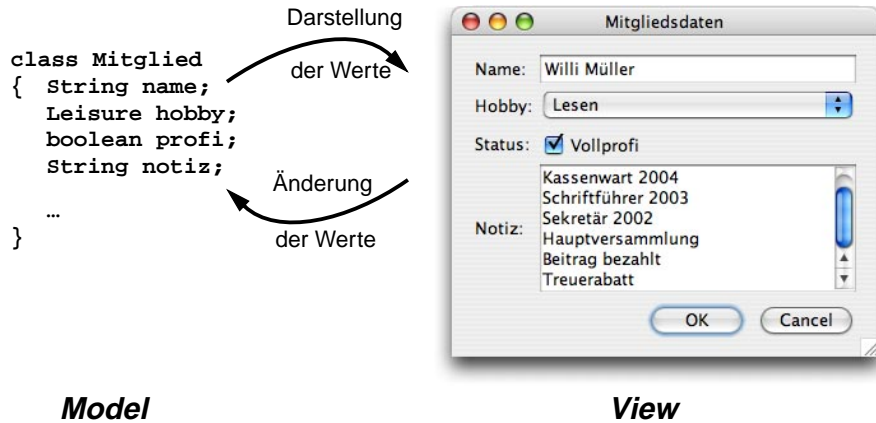


8. Model/View-Paradigma für Komponenten

GP-113



Model

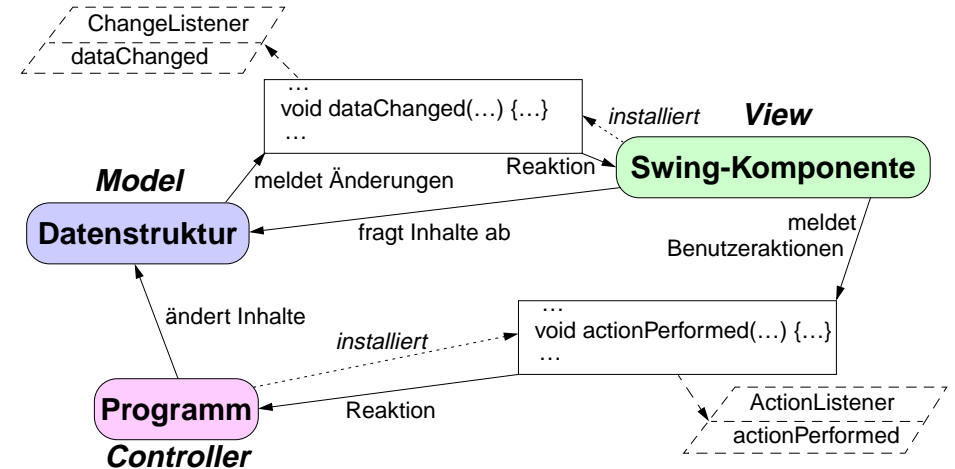
View

Wechselwirkung:

- Komponenten der Benutzungsoberfläche (*view*) stellen Inhalte von Datenstrukturen (*model*) des Programms dar
- **Ziel:** Übereinstimmung zwischen Daten und ihrer Darstellung am Bildschirm **automatisch** sicherstellen

Zusammenarbeit von Model, View und Programm

GP-114



- Anwendung des "Observer"-Prinzips (wie bei der Ereignisbehandlung): eine (oder mehrere) Komponenten beobachten die Datenstruktur
- Datenstruktur ruft bei jeder Änderung zugehörige Methode in jedem Beobachter-Objekt auf

Schematischer Aufbau von Model und View

GP-115

```

class Model
{
    // Datenstruktur mit Zugriffsmethoden
    private String data;
    String getData ()
    { return data; }
    void String setData (String s)
    { data = s;
      fireChangeEvent();
    }
    // Verwaltung der Listener
    ChangeListener[] listener;
    int count = 0;
    void addChangeListener
    (ChangeListener cl)
    { listener[count++] = cl; }
    void fireChangeEvent ()
    { for (int i=0; i<count; i++)
      listener[i].dataChanged();
    }
}
    
```

```

class View extends JComponent
{
    private Model model;
    View (Model m)
    { model = m;
      model.addChangeListener(
        new RedrawListener());
    }
    ...
    class RedrawListener
    implements ChangeListener
    {
        public void dataChanged()
        { String s =
          model.getData();
          ...
          repaint();
        }
    }
}
interface ChangeListener
{
    public void dataChanged();
}
    
```

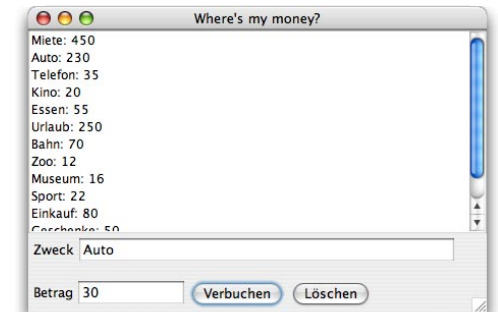
Beispiel: Haushaltsbuch

GP-116

Aufgabe: Programm zur Überwachung von Ausgaben im Haushalt entwerfen

Eigenschaften:

- speichert Geldbetrag (Summe der Ausgaben) für jede Kategorie
- Eingabe von Einzelbuchungen (+/-) und Löschen ganzer Kategorien
- Liste aller Kategorien mit Beträgen wird angezeigt



Entwicklungsschritte:

- Datenstruktur entwerfen
- Datenstruktur zum Datenmodell (*Model*) für eine Liste (*JList*) erweitern
- Komponenten strukturieren
- Komponenten generieren und anordnen
- Ereignisbehandlung entwerfen und implementieren

Datenstruktur für Haushaltsbuch

Speicherung der Kategorien und Beträge in zwei parallel belegten Arrays;
zentrale Operation "Buchung vornehmen" und Abfragemethoden:

```
class ChequeBook
{
    private String[] purposes = new String[20]; // Namen der Kategorien
    private int[] amounts = new int[20]; // Geldbeträge dazu
    private int entries = 0; // Anzahl Einträge

    void addTransaction (String p, int a)
    {
        if (p.length() == 0 || a == 0) return; // keine sinnvolle Buchung
        for (int i = 0; i < entries; i++)
        {
            if (purposes[i].equals(p)) // Kategorie bereits vorhanden
            {
                amounts[i] += a; return; // Geldbetrag anpassen
            }
        }
        int index = entries; entries += 1; // neue Kategorie anfügen
        purposes[index] = p; amounts[index] = a;
    }

    int getEntries () // Abfragemethoden: Anzahl Einträge
    {
        return entries;
    }

    int getAmount (int index) // Geldbetrag eines Eintrags
    {
        return amounts[index];
    }

    String getPurpose (int index) // Kategorienname eines Eintrags
    {
        return purposes[index];
    }
}
```

Model für JList-Komponenten

Das `Interface ListModel` im Package `javax.swing` beschreibt die Anforderungen an das Datenmodell für eine graphisch dargestellte Liste (`JList`-Objekt):

```
public interface ListModel
{
    // Verwaltung von Swing-Komponenten, die das Model beobachten:
    void addListDataListener (ListDataListener ldl);
    void removeListDataListener (ListDataListener ldl);
    int getSize (); // Abfrage der Daten durch das JList-Objekt
    Object getElementAt (int index); // Abfrage eines Eintrags
}
```

Die **abstrakte Klasse AbstractListModel** implementiert das An- und Abmelden von `ListDataListener`-Objekten und ergänzt Methoden, um diese zu benachrichtigen:

```
public abstract class AbstractListModel implements ListModel
{
    public void addListDataListener (ListDataListener ldl) { ... }
    public void removeListDataListener (ListDataListener ldl) { ... }
    protected void fireContentsChanged (Object src, int from, int to)
    { ... } // ruft Methode contentsChanged() in allen Listenern auf
    protected void fireIntervalAdded (...) { ... }
    protected void fireIntervalRemoved (...) { ... }
}
```

Die `fireXXX`-Methoden korrespondieren mit den Methoden im `Interface ListDataListener`: `contentsChanged(ListDataEvent e)`, `intervalAdded(...)`, `intervalRemoved(...)`.

Model für Haushaltsbuch

Erweitern die Datenstruktur zum Datenmodell für eine Liste:

- **Abfragemethoden** für `JList`-Objekt hinzufügen.
- **Benachrichtigung** der Listener in den Operationen ergänzen, die die Datenstruktur verändern.

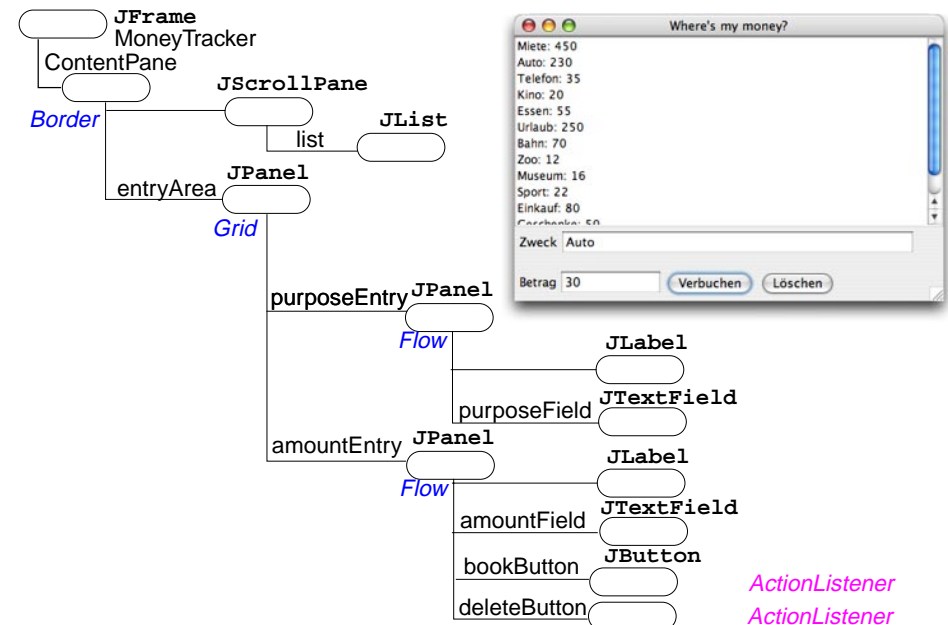
```
class ChequeBook extends AbstractListModel // neue Oberklasse
{
    private ...; // Objektvariablen bleiben unverändert

    public int getSize ()
    {
        return entries;
    }

    public Object getElementAt (int index)
    {
        return purposes[index] + ": " + String.valueOf(amounts[index]);
    }

    void addTransaction (String p, int a)
    {
        if (p.length() == 0 || a == 0) return; // keine sinnvolle Buchung
        for (int i = 0; i < entries; i++)
        {
            if (purposes[i].equals(p)) // Kategorie bereits vorhanden
            {
                amounts[i] += a; // Geldbetrag anpassen
                fireContentsChanged(this, i, i); // Beobachter informieren
                return;
            }
        }
        int index = entries; entries += 1; // neue Kategorie anfügen
        purposes[index] = p; amounts[index] = a;
        fireIntervalAdded(this, index, index); // Beobachter informieren
    }
    // ... } // restliche Abfragemethoden bleiben unverändert
}
```

Objektbaum zum Haushaltsbuch



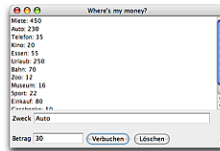
Programm zum Haushaltsbuch

```
class MoneyTracker extends JFrame
{
    private ChequeBook listData; private JList list;
    private JTextField purposeField, amountField;

    MoneyTracker (String title, ChequeBook data)
    {
        super(title); listData = data;
        list = new JList(listData); // Liste erzeugen und mit Model verbinden
        list.setSelectionMode(ListSelectionModel.SINGLE_SELECTION);

        JPanel purposeEntry = new JPanel(new FlowLayout(...));
        purposeEntry.add(new JLabel("Zweck"));
        purposeField = new JTextField(30);
        purposeEntry.add(purposeField);
        JPanel amountEntry = new JPanel(...); // Komponenten einfügen
        JPanel entryArea = new JPanel(new GridLayout(2, 1));
        entryArea.add(purposeEntry); entryArea.add(amountEntry);
        Container content = getContentPane();
        content.setLayout(new BorderLayout());
        content.add(new JScrollPane(list), BorderLayout.CENTER);
        content.add(entryArea, BorderLayout.SOUTH);
        ... setVisible(true); // Eigenschaften des Fensters einstellen
    }

    public static void main (String[] args)
    {
        ChequeBook myMoney = new ChequeBook();
        JFrame f = new MoneyTracker("Where's my money?", myMoney);
    }
}
```



Implementierung der Ereignisbehandlung

Methoden aufrufen, die Änderungen am Datenmodell vornehmen:

```
class MoneyTracker extends JFrame
{
    private ChequeBook listData; private JList list;
    private JTextField purposeField, amountField;

    MoneyTracker (String title, ChequeBook data)
    {
        super(title); listData = data;
        list = new JList(listData); // Liste erzeugen und mit Model verbinden
        ...
        JPanel amountEntry = new JPanel(...); // Komponenten einfügen
        JButton bookButton = new JButton("Verbuchen");
        amountEntry.add(bookButton); ...
        bookButton.addActionListener(new ActionListener()
        {
            public void actionPerformed (ActionEvent e) // Reaktion auf Klick
            {
                String purpose = purposeField.getText(); // Werte auslesen
                String amount = amountField.getText();
                int money;
                try
                {
                    money = Integer.parseInt(amount); // Geldbetrag in Zahl wandeln
                } catch (NumberFormatException nfe)
                {
                    money = 0; // keine Zahl: 0 annehmen
                }
                listData.addTransaction(purpose, money); // Modell verändern
            }
        });
        ... setVisible(true); // Rest des Fensters aufbauen
    }
}
```



Anpassung der Darstellung durch Renderer-Objekte

```
class MoneyTracker extends JFrame
{
    ...
    MoneyTracker (String title, ChequeBook data, ListCellRenderer renderer)
    {
        super(title); listData = data;
        list = new JList(listData); // Liste erzeugen und mit Model verbinden
        list.setSelectionMode(ListSelectionModel.SINGLE_SELECTION);
        list.setCellRenderer(renderer); // eigenes Renderer-Objekt anschliessen
    }

    public static void main (String[] args)
    {
        ChequeBook myMoney = new ChequeBook();
        JFrame f = new MoneyTracker("Where's my money?", myMoney,
            new BalanceRenderer(myMoney));
    }
}

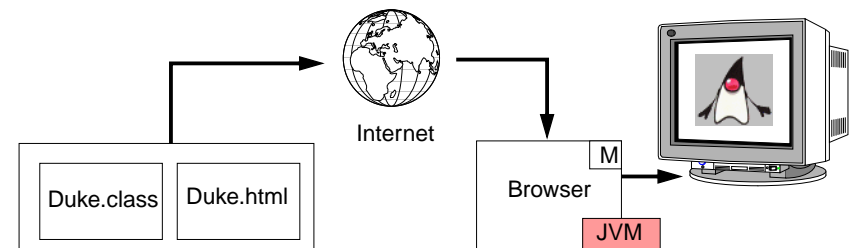
class BalanceRenderer implements ListCellRenderer
{
    private ChequeBook entries;
    BalanceRenderer (ChequeBook data) { entries = data; }
    public Component getListCellRendererComponent (JList list, Object value,
        int index, boolean isSelected, boolean cellHasFocus)
    {
        JLabel lab = new JLabel("*** " + value.toString());
        if (isSelected) { ... } else { ... }
        int amount = entries.getAmount(index);
        if (amount < 0) lab.setForeground(Color.green);
        else if (amount > 500) lab.setForeground(Color.red);
        ... lab.setOpaque(true); return lab;
    }
}
```



9. Applets

Applet (small application):

- kleines Anwendungsprogramm in Java für eine spezielle Aufgabe,
- an eine WWW-Seite (world wide web) gekoppelt;
- das Applet wird mit der WWW-Seite über das Internet übertragen;
- der Internet Browser (Werkzeug zum Navigieren und Anzeigen von Informationen) arbeitet mit einer Java Virtual Machine (JVM) zusammen; sie führt eintreffende Applets aus.



Programm (Java-Applet) wird übertragen, läuft beim Empfänger, bewirkt dort Effekte.

Applets benutzen JVM und Java-Bibliotheken des Empfängers.

Programmierung von Applets

Applets werden wie Programme mit Swing-Benutzungsoberfläche geschrieben, aber:

- Die Hauptklasse ist Unterklasse von `JApplet` statt von `JFrame`.
- Es gibt die Methode `main` nicht.
- `System.exit` darf nicht aufgerufen werden.
- Die Methode `public void init ()` tritt an die Stelle des Konstruktors.
- Ein- und Ausgabe mit Swing-Komponenten statt mit Dateien.

Programmschema: Fenster mit Zeichenfläche (GP-88):

```
class WarningApplet extends JApplet
{
    ...
    public void paint (Graphics g)
    { ... auf g schreiben und zeichnen ... }
}
```

Programmschema: Bedienung mit Swing-Komponenten (GP-103):

```
class TrafficLight extends JApplet
{
    public void init ()
    { ... Objektbaum mit Swing-Komponenten aufbauen ... }
    ...
}
```

Applet-Methoden

Die Klasse `JApplet` liegt in der Hierarchie der graphischen Komponenten:

```
java.awt.Component
  java.awt.Container
    java.awt.Panel
      java.awt.Applet
        javax.swing.JApplet
```

`JApplet` definiert Methoden ohne Effekt zum Überschreiben in Unterklassen:

```
void init ()           wird aufgerufen, wenn das Applet ...
                       geladen wird
void start ()          seine Ausführung (wieder-)beginnen soll
void stop ()           seine Ausführung unterbrechen soll
void destroy ()        das Applet beendet wird
```

Methoden zum Aufruf in Unterklassen von `JApplet`, z. B.

```
void showStatus (String)   Text in der Statuszeile des Browsers anzeigen
String getParameter (String) Wert aus HTML-Seite lesen
```

Weitere Methoden aus Oberklassen, z. B.

```
void paint (Graphics g)    auf g schreiben und zeichnen (aus Container)
```

Applets zu HTML-Seiten

HTML (Hypertext Markup Language):

- Sprache zur Beschreibung formatierter, strukturierter Texte und deren Gestaltung
- Standardsprache zur Beschreibung von Internet-Seiten
- Einfache Sprachstruktur: Marken `<hr>` und Klammern ` ... `, ` ... ` mit bestimmter Bedeutung, z. B.

```
Wir unterscheiden
<ul>
  <li>diesen Fall,</li>
  <li>jenen Fall</li>
</ul>
und viele andere Fälle.
<hr>
```

```
Wir unterscheiden
• diesen Fall,
• jenen Fall
und viele andere Fälle.
```

Ein Applet wird auf einer HTML-Seite aufgerufen, z. B.

```
<title>Catch the Duke!</title>
<hr>
<applet code="CatchM.class" width="300" height="300">
</applet>
```

Beim Anzeigen der HTML-Seite im Browser oder Appletviewer wird das Applet auf einer Fläche der angegebenen Größe ausgeführt.

Java-Programm in Applet umsetzen

Ein Java-Programm kann man in folgenden Schritten in ein Applet transformieren:

1. Alle Datei-E/A in Benutzung von Swing-Komponenten umsetzen, z. B. `System.out.println(...)` in `g.drawString(...)` in der Methode `paint`.
2. Programm nicht anhalten, `System.exit`-Aufrufe und `close`-Button bzw. Aufrufe von `setDefaultCloseOperation` entfernen.
3. Layoutmanager ggf. explizit wählen, Vorgabe ist `BorderLayout` (wie bei `JFrame`).
4. `javax.swing.JApplet` importieren, Hauptklasse als Unterklasse von `JApplet` definieren
5. Konstruktor durch `init`-Methode ersetzen; darin wird der Objektbaum der Komponenten aufgebaut.
6. `main`-Methode entfernen; die Hauptklasse des Java-Programms entfällt einfach, wenn sie nur die `main`-Methode enthält und darin nur das `JFrame`-Objekt erzeugt und platziert wird.
7. HTML-Datei herstellen mit `<applet>`-Element zur Einbindung der `.class`-Datei.
8. Testen des Applets; erst mit dem `appletviewer` dann mit dem Browser.

siehe Beispiel Ampel-Simulation GP-104 bis 108.

Parameter zum Start des Applet

Dem Applet können zum Aufruf von der HTML-Seite Daten mitgegeben werden.

Notation: Paare von Zeichenreihen für Parametername und Parameterwert

```
<param name="Parametername" value="Parameterwert">
```

eingesetzt im Applet-Aufruf:

```
<applet code="CoffeesShop.class" width="600" height="200">
<param name="Columbian" value="12">
<param name="Java" value="15">
<param name="Kenyan" value="9">
</applet>
```

Im Applet auf die Parameterwerte zugreifen:

```
preis = Integer.parseInt (getParameter ("Java"));
```

Sicherheit und Applets

Beim Internet-Surfen kann man nicht verhindern, dass fremde Applets auf dem eigenen Rechner ausgeführt werden. Deshalb sind ihre **Rechte i. a. stark eingeschränkt**:

| Operation | Java Programm | Applet im appletviewer | lokales Applet im Browser | fremdes Applet im Browser |
|-----------------------------------|---------------|------------------------|---------------------------|---------------------------|
| lokale Datei zugreifen | X | X | | |
| lokale Datei löschen | X | | | |
| anderes Programm starten | X | X | | |
| Benutzernamen ermitteln | X | X | X | |
| zum Sender des Applet verbinden | X | X | X | X |
| zu anderem Rechner verbinden | X | X | X | |
| Java-Bibliothek laden | X | X | X | |
| <code>System.exit</code> aufrufen | X | X | | |
| Pop-up Fenster erzeugen | X | X | X | X |

Diese Einstellungen der Rechte können im Browser geändert werden.

Man kann auch **signierten Applets** bestimmter Autoren weitergehende Rechte geben.

Außerdem wird der **Bytecode** jeder Klasse auf Konsistenz **geprüft**, wenn er in die JVM geladen wird.

Applets als Beispiel für Erweiterungsmodule (*plug-ins*)

Erweiterungsmodule (*plug-ins*) sind keine selbständigen Programme, sondern werden in ein anderes Programm (*host program*) eingebettet, um dessen Funktionalität zu erweitern.

Zusammenarbeit zwischen Erweiterungsmodul und Programm basiert (in der Regel) auf:

- fest vorgegebener (abstrakter) Oberklasse für das Erweiterungsmodul
- Methoden, die das Modul unbedingt anbieten muss (Eintrittspunkte)
- Methoden, die das Programm speziell für solche Module bereitstellt (*call backs, services*)
- sonstigen Methoden des Programms, die auch zur Verwendung in Erweiterungsmodulen geeignet sind

