Version 4.0***
http://math.hws.edu/eck, 2002
Umfassende Einführung in wichtige Java-Themen und in die Grundlagen von Swing

Creating a GUI with Swing
http://www.javasoft.com
Teil der Java Tutorials. Enthält eine einführende Diskussion der wichtigsten Themen.