LE 28
1

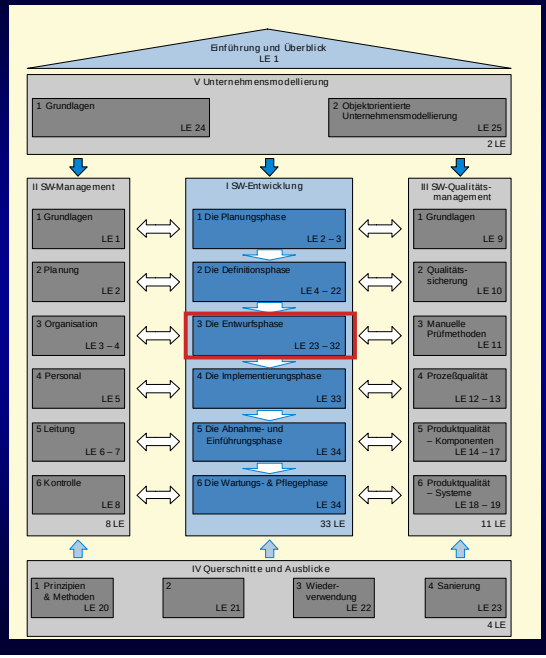
Software-Technik

3 Die Entwurfsphase Software-Komponenten

[COM gekürzt]

Prof. Dr. Helmut Balzert
Lehrstuhl für Software-Technik
Ruhr-Universität Bochum

LE 28
2



LE 28
3

Inhalt

LE 28
4

3.8.1 Halbfabrikate und ihre Schnittstellen

- **Komponentenbasierte Softwareentwicklung**
  - Erlaubt die einfache, schnelle und preiswerte Herstellung individueller, integrierter Anwendungen durch Zusammenbau von vorgefertigten Halbfabrikaten bzw. Komponenten
- Dazu benötigt man **Halbfabrikate**, die...
  - i. Allg. kleiner als Anwendungen sind, d.h. einen stärkeren Komponenten- bzw. Bausteincharakter besitzen
  - deutlich größer als Klassen sind, d.h. mehr Funktionalität kapseln
  - Komplexität verbergen.

LE 28 5

3.8.1 Halbfabrikate und ihre Schnittstellen

- Halbfabrikat *(componentware)*
  - Ist ein abgeschlossener, binärer Software-Baustein, der eine anwendungsorientierte, semantisch zusammengehörende Funktionalität besitzt, die nach außen über Schnittstellen zur Verfügung gestellt wird
  - Beim Entwurf des Halbfabrikats wurde auf hohe Wiederverwendbarkeit großer Wert gelegt
  - Beispiele
    - Rechtschreibprüfung, Silbentrennung, Seitenvorschau, Druckknöpfe, Textfelder, Tabellen.

LE 28 6

3.8.1 Halbfabrikate und ihre Schnittstellen

- Erfolgreicher Einsatz von Halbfabrikaten:
  - Komponenten müssen an bestimmten, wohldurchdachten Stellen offen sein
    - Komponenten sollen daher mit speziellen austauschbaren Ablaufklassen kooperieren
    - Analog sollte die Art der Fehlerentdeckung und -behandlung geändert werden können
  - Schnittstellen wichtiger oder häufig eingesetzter Komponenten müssen standardisiert werden
  - Eine für den Anwendungsbereich sinnvolle Begriffswelt ist nötig
  - Für Pflege, Wartung und Weiterentwicklung ist ein nachvollziehbares Bereichsmodell erforderlich.

LE 28 7

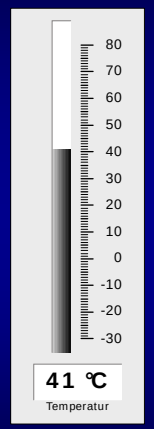
3.8.1 Halbfabrikate und ihre Schnittstellen

- Vorgehensweise
  1. Auswahl eines geeigneten Architekturrahmens
  2. Auswahl geeigneter Halbfabrikate
  3. Anpassung der Halbfabrikate
  4. Verbinden der Halbfabrikate
  5. Überprüfung, ob das entstehende Produkt den gewünschten Anforderungen entspricht
- Notwendige Voraussetzungen
  - Sprachunabhängigkeit
  - Plattformunabhängigkeit
  - Verteilbarkeit.

LE 28 8

3.8.1.1 Komponentenmodelle

- 2 Komponentenmodelle für *Clients*
  - *JavaBeans*-Modell von Sun
  - COM/*ActiveX*-Modell von Microsoft
  - Beispiel
    - Firma ProfiSoft: Es soll ein Messinstrument implementiert werden, das Daten analog über eine Messsäule und digital über eine entsprechende Digitalanzeige visualisiert



- 3 Komponentenmodelle für *Server*
  - COM+ von Microsoft
  - *Enterprise JavaBeans* von Sun
  - CORBA-Modell der OMG

LE 28
9

3.8.2 *JavaBeans*

- *JavaBeans*-Modell
  - Ziel: Das Zusammenfügen von Komponenten mit Hilfe von visueller Programmierung zu ermöglichen
  - Komponenten können einfach mit der Maus »zusammengeklickt« werden
  - *»A Java Bean is a reusable software component that can be manipulated visually in a builder tool«* /Sun 97/
  - Jede Java-Klasse ist im Prinzip eine *JavaBean*
  - Um jedoch die Kompositions- und Anpassungsfähigkeit einer *JavaBean* optimal zu unterstützen, müssen gewisse Konventionen beachtet werden.

LE 28
10

3.8.2.1 Introspektion

- Introspektion
  - Komponente besitzt die Fähigkeit, relevante Informationen nach außen offen zu legen
  - Durch Beachtung von Namenskonventionen bei Operations- und Klassennamen können Entwicklungswerkzeuge die gewünschten Informationen über Eigenschaften, Operationen und Ereignisse erhalten
  - Zusätzliche Informationen können über die `BeanInfo`-Schnittstelle nach außen mitgeteilt werden
    - Beispiel: Piktogramm, das die *JavaBean* in der Werkzeugleiste einer Entwicklungsumgebung repräsentieren soll.

LE 28
11

3.8.2.1 Introspektion

- Entwurfsüberlegungen
  - Welches Problem soll durch die *JavaBean* gelöst werden?
  - Wo und wie wird die *JavaBean* eingesetzt (Kontext)?
  - Welche Veränderungen und Erweiterungen sind in Zukunft denkbar?
- Antwort
  - Detaillierte Beschreibung der 3 Hauptteile einer *JavaBean*:
    - Eigenschaften
    - Ereignisse
    - Operationen.

LE 28
12

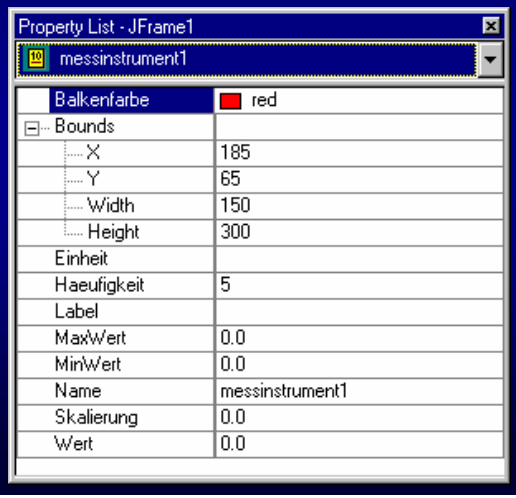
3.8.2.2 Anpassbarkeit

- Nachträgliche Anpassung *(customization)*
  - Ein Entwickler, der die Komponente nach ihrer Übersetzung und Auslieferung benutzt, kann sie an seine Bedürfnisse in gewissem Maße anpassen
  - Er kann Eigenschaften setzen, die das Erscheinungsbild und Verhalten beeinflussen
  - Hierfür gibt es 2 Möglichkeiten
    - Es wird ein Eigenschaftseditor *(property editor)* des Entwicklungswerkzeuges benutzt
    - Die *JavaBean* stellt eigene Klasse *(customizer)* zur Verfügung, die eine Anpassung ermöglicht
      - Oftmals wird hierfür ein Assistent *(wizard)* benutzt, der Schritt für Schritt durch die nötigen Schritte führt.

LE 28
13

3.8.2.2 Anpassbarkeit

- **Beispiel**
  - Die Firma ProfiSoft verwendet Eigenschaftseditoren



LE 28
14

3.8.2.3 Eigenschaften

- **Indizierte Eigenschaften**
  - Können mehrere Werte annehmen, die über einen Index angesprochen werden
- **Gebundene Eigenschaften**
  - Werden verwendet, um Änderungen von Eigenschaften externen Beobachtern mitzuteilen
  - Andere Objekte können sich bei der *JavaBean* als Eigenschaftsabhörer registrieren
- **Eigenschaften mit Nebenbedingungen**
  - Erlauben es externen Beobachtern *(vetoable change listener), vor* der Änderung von Eigenschaften, ihr Veto einzulegen und damit eine Änderung des Eigenschaftswertes zu verbieten
  - →Entwurf.

LE 28
15

JavaBeans-Konventionen für Eigenschaften

- **Zugriffsoperationen**
  - Für jede Eigenschaft wird eine `get`- und eine `set`-Operation implementiert
  - Soll die Eigenschaft nur lesbar oder nur schreibbar sein, dann ist nur eine der beiden Operationen zu implementieren
- **Einfache Eigenschaften *(simple properties)***

```
public <Eigenschaftstyp> get<Eigenschaftsname>()
public void set<Eigenschaftsname>
   (<Eigenschaftstyp> a)
```

  - Für Eigenschaften vom Typ `boolean` auch:

```
public boolean is<Eigenschaftsname>.
```


LE 28
16

JavaBeans-Konventionen für Eigenschaften

- **Indizierte Eigenschaften *(indexed properties)***

```
public <Eigenschaftstyp> get<Eigenschaftsname>(int
index)
public void set<Eigenschaftsname>(int index,
   <Eigenschaftstyp> a)
```

- **Gebundene Eigenschaften**
  - Zusätzlich zu den normalen Zugriffsoperationen müssen Operationen zum Verwalten von `PropertyChangeListener`-Objekten implementiert werden:

```
public void
addPropertyChangeListener(PropertyChangeListener x)
public void
removePropertyChangeListener
                        (PropertyChangeListener x).
```

LE 28
17

JavaBeans-Konventionen für Eigenschaften

- Eigenschaften mit Nebenbedingungen
  - Bei Eigenschaften mit Nebenbedingungen muss die `set`-Operation folgende Signatur besitzen:

```
public void set<Eigenschaftsname>(<Eigenschaftstyp>
   wert) throws PropertyVetoException
```

  - Zusätzlich müssen Operationen zum Verwalten von `VetoableChangeListener`-Objekten implementiert werden:

```
public void addVetoableChangeListener
                        (VetoableChangeListener x)
public void removeVetoableChangeListener
                        (VetoableChangeListener x).
```


LE 28
18

3.8.2.3 Eigenschaften

- Entwurf
  - Die Eigenschaften sollten möglichst beim 1. Entwurf spezifiziert werden, um Änderungen an der Schnittstelle zu vermeiden
  - Zusätzlich muss für jede Eigenschaft entschieden werden
    - ob sie nur lesbar, nur schreibbar, lesbar und schreibbar ist
    - ob sie gebunden, indiziert oder mit Nebenbedingungen realisiert wird.

LE 28
19

3.8.2.3 Eigenschaften

- Beispiel:
  Eigenschaften für das Messinstrument-*Bean*
  - Minimaler/maximaler Wert
    - 2 Eigenschaften für den Wertebereich
    - Die Eigenschaften sind lesbar und schreibbar
    - Die Eigenschaften sind gebunden
      - Wenn eine Grenze zur Laufzeit verändert wird, sollen registrierte Abhörer darüber informiert werden
  - Wert
    - 1 Eigenschaft für den aktuell angezeigten Wert
    - Die Eigenschaft ist lesbar und schreibbar
    - Mit Nebenbedingungen, falls später weitere Restriktionen geltend gemacht werden.

LE 28
20

3.8.2.3 Eigenschaften

- Skalierung
  - 1 Eigenschaft über die die Feinheit der Skala geändert wird (einfach, lesbar, schreibbar)
- Einheit
  - Einheit für die angezeigten Messdaten (einfach, lesbar, schreibbar), z.B. °C
- Beschriftung
  - Kurztext, der beschreibt, was gerade angezeigt wird (Windgeschwindigkeit, Innentemperatur)
- Häufigkeit
  - In welchem Abstand werden Striche der Skala mit Ziffern beschriftet
- Balkenfarbe
  - Änderung der Farbe des Anzeigebalkens.

LE 28
21

3.8.2.4 Ereignisse

- Ereignisse
  - Eine Komponente kann sich zur Laufzeit bei einer anderen Komponente als Empfänger eines Ereignisses an- und abmelden
  - 2 Ereignisquellen
    - Nur 1 Abhörer anmeldbar *(unicast event source)*
    - Mehrere Abhörer anmeldbar *(multicast event s.)*

LE 28
22

3.8.2.4 Ereignisse

- Beispiel
  - Die Messinstrument-*Bean* soll ein Ereignis verschicken, wenn der gesetzte Wert nicht im Wertebereich liegt, der durch den minimalen und maximalen Wert vorgegeben ist
  - Gegebenenfalls kann nun ein Beobachter bei der Komponente angemeldet werden, der entsprechende Maßnahmen beim Eintreten des Ereignisses ergreift
- →Operationen.

LE 28
23

JavaBeans-Konventionen für Ereignisse

- Ereignisse
  - Die Operationen zur Verwaltung von Ereignisabhörern müssen konform zu den folgenden Namenskonventionen implementiert werden
- *Unicast*-Ereignisquelle

```
public void add<Abhörertyp>(<Abhörertyp> l) throws
   TooManyListenersException
public void remove<Abhörertyp>(<Abhörertyp> l)
```

- *Multicast*-Ereignisquelle
- `public void add<Abhörertyp>(<Abhörertyp> l)`

```
public void remove<Abhörertyp>(<Abhörertyp> l).
```


LE 28
24

3.8.2.5 Operationen

- Operationen
  - Alle zusätzlich zu den `get`- und `set`-Operationen implementierten öffentlichen Operationen werden nach außen exportiert
  - Sie können bei vielen Werkzeugen durch visuelle Programmierung zum Beispiel mit Ereignisabhörern verknüpft werden
  - Beispiel
    - Die Firma ProfiSoft identifiziert folgende öffentliche Methoden für das Messinstrument-*Bean:*
      - Zu jeder Eigenschaft wird eine `get`- und `set`-Operation implementiert.

LE 28
25

3.8.2.6 Implementierung einer *JavaBean*

- **Beispiel**
  - Implementierung der einfachen Eigenschaft »Balkenfarbe«

```
//Eigenschaften ...
//Voreinstellung: Farbe Rot
private Color Balkenfarbe = Color.red;
...
//get-/set-Operationen
...
public void setBalkenfarbe(Color c)
{  Balkenfarbe = c; repaint();
}
public Color getBalkenfarbe()
{  return Balkenfarbe;
}.
```


LE 28
26

3.8.2.6 Implementierung einer *JavaBean*

- **Beispiel**
  - Implementierung der gebundenen Eigenschaften minimaler und maximaler Wert mit Hilfe der Klasse `PropertyChangeSupport`

```
//Eigenschaften ...
private float MaxWert, MinWert;
//Für gebundene Eigenschaften wird ein Objekt der
//JavaBeans-API-Klasse PropertyChangeSupport erzeugt
private PropertyChangeSupport
   AenderungsUnterstuetzung =
                  new PropertyChangeSupport(this);. .
```


LE 28
27

3.8.2.6 Implementierung einer *JavaBean*

```
//An- & Abmelden von PropertyChangeListener-Objekten
public void addPropertyChangeListener
                        (PropertyChangeListener l)
{AenderungsUnterstuetzung.
                     addPropertyChangeListener(l);}
public void removePropertyChangeListener
                         (PropertyChangeListener l)
{AenderungsUnterstuetzung.
                  removePropertyChangeListener(l);}
//get-/set-Operationen
public void setMaxWert(float MaxWert)
{ float alterMaxWert = this.MaxWert;
  this.MaxWert = MaxWert;   repaint();
  AenderungsUnterstuetzung.firePropertyChange("MaxWert
  ",new Float(alterMaxWert),new Float(MaxWert));}
public float getMaxWert(){  return MaxWert; }... .
```


LE 28
28

3.8.2.6 Implementierung einer *JavaBean*

- Implementierung der Eigenschaft mit Nebenbedingungen `Wert`

```
//Es wird ein Objekt der Hilfsklasse
//VetoableChangeSupport erzeugt
private VetoableChangeSupport VetoUnterstuetzung = new
  VetoableChangeSupport(this);...
//An- & Abmelden von VetoabelChangeListener-Objekten
public void
 addVetoableChangeListener(VetoableChangeListener l)
{ //An die JavaBeans-API-Hilfsklasse delegieren
  VetoUnterstuetzung.addVetoableChangeListener(l);}
public void
removeVetoableChangeListener(VetoableChangeListener l)
{VetoUnterstuetzung.removeVetoableChangeListener(l);
}... .
```

LE 28
29

3.8.2.6 Implementierung einer *JavaBean*

```
public void setWert(float Wert) throws
   PropertyVetoException
{ float alterWert = this.Wert;
   //Alle Beobachter benachrichtigen. Wenn einer der
   //Beobachter sein Veto einlegen möchte, so wird eine
   //PropertyVetoException ausgelöst
   VetoUnterstuetzung.fireVetoableChange("Wert",
       new Float(alterWert), new Float(Wert));
   if ((MinWert <= Wert) && (Wert <= MaxWert))
   { this.Wert = Wert; repaint()}  ...
}
public float getWert()
{ return Wert;
}.
```


LE 28
30

3.8.2.6 Implementierung einer *JavaBean*

- Für benutzerdefinierte Ereignisse sind 3 Schritte durchzuführen:
  - 1 Implementieren einer Beobachter-Schnittstelle mit dem Namen `<Ereignisname>Listener`
    - a Sie deklariert eine Operation, die aufgerufen wird, um den Beobachter zu benachrichtigen: `void <ereignisname>(<Ereignisname>Event e)`
      - Die Operation heißt wie das Ereignis, nur dass der Operationsname mit einem kleinen Buchstaben beginnen muss
  - 2 Implementieren einer Ereignis-Klasse mit dem Namen `<Ereignisname>Event`, die von der Klasse `java.util.EventObject erbt.`

LE 28
31

3.8.2.6 Implementierung einer *JavaBean*

- 3 Hinzufügen von Funktionalität, um Beobachter an- und abzumelden sowie zu benachrichtigen
  - a Zunächst muss eine Liste für die Verwaltung der Beobachter deklariert werden
    - In der Regel die Klasse `java.util.Vector`
  - b Die Operationen zum An- und Abmelden von Abhörern müssen implementiert werden:
    - `add<Ereignisname>Listener` und `remove<Ereignisname>Listener`
    - Achtung: Die Signatur der Operation `add<Ereignisname>Listener ist` für *unicast*- und *multicast*-Ereignisquellen unterschiedlich
  - c Es muss noch eine Operation zum Benachrichtigen der registrierten Abhörer implementiert werden.

LE 28
32

3.8.2.6 Implementierung einer *JavaBean*

- Beispiel
  - Die Messinstrument-*Bean* soll das Ereignis `BereichsFehler` verschicken, wenn der Wertaußerhalb des durch den minimalen und maximalen Wert vorgegebenen Bereichs liegt
  - Schnittstelle für die Abhörer

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

LE 28
33

3.8.2.6 Implementierung einer *JavaBean*

- Klasse für das Ereignis

```
public class BereichsFehlerEvent extends
   java.util.EventObject
{  public float minNeu, maxNeu;
   BereichsFehlerEvent(Object Ereignisquelle,
        float minNeu, float maxNeu)
    {   super(Ereignisquelle);
        this.minNeu = minNeu;
        this.maxNeu = maxNeu;
    }
}.
```


LE 28
34

3.8.2.6 Implementierung einer *JavaBean*

- Funktionalität zum An- und Abmelden sowie zum Benachrichtigen der Abhörer in der Klasse `Messinstrument`

```
//Vector, der die Abhörer verwaltet
private java.util.Vector Abhoerer =
                    new java.util.Vector();...
//An- & Abmelden von BereichsFehlerListener-Objekten
public synchronized void
addBereichsFehlerListener(BereichsFehlerListener l)
{ Abhoerer.addElement(l);}
public synchronized void
removeBereichsFehlerListener(BereichsFehlerListener l)
{ Abhoerer.removeElement(l);}.
```


LE 28
35

3.8.2.6 Implementierung einer *JavaBean*

```
//Alle angemeldeten BereichsFehlerListener
//benachrichtigen
protected void notifyBereichsFehler()
{ java.util.Vector AbhoererKopie;
  //Hier wird ein synchronized-Block eingefügt,
  //um zu gewährleisten,
  //dass alle vor dem Verschicken des Ereignisses
  //angemeldeten Beobachter benachrichtigt werden
  synchronized(this)
  {AbhoererKopie = (java.util.Vector)Abhoerer.clone();
  }
  for (int i=0; i < AbhoererKopie.size(); i++)
  {((BereichsFehlerListener)AbhoererKopie.
       elementAt(i)).bereichsFehler(einEreignis);
  }
}.
```


LE 28
36

3.8.2.6 Implementierung einer *JavaBean*

- Beispiel
  - `BeanInfo`-Klasse für die Messinstrument-*Bean*

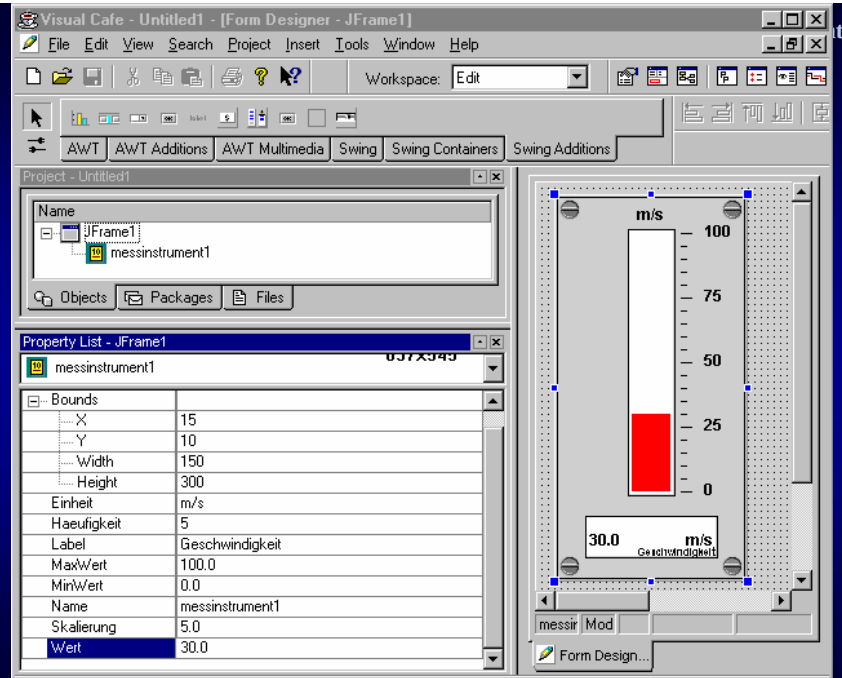
```
public class MessinstrumentBeanInfo extends
   SimpleBeanInfo
{ //Liefere die entsprechenden Piktogramme
    public Image getIcon(int Groesse)
    { Image Bild = null;
      if (Groesse == BeanInfo.ICON_COLOR_16x16)
        { Bild = loadImage("Mess16.gif");
        }
        else if(Groesse==BeanInfo.ICON_COLOR_32x32)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

LE 28
37

3.8.2.7 Auslieferung und Benutzung

- **Auslieferung**
  - Die *JavaBean* mit allen benötigten Ressourcen in eine Archiv-Datei mit der Endung `.jar` verpackt, um sie als Einheit verschicken zu können.
  - Eine `.jar`-Datei ist eine nach dem bekannten ZIP-Algorithmus komprimierte Archiv-Datei
  - Will man eine *JavaBean* in einem Entwicklungswerkzeug verwenden, so muss man in der Regel zunächst mit dem Entwicklungswerkzeug die `.jar` Datei der Komponente einlesen
  - Das Entwicklungswerkzeug merkt sich dann den Pfad unter dem das Archiv zu finden ist und welche Komponenten in dem Archiv enthalten sind.

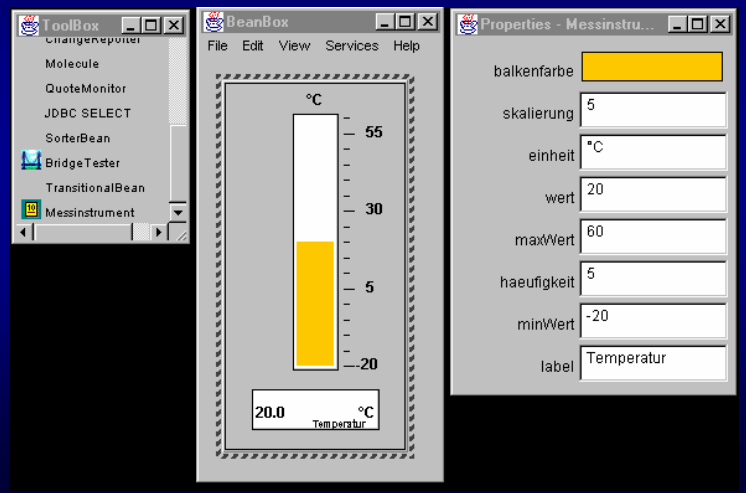
LE 28
38



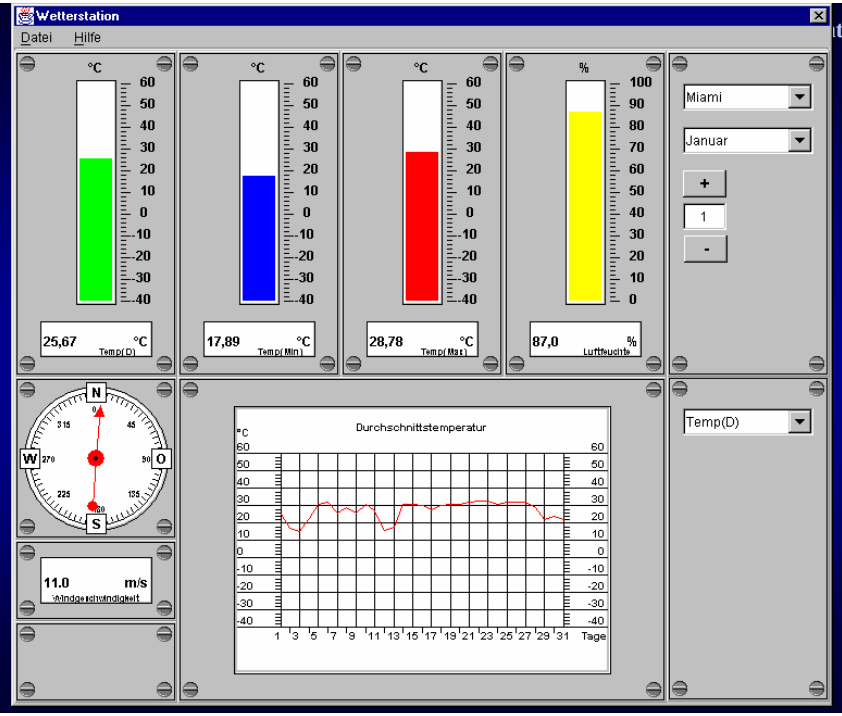
LE 28
39

3.8.2.7 Auslieferung und Benutzung

- Test der Komponente in verschiedenen Umgebungen: *BeanBox* der Firma Sun



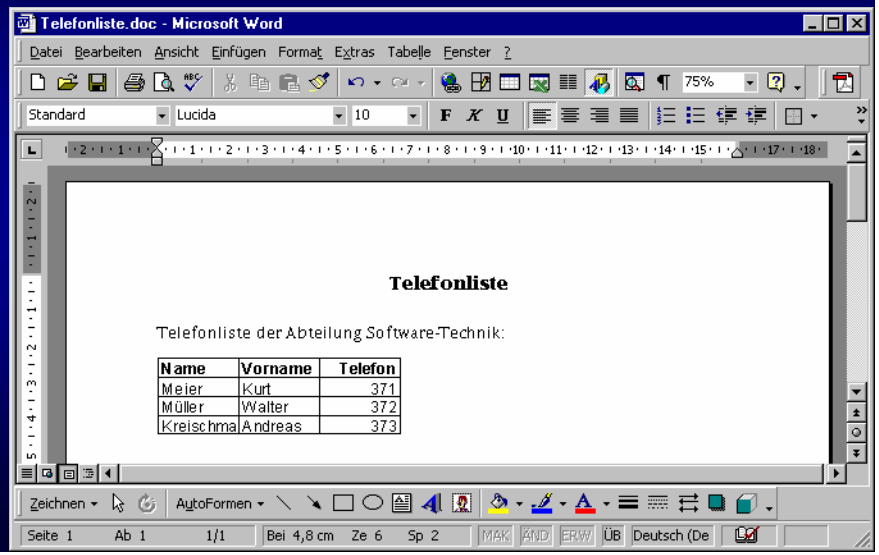
LE 28
40



LE 28
41

3.8.3 COM *(Component Object Model)*

- **Historie**



LE 28
42

3.8.3 COM *(Component Object Model)*

- Teildokumente bzw. Komponenten werden in ein Zentraldokument / einen Container eingebettet
- OLE 1: Modell speziell für Verbunddokumente
- OLE 2: Allgemeines Modell für Komponenten
  - Umbenennung in *Component Object Model*

- ***ActiveX***
  - Subsummiert alle Konzepte von OLE
  - + *ActiveX*-Steuerelemente
- **DCOM *(Distributed COM)***
  - Zugriff auf COM-Objekte auf entfernten CS
- **COM+**
  - Erweiterung von COM um *server*seitige Konzepte.

LE 28
43

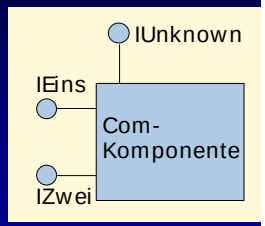
3.8.3 COM *(Component Object Model)*

- **Grundlagen**
  - COM ist Bestandteil der *Windows*-Betriebssysteme
  - Im Gegensatz zu *JavaBeans* definiert COM einen binären Standard
    - Keine Vorgabe, wie die Bindung einer konkreten Programmiersprache an COM erfolgen soll
  - Unabhängig von Programmiersprachen
  - Erzeugung von Objekten
    - *Clients* können von einer COM-Komponente Objekte erzeugen und diese COM-Objekte benutzen, indem sie einen Zeiger auf eine Schnittstelle der Komponente anfordern
    - Über diesen Zeiger kann der *Client* dann Operationen der Komponente aufrufen.

LE 28
44

3.8.3 COM *(Component Object Model)*

- **COM und Schnittstellen**
  - COM-Komponenten implementieren i. Allg. nicht nur eine Schnittstelle, sondern mehrere



  - COM-Objekt vs. COM-Komponente
    - Literatur
      - COM-Objekt (COM *object)* wird irreführenderweise synonym für die Komponenten und die von ihnen erzeugten Objekte benutzt
    - Nachfolgend wird explizit zwischen einer COM-Komponente und einem COM-Objekt unterschieden
    - Ein COM-Objekt ist ein von einer COM-Komponente erzeugtes Objekt.

LE 28
45

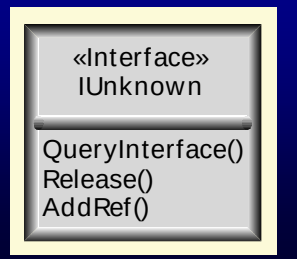
3.8.3 COM *(Component Object Model)*

- 2 Schnittstellen-Namen
  - Einen textueller Name, der nicht eindeutig ist und per Konvention mit dem Buchstaben »I« beginnt
  - GUID *(Global Unique Identifier)*
    - Global eindeutige Nummer, die nach einem speziellen Algorithmus gebildet wird und in Zeit und Raum eindeutig ist
    - Im Zusammenhang mit Schnittstellen werden GUIDs auch als *Interface Identifier* (IIDs) bezeichnet
  - Beispiel
    - »IUnknown« vs. `{00000000-0000-0000-C000-000000000046}`.

LE 28
46

3.8.3 COM *(Component Object Model)*

- Vererbung von Schnittstellen
  - COM unterstützt die Schnittstellenvererbung
- Schnittstelle `IUnknown`
  - Es gibt eine von COM vorgegebene Schnittstelle, die jede COM-Komponente implementieren muss
    - Schnittstelle `IUnknown`
    - *Alle* benutzerdefinierten Schnittstellen müssen von `IUnknown` erben
    - `IUnkown` enthält die Operationen `QueryInterface`, `AddRef` und `Release`



LE 28
47

3.8.3 COM *(Component Object Model)*

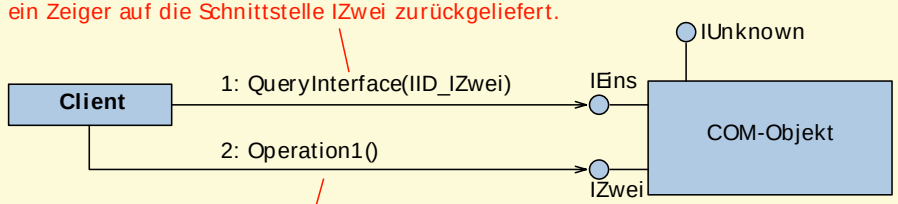
- `QueryInterface`
  - Erlaubt es, einen Zeiger auf eine der Schnittstellen einer Komponente anzufordern
  - Dies geschieht, indem `QueryInterface` die IID einer Schnittstelle als Eingabe übergeben bekommt
  - Alle Standard-Schnittstellen sind von Microsoft dokumentiert
  - Die IID der gewünschten Schnittstelle entnimmt der Entwickler vor dem Aufruf von `QueryInterface` der zugehörigen Dokumentation
  - Als Ergebnis liefert `QueryInterface` im Erfolgsfall, einen gültigen Zeiger auf die angeforderte Schnittstelle
  - Im Fehlerfall wird eine Fehlercode zurückgegeben.

LE 28
48

3.8.3 COM *(Component Object Model)*

- Gebrauch von `IUnknown::QueryInterface`

Client benutzt den Zeiger auf die Schnittstelle IEins, um mit Hilfe von QueryInterface(IID_IZwei) einen Zeiger auf die Schnittstelle IZwei anzufordern. Als Ergebnis wird ein Zeiger auf die Schnittstelle IZwei zurückgeliefert.



Der *Client* besitzt nun einen Zeiger auf die Schnittstelle IZwei. Er kann nun die Operation Operation1 von IZwei aufrufen.

LE 28
49

3.8.3 COM *(Component Object Model)*

- **Lebenszyklus eines COM-Objekts**
  - Über eine Referenzzählung *(reference counting)*
    - Jedes COM-Objekt verwaltet einen Zähler, der mit Hilfe von `AddRef` erhöht wird, wenn ein *Client* einen Zeiger auf eine Schnittstelle der Komponente anfordert
    - Benötigt ein *Client* den Zeiger nicht mehr, dann dekrementiert er den Referenzzähler mit der Operation `Release`
    - Ist Referenzzähler = 0, dann sind keine Schnittstellenzeiger mehr auf das COM-Objekt gerichtet und das COM-Objekt vernichtet sich selbst.

LE 28
50

3.8.3 COM *(Component Object Model)*

- **Module**
  - Auslieferung als DLLs oder Anwendungen (.exe)
  - Jedes Modul: ein oder mehrere COM-Komponenten
- **Komponente in einer DLL**
  - Die DLL wird in den Prozessraum des aufrufenden *Clients* geladen
    - Deshalb auch prozessinterner *Server* genannt
    - Vorteil: Schnelle Kommunikation
- **Komponente in einer Anwendung**
  - Als lokaler *Server (local server)* bezeichnet
    - Das Modul ist eine Anwendung (.exe-Datei)
    - Nachteil: langsamer Zugriff
    - Vorteil: Laufen als eigenständige Anwendung.

LE 28
51

3.8.3 COM *(Component Object Model)*

- **COM-Komponenten**
  - Stück Binärcode
  - Kann eine oder mehrere Schnittstellen implementieren
  - Können nicht voneinander erben
  - Nur Schnittstellen-Vererbung
  - Es kann nur über ihre Schnittstellen kommuniziert werden
  - Ein *Client* kann einen Zeiger auf alle Schnittstellen, die eine Komponente implementiert, mit Hilfe der Operation `QueryInterface` und der entsprechenden IID erhalten
  - Wie erzeugt man nun aber COM-Objekte und wie gelangt man an den ersten Schnittstellenzeiger?

LE 28
52

3.8.3 COM *(Component Object Model)*

- Jede COM-Komponente wird, wie die Schnittstellen, über 2 Bezeichner angesprochen
  - eine GUID
  - einen Namen
- Eine COM-Komponente wird auch als COM-Klasse bezeichnet und die zugehörige GUID als Klassen-Bezeichner *(class identifier)* oder kurz CLSID
- Die CLSID einer Komponente wird zusammen mit dem Aufenthaltsort des Moduls, das sie implementiert, in der *Windows Registry* abgelegt
- →Erzeugen eines COM-Objekts.

LE 28
53

COM und die *Windows Registry*

- Eine COM-Komponente muss registriert werden, bevor sie genutzt werden kann
- Im Zweig `HKEY_CLASSES_ROOT/CLSID` hinterlegt jede Komponente als Schlüssel ihre CLSID
  - Dieser Schlüssel enthält als Wert den Namen der Komponente
- Ein Unterschlüssel enthält den Dateinamen des Moduls, welches die Komponente implementiert
  - Der Name des Schlüssels ist
    - `InprocServer32` bei DLL-Modulen
    - `LocalServer32` bei Exe-Modulen.

LE 28
54

3.8.3 COM *(Component Object Model)*

- Erzeugen eines COM-Objekts
  - Die CLSID der zugehörigen COM-Komponente muss bekannt sein
  - Zusätzlich benötigt man die IID der Schnittstelle, die die Komponente implementiert und auf die man einen Zeiger zurückerhalten möchte
  - Mit beiden Informationen kann man die Operation `CoCreateInstance` der COM-Bibliothek aufrufen
  - `CoCreateInstance` sucht zunächst nach der CLSID der Komponente in der *Registry*
  - Ist der entsprechende Eintrag gefunden, so wird der zugeordnete Aufenthaltsort des Moduls der Komponente bestimmt, das Modul geladen und die Komponente erzeugt.

LE 28
55

3.8.3 COM *(Component Object Model)*

- Nachdem die Komponente erzeugt wurde, gibt `CoCreateInstance` eine Zeiger auf die angeforderte Schnittstelle zurück und terminiert
- Besitzt man einmal einen Zeiger auf eine Schnittstelle, so können alle Operationen der Schnittstelle, wie bei einem normalen Objekt aufgerufen werden

LE 28
56

3.8.3 COM *(Component Object Model)*

- IDL *(interface definition language)*
  - Sollen Komponenten über Prozessgrenzen hinweg kommunizieren *(Local Server)*, dann kann die Kommunikation *nicht* mehr direkt erfolgen
    - *Client* und Komponente laufen in eigenen Adressräumen
  - Stellvertreter und Stummel
    - Kommunikation zwischen *Client* und *Server* erfolgt über Stellvertreter *(proxies)* und Stummel *(stubs)*
      - Hinweis: Außerhalb des COM-Kontextes werden Stellvertreter als Stummel *(stubs)* und Stummel als Skelette *(skeletons)* bezeichnet.

LE 28
57

3.8.3 COM *(Component Object Model)*

- **Stellvertreter** laufen im Adressraum des *Clients*
  - Vertreten die eigentliche Komponente
- **Stummel** laufen im Adressraum des *Servers*
  - Stellen den Kontakt zwischen Stellvertreter und eigentlicher Komponente her
- Stellvertreter und Stummel werden von einem **Werkzeug generiert**
- Werkzeug benötigt dazu eine Beschreibung der Komponente und vor allem ihrer Schnittstellen
- Diese Beschreibung liefert die **IDL**
- Hat man eine Schnittstelle oder Komponenteninformationen in der **IDL** formuliert, dann kann man anschließend mit dem *Microsoft IDL-Compiler* (MIDL-Compiler) Stellvertreter und Stummel automatisch erzeugen.

LE 28
58

3.8.3 COM *(Component Object Model)*

- Beispiel
  - Die IDL-Deklaration von `IUnknown` sieht folgendermaßen aus:

```
[ local,
  object,
  uuid(00000000-0000-0000-C000-000000000046),
  pointer_default(unique)
]
interface IUnknown
{ typedef [unique] IUnknown *LPUNKNOWN;
  HRESULT QueryInterface([in] REFIID riid,
        [out, iid_is(riid)] void **ppvObject);
    ULONG AddRef();
    ULONG Release();
}.
```


LE 28
59

3.8.3 COM *(Component Object Model)*

- Typbibliotheken
  - In COM-IDL kann angegeben werden, ob eine Typbibliothek für die Komponente erzeugt werden soll
  - **Introspektion**
    - Typbibliotheken stellen den COM-Weg dar, um Introspektion zu ermöglichen
    - Mit Hilfe einer **CLSID** kann ein *Client* eine Anfrage nach Typinformationen an die *Registry* stellen
    - Existiert eine Typbibliothek, wird ein Zeiger auf die Schnittstelle `ITypeLib` zurückgegeben, die es ermöglicht, auf die in der Typbibliothek enthaltenen Typinformationen zuzugreifen.

LE 28
60

3.8.3 COM *(Component Object Model)*

- Zu jeder Komponente und jeder Schnittstelle kann zusätzlich über die `ITypeLib` Schnittstelle ein Zeiger auf die `ITypeInfo` Schnittstelle angefordert werden, um Metainformationen zu erhalten
- Hat man in einer IDL-Datei über das IDL-Schlüsselwort `library` spezifiziert, dass eine Typbibliothek erstellt werden soll, so kann wieder der MIDL-Compiler benutzt werden, um die Typbibliothek aus der IDL-Datei zu erzeugen

LE 28
61

3.8.4 *ActiveX* und OLE

- Automatisierung
  - »Fernsteuerung« von Anwendungen
  - Problem
    - Will man zur Laufzeit auf eine COM-Komponente aus einem Skript/einer Anwendung heraus direkt über ihre Schnittstellen zugreifen, so ist dies i. Allg. nicht möglich
      - Skript-Interpreter/Anwendung müssten hierfür alle potentiellen Schnittstellen der dynamisch zu ladenden COM-Komponenten im Voraus (zur Übersetzungszeit des Interpreters/der Anwendung), kennen
    - Skript hat i. Allg. nur folgende Informationen:
      - Der Bezeichner der Komponente
      - der Name der aufzurufenden Operation
      - ihre Parameter.

LE 28
62

3.8.4 *ActiveX* und OLE

- Schnittstelle `IDispatch`
  - Deklariert Operation `Invoke`, die den Namen und die Parameter einer Operation übergeben bekommt
  - `Invoke` bekommt keinen Operationsnamen übergeben, sondern eine *dispatch*-ID (DISPID)
    - Jeder Operation der Komponente wird eine Nummer zugeordnet
  - Operation `GetIDsOfNames` von `IDispatch` erlaubt es, zu einem Operationsnamen die zugehörige DISPID herauszufinden
  - Mit der DISPID und den Parametern kann nun die Operation mit Hilfe von `Invoke` aufgerufen werden
  - Eine Komponente, die die `IDispatch`-Schnittstelle implementiert, heisst Automations-*Server.*

LE 28
63

3.8.4 *ActiveX* und OLE

- Beispiel
  - Über Automatisierung soll das CASE-Werkzeug *Rational Rose* angesprochen werden
  - Es wird hierfür *Visual Basic* benutzt
  - *Visual Basic* gestattet einen einfachen Zugriff auf Automations-*Server*
  - Der Zugriff über die Schnittstelle `IDispatch` erfolgt intern, für den Programmierer unsichtbar

```
'Referenz auf ein Rational Rose-Anwendungsobjekt
Dim RoseApp As RoseApplication
'Anwendungsobjekt besorgen
Set RoseApp = GetObject(, "Rose.Application")
'Operation der Anwendung aufrufen
Set theModel = RoseApp.CurrentModel.
```


LE 28
64

3.8.4 *ActiveX* und OLE

- Vor- und Nachteile
  - \+ Einfacher Zugriffsmechanismus
  - – Die Automatisierungsschnittstelle erlaubt einen untypisierten Aufruf von Operationen
    - Die Typen müssen daher zur Laufzeit überprüft werden, was Zeit kostet
    - Zusätzlich ist die Fehlersuche schwieriger, da der Entwickler vom Compiler nicht bei der Typüberprüfung unterstützt wird.

LE 28
65

3.8.4 *ActiveX* und OLE

- OLE
  - Anforderung
    - In dokumentenbasierten Anwendungen besteht der Wunsch, Bilder, Tabellen usw. per »*drag and drop*« in andere Dokumente zu übernehmen
  - OLE-Architektur
    - Dokument-*Server*
      - Kann Inhalte verwalten und darstellen
    - Dokument-Container
      - Kann Dokumenten-*Server* aufnehmen
    - Eingebettete Dokumente sollen dort bearbeitet werden, wo sie im umgebenen Dokument dargestellt werden *(in-place activation).*

LE 28
66

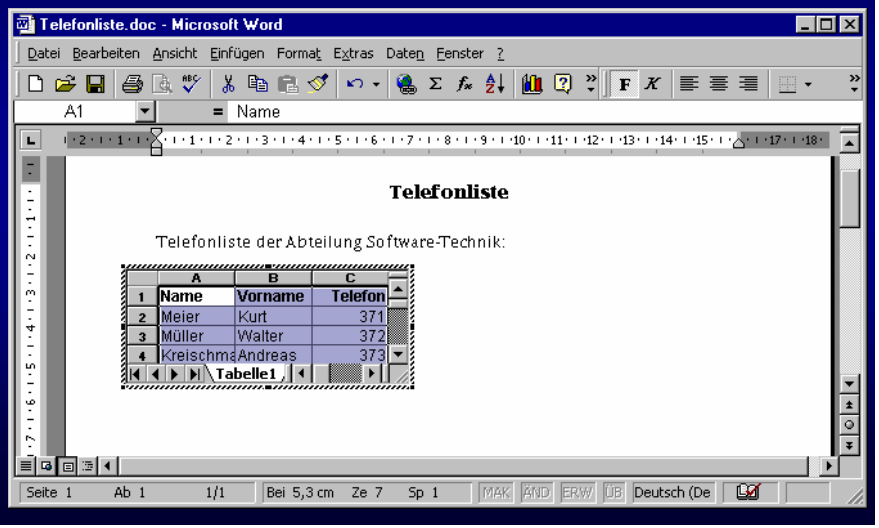
3.8.4 *ActiveX* und OLE

- Wird einem eingebetteten Dokumenten-*Server* mitgeteilt, dass der dargestellte Inhalt verändert werden soll, so wird – unsichtbar für den Benutzer – vom Dokument-*Server* über dem eigentlichen Inhalt ein neues Fenster geöffnet, in dem der Inhalt verändert werden kann
- Es entsteht die Illusion, dass der Inhalt des eingebetteten Dokuments direkt im umgebenden Dokument verändert werden kann
- Zusätzlich wird der Inhalt und/oder die Semantik einiger Menüs und Werkzeugleisten ausgetauscht.

LE 28
67

3.8.4 *ActiveX* und OLE

- Die in das *Word*-Dokument eingebettete *Excel*-Tabelle ist aktiviert



LE 28
68

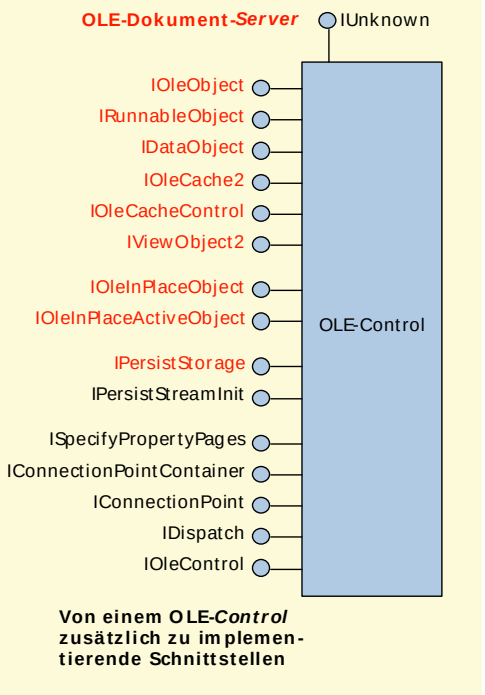
3.8.4 *ActiveX* und OLE

- Verweise *(linking)*
  - Nur ein Verweis auf das eigentliche Dokument wird in dem Dokument-Container gespeichert
  - Die Daten des Dokuments, auf das verwiesen wird, werden in einer externen Datei gespeichert
  - \+ Derselbe Inhalt kann in mehrere Dokumente gleichzeitig integriert werden
  - \+ Änderungen am Dokument bleiben dadurch in allen Dokumenten konsistent
  - – Verweis wird ungültig, wenn das externe Dokument gelöscht oder verschoben wird
  - – Keine *in-place activation*
    - Es wird ein separates Fenster geöffnet.

LE 28
69

3.8.4 *ActiveX* und OL

- OCXs
  - OLE-Steuerelemente
  - Normale OLE-Dokument-*Server*, ergänzt um einige Schnittstellen zur Ereignis-verarbeitung



LE 28
70

3.8.4 *ActiveX* und OLE

- *ActiveX*-Steuerelemente
  - Nachteil der OCXe
    - Durch die große Anzahl der zu implementierenden Schnittstellen werden OCXe relativ groß
    - Dies ist nachteilig, wenn Komponenten auf Web-Seiten eingesetzt werden sollen
      - Die Komponenten müssen über das Internet auf das *Client*-Computersystem geladen werden
      - Je größer die Komponente, desto länger dauert dieser Vorgang
    - Daher wurden *ActiveX*-Steuerelemente spezifiziert, die außer `IUnknown` keine weitere Schnittstelle implementieren.

LE 28
71

3.8.4 *ActiveX* und OLE

- Selbstregistrierung
  - Jede COM-Komponente ist ein *ActiveX*-Steuerelement, wenn sie zusätzlich noch die Fähigkeit zur Selbstregistrierung unterstützt
  - Wird eine Komponente auf einer Web-Seite platziert, dann soll ihre Installation automatisch und dauerhaft erfolgen
  - Alle COM-Module, die *ActiveX*-Steuerelemente enthalten, müssen die Operationen `DllRegisterServer` und `DllUnRegisterServer` zur Verfügung stellen, die das Steuerelement in der *Registry* an- und abmelden
  - *ActiveX*-Steuerelemente werden auf dem *Client* dauerhaft installiert (im Gegensatz zu Java).

LE 28
72

3.8.4 *ActiveX* und OLE

- Auslieferung & Benutzung
  - Verpackungsalternativen
    - Eine einzige ausführbare Datei mit Endungen wie `.dll` oder `.ocx`
    - Eine Archiv-Datei *(cabinet file)*
      - Alle zum Steuerelement gehörenden Dateien und Ressourcen werden zusammen mit einer die Installationsanweisungen enthaltenden `.INF`-Datei in ein Archiv mit der Endung `.cab` verpackt
    - Eine einzige `.INF`-Datei
      - Enthält neben Installationsanweisungen nur Verweise auf das Steuerelement und alle seine Dateien.

LE 28
73

3.8.4 *ActiveX* und OLE

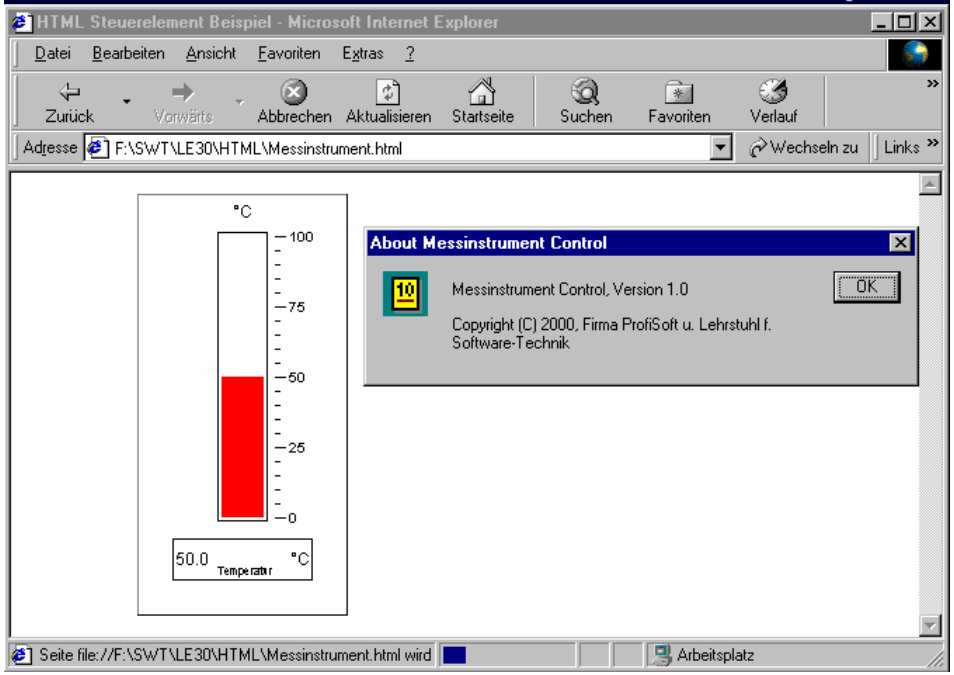
- **Automatische Installation**
  - Nach dem Herunterladen kann das *ActiveX*-Steuerelement automatisch installiert und deinstalliert werden
  - Ist ein ActiveX-Steuerelement korrekt installiert und damit auch registriert, so kann das Steuerelement in den verschiedensten Containern verwendet werden.

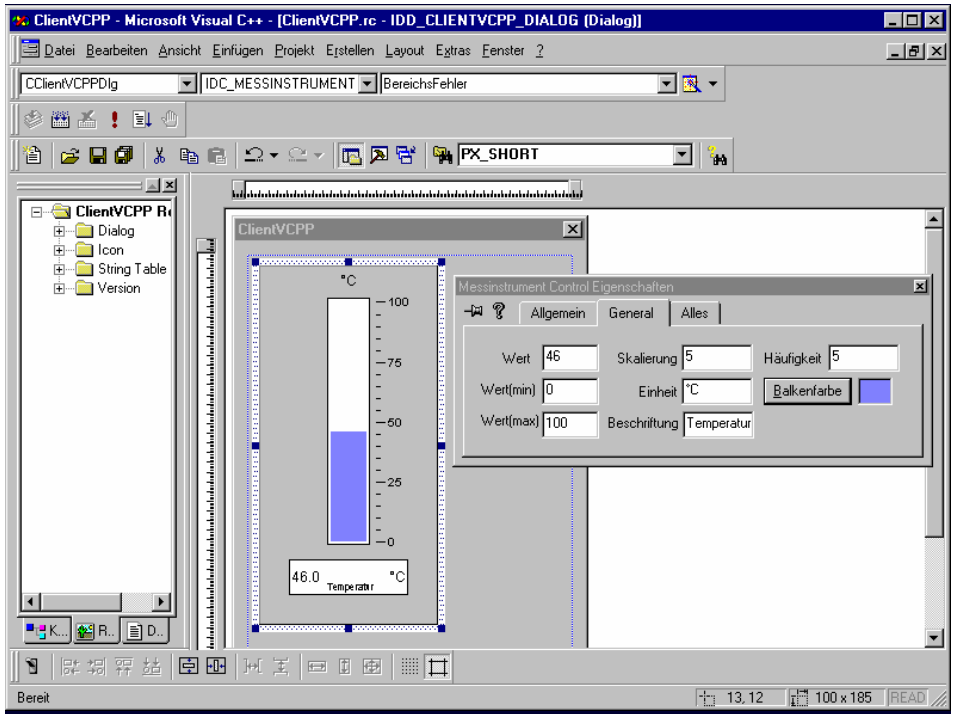
LE 28
74

3.8.4 *ActiveX* und OLE

- **Beispiel**
  - Messinstrument in einer Web-Seite

```
<HTML>
<TITLE>HTML Steuerelement Beispiel</TITLE>
<BODY>
<OBJECT CLASSID="clsid:BCCF0DF5-5D46-11D4-8C34-
   0000E8781BB8"
   ID=Messinstrument HEIGHT=300 WIDTH=150 HSPACE=80>
<PARAM NAME="Einheit" VALUE="°C">
<PARAM NAME="MaxWert" VALUE="100">
<PARAM NAME="Wert" VALUE="50">
<PARAM NAME="Beschriftung" VALUE="Temperatur">
</OBJECT> <SCRIPT LANGUAGE=VBScript>
Messinstrument.AboutBox()</SCRIPT>
</BODY> </HTML>.
```



- Danke!
- Aufgaben

- Diese Präsentation bzw. Teile dieser Präsentation enthalten Inhalte und Grafiken des Lehrbuchs der Software-Technik (Band 1), 2. Auflage von Helmut Balzert, Spektrum Akademischer Verlag, Heidelberg 2001