Vorlesung
Component Ware und Web-Services

- Komponentenkonzepte -

8. Entwicklung und Verwendung von JavaBeans

Prof. Hans-Gert Gräbe, F. Schumacher
Wintersemester 2003/2004

Inhalt

Beispiel aus „Lehrbuch der Software-Technik“ von Helmut Balzert

Lektion 28 | 3.8 Software-Komponenten

3.8.2 JavaBeans (Seiten 858-870)

1. Aufgabe

- Erstellen eines softwaremäßigen Messinstrumentes
- Daten sollen analog und digital angezeigt werden

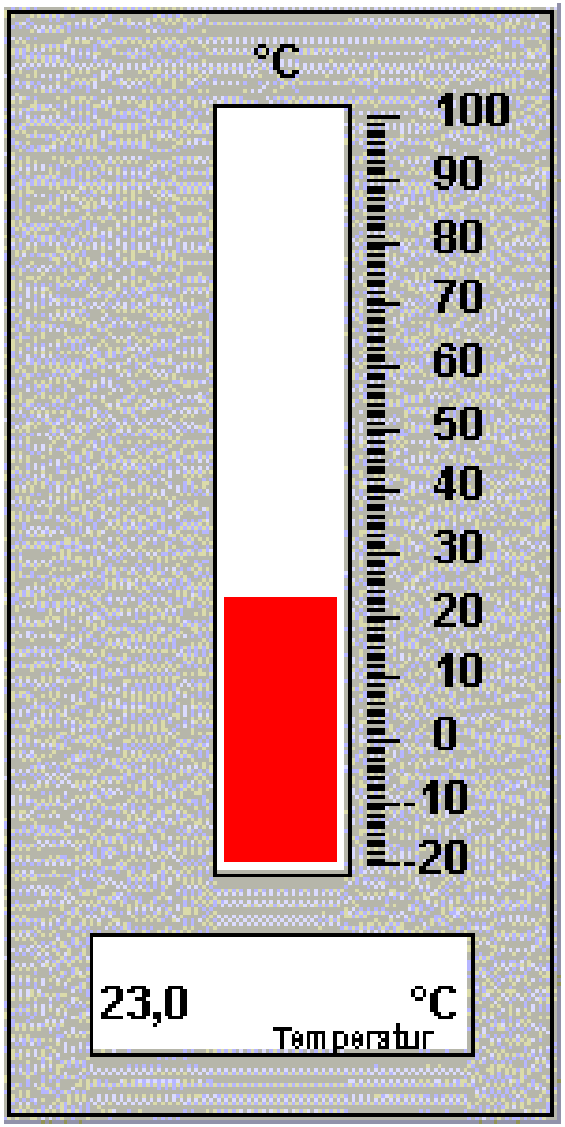
2. JavaBean erstellen

- Eigenschaften
- Ereignisse
- Zusatzinformationen

3. JavaBean in einem einfachen Applet verwenden

- Allgemeine Vorgehensweise
- Einfaches Beispiel zur Steuerung des JavaBeans
- Einbinden des JavaBeans in das Applet
- Verwenden der Eigenschaften und Schnittstellen

Konzipieren eines JavaBeans



Anforderungen:

- minimaler/maximaler Wert (gebunden)
- aktueller Wert (Nebenbedingung)
- Skalierung
- Einheit
- Beschriftung
- Häufigkeit
- Balkenfarbe
- Beobachter sollen von einer Änderung des minimalen, maximalen und aktuellen Wertes benachrichtigt werden
- bei der Änderung des aktuellen Wertes sollen sie die Möglichkeit zum Einlegen eines Vetos haben
- sollte der aktuelle Wert größer als der maximale oder kleiner als der minimale Wert werden, sollen angemeldete Beobachter davon benachrichtigt werden

Konzipieren eines JavaBeans

Spezifikation:

- minimaler/maximaler Wert
  - lesbar und schreibbar
  - Beobachter über Änderungen informieren
- aktueller Wert
  - lesbar und schreibbar
  - mit Nebenbedingung zu realisieren
- Skalierung
  - Feinheitsgrad soll über Parameter eingestellt werden können
- Einheit
  - soll gesetzt werden können
- Beschriftung
  - Kurztext, der angibt, was angezeigt wird
- Häufigkeit
  - Angabe, in welchem Abstand Skalenstriche beschriftet werden
- Balkenfarbe
  - soll geändert werden können

einfache Eigenschaft

Beispiel: einfache Eigenschaft 'Balkenfarbe'

Innerer Zustand **Balkenfarbe** vom Typ **java.awt.Color**

// Initialisierung mit dem Wert **java.awt.Color.red**

```
private java.awt.Color Balkenfarbe = java.awt.Color.red;
```

Funktion wird gerufen, wenn der Eigenschaft **Balkenfarbe** des Objektes ein neuer Wert zugewiesen wird

```
public void setBalkenfarbe(java.awt.Color c) { Balkenfarbe=c; repaint(); }
```

Funktion wird gerufen, wenn die Eigenschaft **Balkenfarbe** des Objektes abgefragt wird

```
public java.awt.Color getBalkenfarbe() { return Balkenfarbe; }
```

Indizierte Eigenschaft

Beispiel: Eigenschaftenvektor x

Innerer Zustand **x** vom Typ **String**, der ein Feld (Vektor, Array) von Werten speichern kann (z.B. Herstellerinformationen)

```
private String[] x;
// Initialisierung von x => Feld mit 5 Elementen => x[0] … x[4]
x = new String[5];
```

Funktion wird gerufen, wenn der Eigenschaft **x[index]** des Objektes ein neuer Wert zugewiesen wird

```
public void setx(int index, String s) {
    if ((index <= x.length – 1) && (index >= 0)) {
        // Index liegt innerhalb der Arraygrenzen
        x[index] = s;
    }
}
```

Indizierte Eigenschaft

Funktion wird gerufen, wenn die Eigenschaft **x[index]** des Objektes abgefragt wird

```
public String getx(int index) {
        if ((index <= x.length – 1) && (index >= 0)) {
                // Index liegt innerhalb der Arraygrenzen
                return x[index];
        }
        else { return ""; }
}
```

Beispiel: gebundene Eigenschaft MaxWert

Innerer Zustand **MaxWert** vom Typ **Float**

```
private float MaxWert;
```

Funktion **get**MaxWert analog zu Balkenfarbe.

```
public float getMaxWert() { return MaxWert; }
```

Gebundene Eigenschaft

Für gebundene Eigenschaften wird ein Instanzattribut angelegt mit einem Objekt der Klasse **java.beans.PropertyChangeSupport** als Wert, welche die Implementierung der gewünschten Funktionalität kapselt.

```
private java.beans.PropertyChangeSupport
        AenderungsUnterstuetzung = new java.beans.PropertyChangeSupport(this);
```

An- und Abmelden von Eigenschaftsänderungs-Beobachtern

```
public void addPropertyChangeListener(java.beans.PropertyChangeListener l) {
        AenderungsUnterstuetzung.addPropertyChangeListener(l);
}

public void removePropertyChangeListener(java.beans.PropertyChangeListener l) {
        AenderungsUnterstuetzung.removePropertyChangeListener(l);
}
```

Gebundene Eigenschaft

Wenn die Eigenschaft **MaxWert** des Objektes gesetzt wird, muss zusätzlich ein Änderungsereignis abgesetzt werden

Syntax:

firePropertyChange(String Name_der_Eigenschaft, Object alter_Wert, Object neuer_Wert)

```
public void setMaxWert(float neuerMaxWert) {
        float alterMaxWert = this.MaxWert;
        this.MaxWert = neuerMaxWert; repaint();
        // Benachrichtigen der Lauscher
        AenderungsUnterstuetzung.firePropertyChange("MaxWert",
                new Float(alterMaxWert), new Float(this.MaxWert))
}
```

Eigenschaft mit Nebenbedingung

Beispiel: Eigenschaft Wert mit Vetobedingung

```
private float Wert;
public float getWert() { return MaxWert; }
```

Für Eigenschaft mit Nebenbedingung wird ein Instanzattribut angelegt mit einem Objekt der Klasse **java.beans.VetoableChangeSupport** als Wert, welche die Implementierung der gewünschten Funktionalitäten kapselt

```
private java.beans.VetoableChangeSupport
    VetoUnterstuetzung = new java.beans.VetoableChangeSupport(this);
```

An- und Abmelden der Beobachter

```
public void addVetoableChangeListener(java.beans.VetoableChangeListener l) {
    VetoUnterstuetzung.addVetoableChangeListener(l);
}

public void removeVetoableChangeListener(java.beans.VetoableChangeListener l){
    VetoUnterstuetzung.removeVetoableChangeListener(l);
}
```

Eigenschaft mit Nebenbedingung

Beim Setzen der Eigenschaft **Wert** des Objektes müssen Vetos berücksichtigt werden

Syntax

fireVetoableChange(String Name_der_Eigenschaft, Object alter_Wert, Object neuer_Wert)

Im Falle eines Vetos wird vom vetoberechtigten Objekt eine (geprüfte) Ausnahme ausgelöst, die mit **throws ...** abgefangen wird

```
public void setWert(float neuerWert) throws PropertyVetoException {
        float alterWert = this.Wert;
        // Benachrichtigen der Beobachter
        VetoUnterstuetzung.fireVetoableChange("Wert",
                new Float(alterWert), new Float(neuerWert));
        if ((MinWert <= neuerWert) && (neuerWert <= MaxWert)) {
                this.Wert = neuerWert;
                repaint();
        }
}
```

Beispiel: Ereignis Bereichsfehler

Dieses Ereignis soll ausgelöst werden, wenn der darzustellende Wert außerhalb des von MinWert und MaxWert vorgegebenen Bereich liegt.

1. Implementieren einer Ereignisklasse mit dem Namen 'Bereichsfehler**Event**', die von der Klasse **java.util.EventObject** erbt
2. Implementieren einer Beobachterschnittstelle mit dem Namen 'Bereichsfehler**Listener**', die von der Klasse **java.util.EventListener** erbt
3. Implementieren der Funktionalitäten, um Beobachter zu verwalten, anzumelden, abzumelden und zu benachrichtigen

Deklaration der Beobachterschnittstelle

Der Name der Operation ist gleich dem Namen des Ereignisses und muss mit einem kleinen Buchstaben beginnen.

```
public interface BereichsFehlerListener extends java.util.EventListener {
    void bereichsFehler(BereichsFehlerEvent e);
}
```

Ereignisse

Deklaration der Ereignisklasse

```
public class BereichsFehlerEvent extends java.util.EventObject {
// 2 Variablen zum Speichern des alten und des neuen Wertes
    public float alterWert, neuerWert;

// Konstruktor, in dem alter und neuer Wert übergeben werden
    BereichsFehlerEvent(Object Ereignisquelle, float alterWert, float neuerWert) {
       // Konstruktor von java.util.EventObject aufrufen
        super(Ereignisquelle);
       // Speichern der Werte
        this.alterWert = alterWert;
        this.neuerWert = neuerWert;
    }
}
```

Verwaltung der Beobachter des Ereignisses

Vektor, der die Beobachter in der Klasse **Messinstrument** verwaltet

```
private java.util.Vector Beobachter = new java.util.Vector();
```

An- und Abmelden von BereichsFehlerListener-Objekten. Da dies nicht mit der Ereignisverteilung vermischt werden darf, sind die Methoden **synchronized**.

```
public synchronized void addBereichsFehlerListener(BereichsFehlerListener l) {
    Abhoerer.addElement(l);
}

public synchronized void removeBereichsFehlerListener(BereichsFehlerListener l) {
    Abhoerer.removeElement(l);
}
```

Ereignisse

Alle angemeldeten BereichsFehlerListener benachrichtigen

Der synchronized-Block gewährleistet, dass während des Kopierens keine Änderungen am Vektor der angemeldeten Beobachter möglich sind.

```
protected void notifyBereichsFehler(BereichsFehlerEvent einEreignis) {
    java.util.Vector AbhoererKopie;

    synchronized(this) {AbhoererKopie = (java.util.Vector)Abhoerer.clone(); }

    for (int i=0; i<AbhoererKopie.size(); i++) {
        ((BereichsFehlerListener)AbhoererKopie.elementAt(i)).bereichsFehler(einEreignis);
    }
}
```

Benachrichtigung auslösen

Damit die angemeldeten Beobachter von dem Ereignis benachrichtigt werden, muss die setter-Methode der Eigenschaft **Wert** aufgebohrt werden.

```
public void setWert(float Wert) throws PropertyVetoException {
    // ... wie bisher
    if ((MinWert<=Wert)&&(Wert<= MaxWert)) {
    // ... wie bisher und weiter
    }
    else {
        // Instanz der Ereignisklasse erzeugen
         BereichsFehlerEvent einEreignis
                = new BereichsFehlerEvent(this, alterWert, Wert);
        // Aufrufen der Benachrichtigungsfunktion
        notifyBereichsFehler(einEreignis);
    }
}
```

BeanInfo

Um zusätzliche Informationen zum Bean (z.B. das repräsentierende Piktogramm) zur Verfügung zu stellen, kann eine begleitende Klasse mit dem Standardnamen **MessinstrumentBeanInfo** erstellt werden, welche die Schnittstelle **BeanInfo** implementiert. Prototyp ist die Klasse **SimpleBeanInfo** mit einer Standardimplementierung der Schnittstelle

```
public class Messinstrument BeanInfo extends SimpleBeanInfo {
    // Lade die entsprechenden Piktogramme
    public Image getIcon(int Groesse) {
        Image Bild = null;
        if (Groesse == BeanInfo.ICON_COLOR_16x16) {
                Bild = loadImage("Mess16.gif");
        }
        else if (Groesse == BeanInfo.ICON_COLOR_32x32) {
                Bild = loadImage("Mess32.gif");
        }
        return Bild;
    }
```

BeanInfo

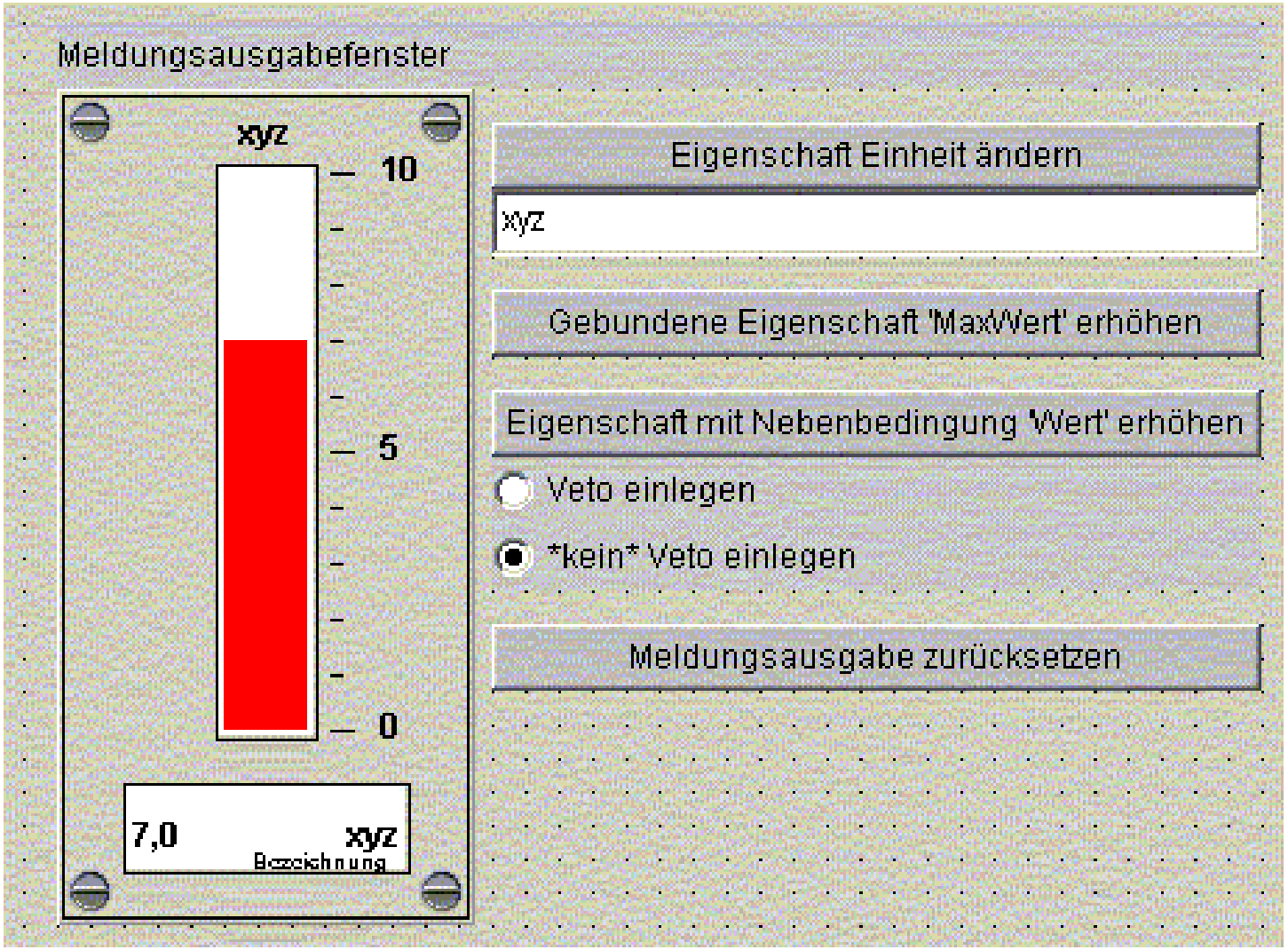
```
        // Liefert alle Eigenschaften, die sichtbar sein sollen
        public PropertyDescriptor[] getPropertyDescriptors() {
                try {
                        PropertyDescriptor wert
                                = new PropertyDescriptor("Wert", Messinstrument.class);
                         PropertyDescriptor maxWert
                                = new PropertyDescriptor("maxWert", Messinstrument.class);
                        …
                        PropertyDescriptor[] pd = {wert,maxWert,...};
                        return pd;
                }
                catch (IntrospectionException e) {
                        throw new Error(e.toString());
                }
        }
}
```

Auslieferung und Benutzung

- Fehlen noch einige weitere Eigenschaften aus der Spezifikation (Skalierung, Beschriftung, Einheit, Häufigkeit)
- Die grafische Realisierung muss natürlich auch noch implementiert werden.
  - Nutzung eines grafischen Programmierwerkzeugs
- Packen aller Bestandteile als package **Messkomponenten** in eine jar-Datei **MessKomponenten.jar**
- Siehe Begleit-CD zu [Balzert: Lehrbuch der Software-Technik]
- Testumgebung mit einem einfachen Applet als Beispiel, wie eine solche Bean in Anwendungen eingebunden wird.

Testen des JavaBeans

Testen der Funktionalität des JavaBeans in einem einfachen Applet



Testen des JavaBeans

Anhand des Applets sollen folgende Funktionalitäten des JavaBeans veranschaulicht werden:

Eigenschaft 'Einheit' ändern:

- Zuweisung eines neuen Wertes zu einer einfachen Eigenschaft

Gebundene Eigenschaft 'MaxWert' erhöhen:

- Erhöhen des MaxWertes um 1
- Benachrichtigung des angemeldeten Beobachter

Eigenschaft mit Nebenbedingung 'Wert' erhöhen:

- Erhöhen des Wertes um 1
- Möglichkeit, ein Veto einzulegen
- Auslösen des benutzerdefinierten Ereignisses BereichsFehler, wenn Wert MaxWert übersteigt

Einbinden des JavaBeans in ein Applet

```
class TestApplet extends Container implements ... // dazu später
{
// Attribute der Containerklasse anlegen und mit Aktionen verbinden
private JButton B_WertChange=new JButton("Wert ändern");
B_WertChange.add(new WertChangeAction());

private JTextField TF_Einheit=new JTextField("°C",50);
public String getEinheit() { return TF_Einheit.getText(); }

// Attribut M_Test mit Messinstrument-Bean-Instanz
private Messinstrument M_Test = new Messinstrument();
public Messinstrument getMessinstrument() { return M_Test; }

// Anpassung der Bean beim Einbau in die Container-Umgebung (Assemblier-Zeit)
      M_Test.setLabel("Bezeichnung");
      try { M_Test.setWert(7.0F); }
      catch(PropertyVetoException e) { .. }
      ...
```

Eigenschaften ändern

Ändern einer einfachen Eigenschaft

```
class EinheitSetzen extends AbstractAction { ...
        public void actionPerformed(ActionEvent a) {
                Messinstrument M=adaptee.getMessinstrument();
                M.setEinheit(adaptee.getEinheit());
        }
}
```

Ändern einer gebundenen Eigenschaft

```
...     M.setMaxWert(M.getMaxWert() + 1);
```

Das Feuern des PropertyChangeEvent wird von M besorgt. Dafür sind keine zusätzlichen Vorkehrungen zu treffen.

Eigenschaften beobachten

Zum Abhören des PropertyChangeEvent muss TestApplet die entsprechende Schnittstelle **java.beans.PropertyChangeListener** implementieren ...

```
class TestApplet ... implements PropertyChangeListener { ...

    public void propertyChange(PropertyChangeEvent event) {
        TF_Ausgabe.setText("Property " + event.getPropertyName() + " wurde von " +
        event.getOldValue() + " auf " + event.getNewValue() + " geändert" );
    }
```

... und sich als Beobachter anmelden

```
M_Test.addPropertyChangeListener(this);
```

Eigenschaft mit Nebenbedingung

Ändern einer Eigenschaft mit Nebenbedingung

```
public void actionPerformed(ActionEvent a) {
        Messinstrument M=adaptee.getMessinstrument();
        try {
                M.setWert(M.getWert() + 1);
                TF_Ausgabe.setText("Aktueller Wert wurde erhöht");
        } catch (PropertyVetoException e) {
                TF_Ausgabe.setText("Wegen Veto wurde der aktuelle Wert nicht geändert");
        }
}
```

Der Programmfluss wird ggf. unterbrochen und zur aufrufenden Methode zurückgekehrt, welche auf die Ausnahme reagieren kann.

Dieses Verhalten wird von TestApplet als vetoberechtigtem Beobachter gesteuert.

Veto einlegen

Dazu muss TestApplet die Schnittstelle **java.beans.VetoableChangeListener** implementieren ...

```
class TestApplet ... implements VetoableChangeListener { ...

    public void vetoableChange(PropertyChangeEvent event) throws
        PropertyVetoException {
        if (this.isVeto) { throw new PropertyVetoException(); }
    }
```

... und sich als Vetoberechtigter anmelden

```
M_Test.addVetoableChangeListener(this);
```

Benutzerdefinierte Ereignisse

Ähnlich funktioniert der Umgang mit benutzerdefinierten Ereignissen.

Um ein **BereichsFehlerEvent** zu verarbeiten, muss TestApplet die Schnittstelle **Messkomponenten.BereichsFehlerListener** implementieren ...

```
class TestApplet ... implements BereichsFehlerListener { ...

    public void bereichsFehler(BereichsFehlerEvent event) {
        TF_Ausgabe.setText("BereichsfehlerEvent gefeuert: Änderung von " +
                event.alterWert + " nach " + event.neuerWert + " nicht möglich!");
    }
}
```

.. und sich als Beobachter anmelden

```
M_Test.addBereichsFehlerListener(this);
```

In diesem Fall wird **keine** Ausnahme ausgelöst (siehe die Implementierung in Messinstrument.setWert())