6. Objektorientiertes Design

6.1 Entwurfsmuster

6.2 Zusammenfassendes Beispiel

6.3 Umsetzung des Model-View-Controller-Musters in Java

Grundlage des objektorientierten Designs

Das objektorientierte Design besteht im Wesentlichen aus dem Abbilden des in der OO-Analyse gewonnenen Modells auf **softwaretechnische Klassen**.

Typische Klassen sind

- Klassen, die Hardware- und Software-Ressourcen repräsentieren
- GUI-Klassen, insbesondere für die Erstellung von Fenster-Oberflächen
- Middleware-Klassen
- Abstrakte Klassen
- Interface-Klassen
- Behälterklassen (Container, Collections)
- und viele mehr.

Beispiel für eine Datenbankanwendung

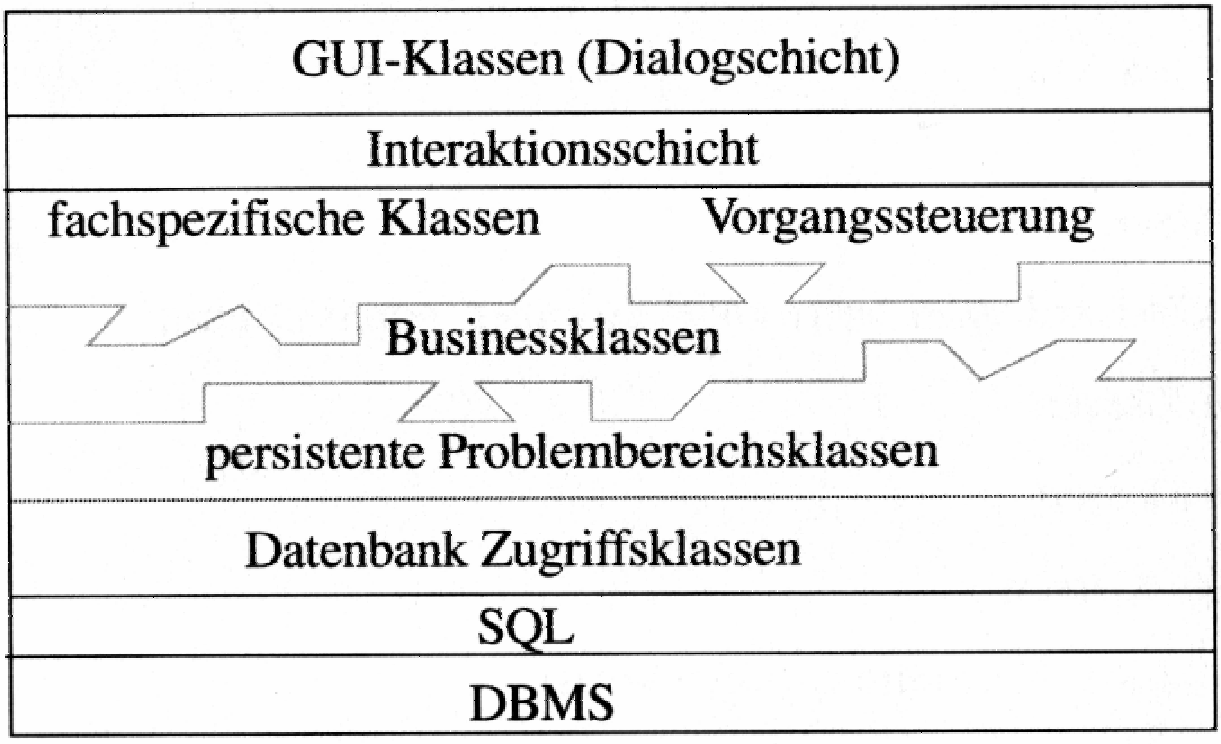


Abbildung 6.1 Architekturmodell

Viele dieser Klassen werden Klassenbibliotheken oder Frameworks entnommen und brauchen nicht selbst implementiert zu werden. Das ist ein wesentlicher Unterschied zum konventionellen Software-Engineering.

6.1 Entwurfsmuster

Neuere Arbeiten zum objektorientierten Design schlagen die Einführung von **Entwurfsmustern** (design patterns) vor. Die Idee ist dabei, dass man in der Softwareentwicklung immer wieder ähnlich geartete Probleme löst.

Entwurfsmuster sind also **verallgemeinerte Problemlösungskonzepte.** Es sind keine Klassenbibliotheken.

Im Zusammenhang mit Entwurfsmustern ist die Trennung zwischen Schnittstelle (Interface) und Implementierung sinnvoll. Eine solche Trennung kann in Java durch abstrakte Klassen und durch Interface-Klassen erfolgen.

Interface-Klassen

Eine Interface-Klasse in Java enthält nur die Deklaration der Methoden, nicht ihre Implementierung.

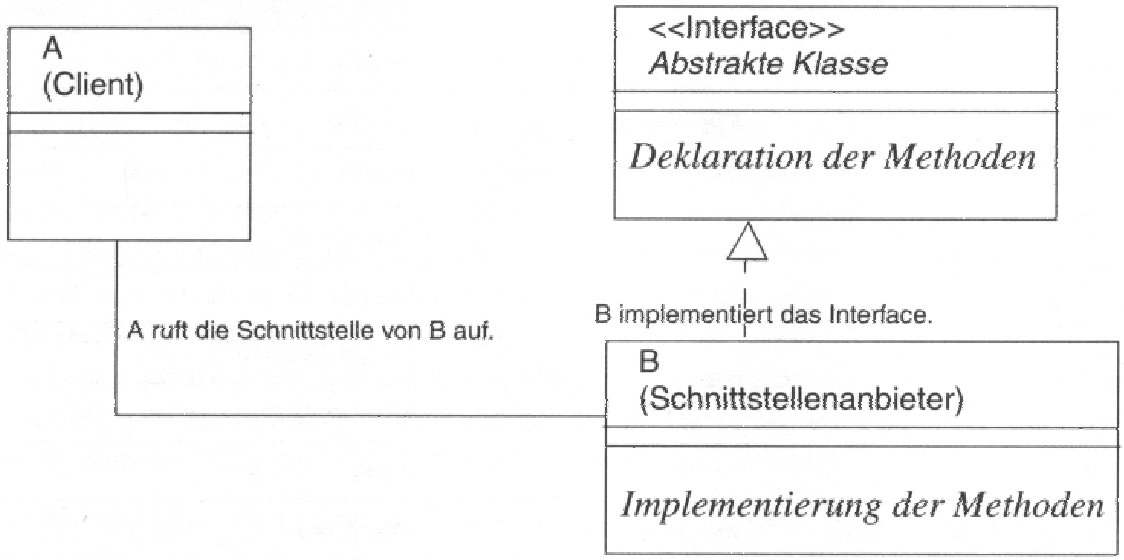


Abbildung 6.2 Trennung des Interface von der Implementierung

Model – View – Controller (1)

Model-View-Controller ist ein wichtiges und in der Praxis oft nützliches Entwurfsmuster.

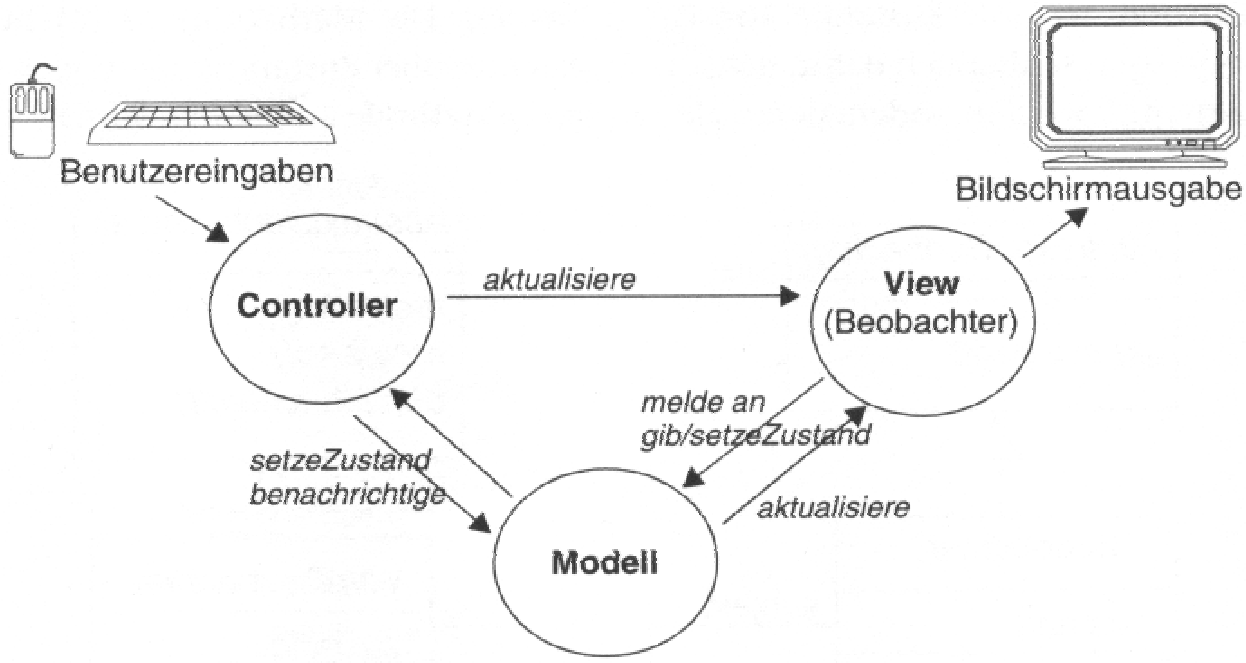


Abbildung 6.3 Modell View Controller

Model – View – Controller (2)

Model

Im Model ist die Anwendungslogik enthalten. Es beschreibt das Systemmodell und sein gesamtes immanentes Verhalten.

View

In der View ist die Repräsentation des Zustandes des Modells und seiner Ausgaben gegenüber dem Benutzer zusammen gefasst.

Controller

Der Controller implementiert die Interaktion mit dem Benutzer, nimmt seine Eingaben entgegen.

Die saubere Trennung dieser drei Komponenten hat den Vorteil, dass man eine Einzelne leicht neu implementieren kann, ohne dass die anderen betroffen sind.

Das Beobachtermuster (1)

Das Beobachtermuster stellt eine Kopplung zwischen Beobachtern (View-Objekten) her, die ein observierbares Objekt beobachten.

Das **beobachtbare Objekt** stellt folgende Methoden zur Verfügung:

- Anmelden
- Abmelden
- Zustand abfragen
- Zustand setzen

Die **Beobachter** haben eine Aktualisierungsmethode, die vom beobachtbaren Objekt aufgerufen wird, wenn sich im beobachteten Objekt etwas ändert.

Das Beobachtermuster (2)

Die Struktur des Beobachter-Musters

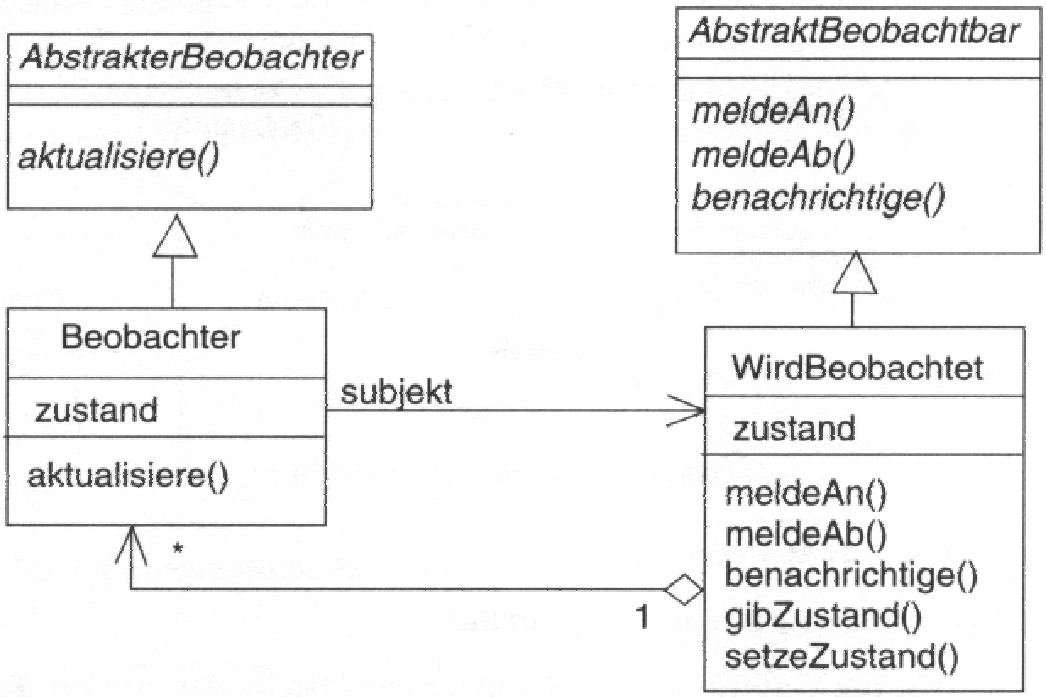


Abbildung 6.4 Beobachtermuster

Interaktionsdiagramm des Beobachtermusters

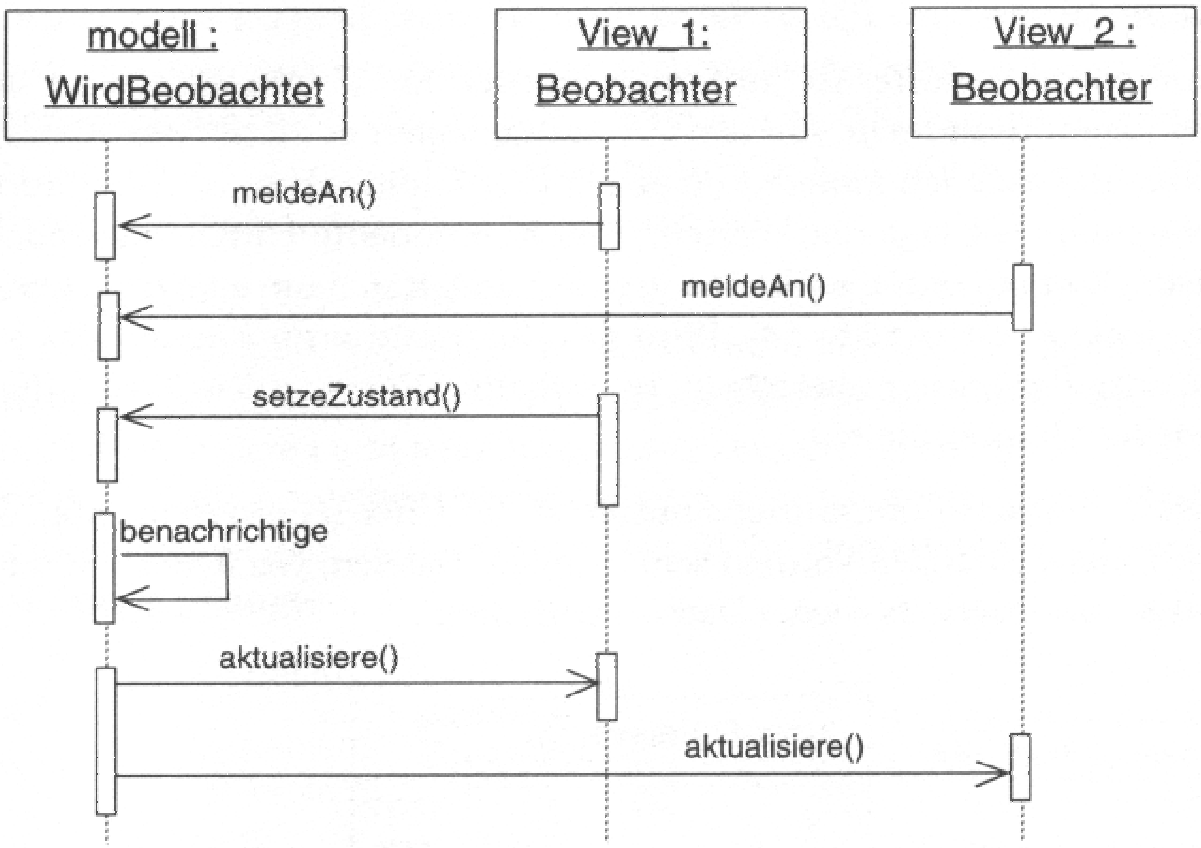


Abbildung 6.5 Interaktionsdiagramm für Beobachtermuster

Das Kompositum

Das Kompositum ist ein Entwurfsmuster, das Rekursion zulässt. Es beschreibt Komponenten, die andere Komponenten enthalten können.

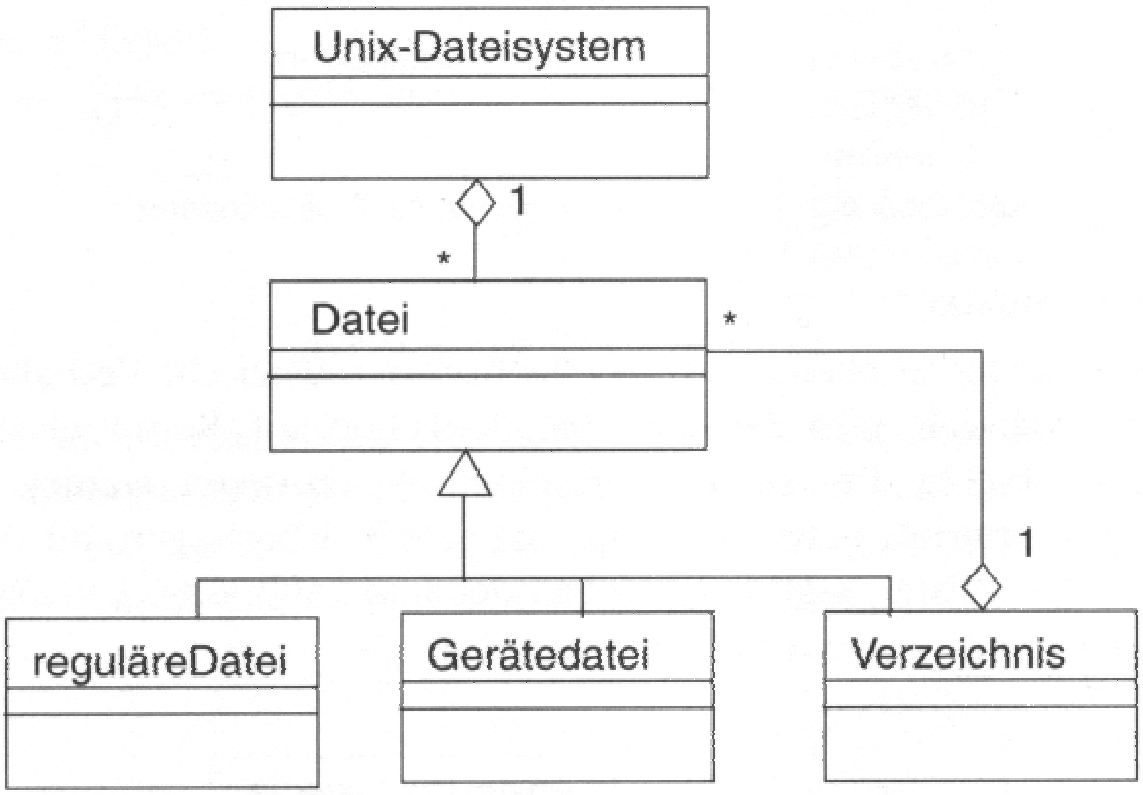


Abbildung 6.7 Beispiel für Kompositum

6.2 Zusammenfassendes Beispiel

Reservierung von Räumen für Lehrveranstaltungen

Use-Cases für das Raumreservierungsbeispiel

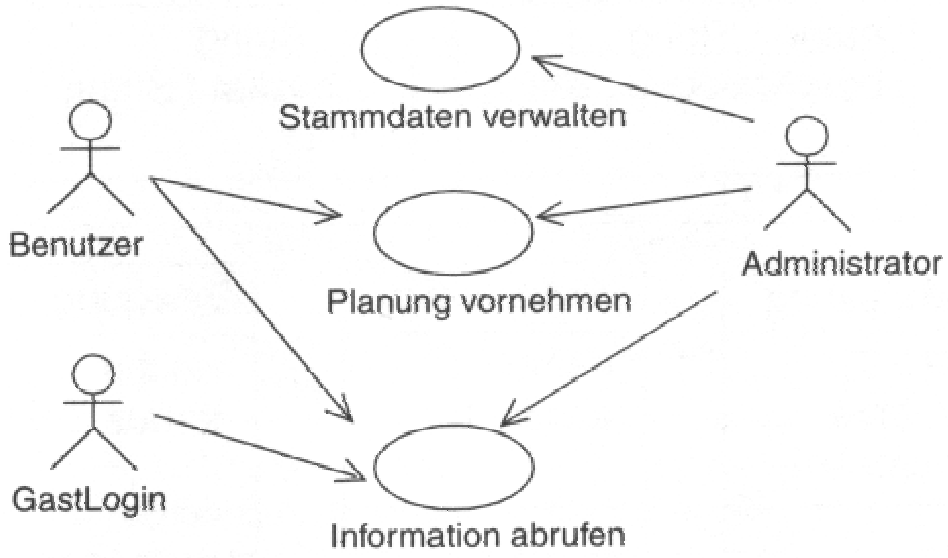


Abbildung 6.13 Use-Case-Szenario

Fachklassen für das Raumreservierungsbeispiel

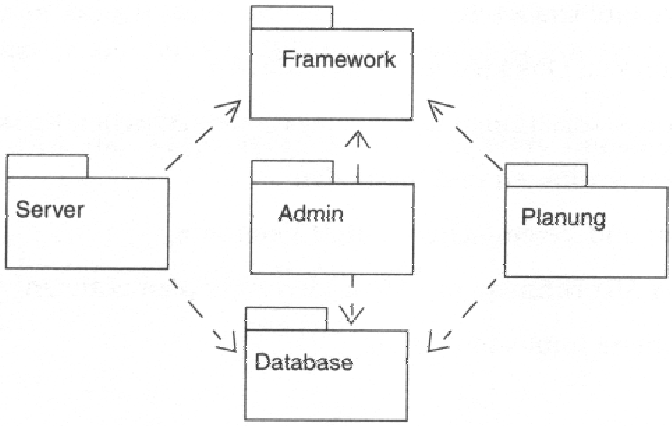


Abbildung 6.14 Pakete

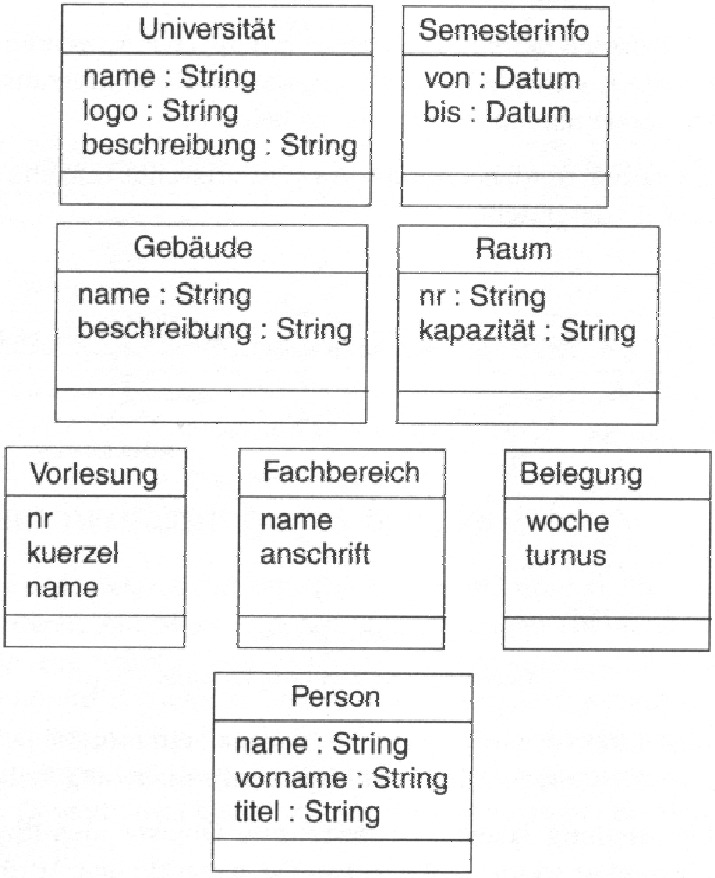


Abbildung 6.15 Fachklassen der Problemdomäne

Beziehungen zwischen den Klassen des Raumreservierungsbeispiels

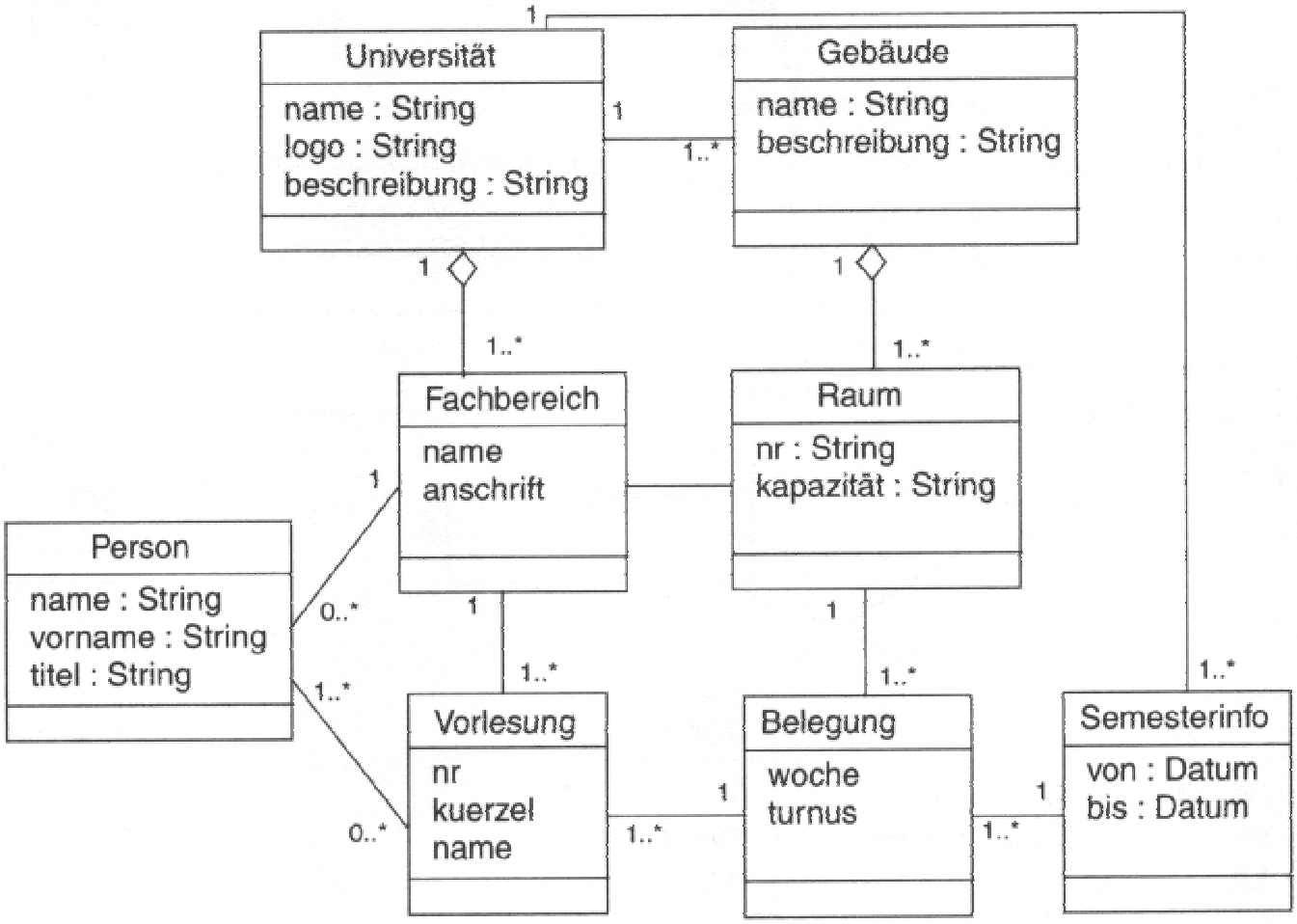


Abbildung 6.16 Beziehungen zwischen Klassen

RMI-Klassen für die Verteilung

Basisklassen für die Verteilung über RMI hinzu fügen

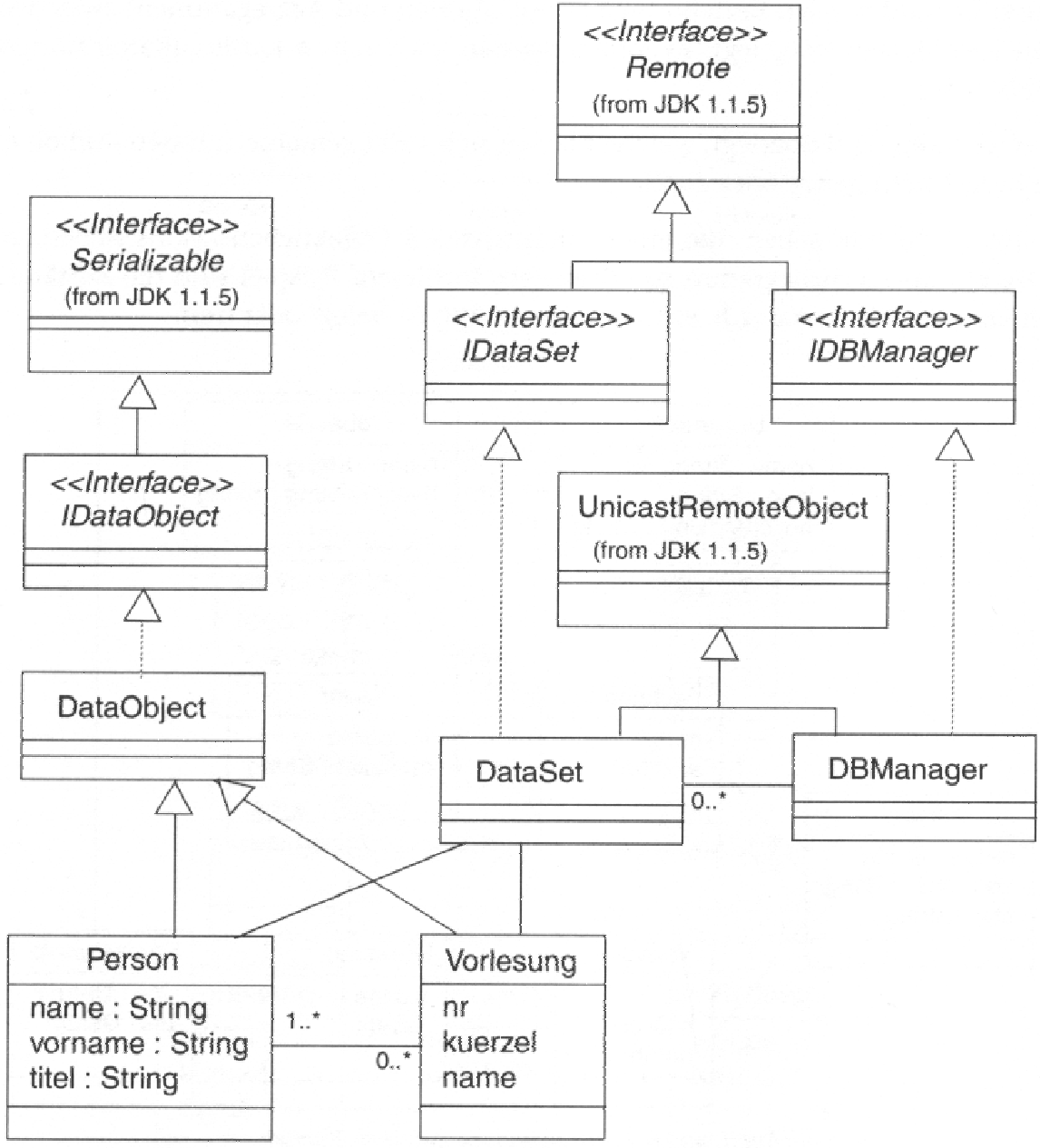
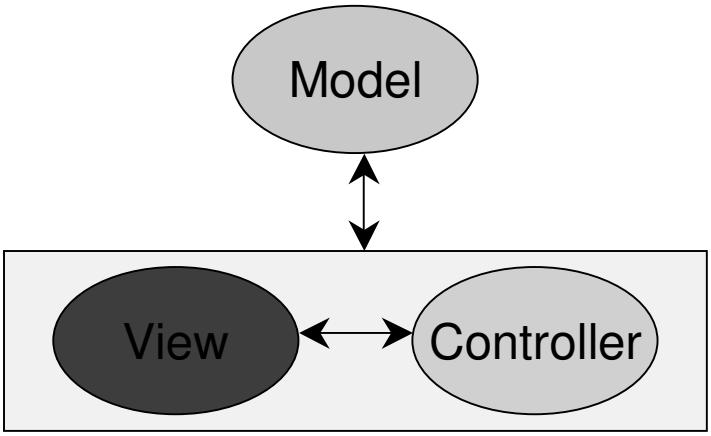


Abbildung 6.17 Basisklassen für Verteilung über RMI hinzufügen

6.3 Umsetzung des Model-View-Controller-Musters in Java

Model - Delegate-Architektur 1/2

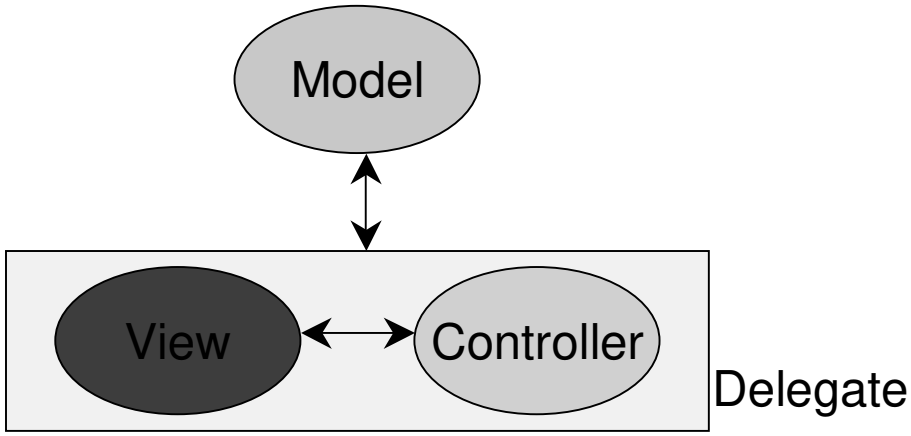


Swing implementiert die Model-Delegate-Architektur.

Jedes Dialogelement von Swing (component) gliedert sich in das Modell (model) und die Benutzerschnittstelle (UI delegate). Innerhalb der Benutzerschnittstelle kann man die Anteile der Benutzeroberfläche (view) und der Anwendungssteuerung (controller) wiederfinden.

Bei dieser Strukturierung verläuft die Kommunikation zwischen den Komponenten etwas anders als bei dem MVC-Modell: Es erfolgt eine Synchronisation des Zustandes des Modells und des Zustandes der Benutzerschnittstelle, die wahlweise von beiden Seiten ausgelöst werden kann.

Model - Delegate Architektur 2/2



Dafür werden `stateChanged-Events` verwendet. Ferner erfolgt ein direkter Abgleich von Controller und View durch andere Ereignisse, die von der jeweiligen Art des Dialogelements abhängig sind. Diese Struktur weist eine Reihe von Vorteilen auf: Sie bindet die Callback-Funktionen der Anwendungssteuerungskomponente enger an das jeweilige Dialogelement. Es hat sich nämlich herausgestellt, daß das Wiederverwenden von Callback-Funktionen in unterschiedlichen Kontexten nur ganz selten möglich ist. (Beispiele dafür sind: Beenden einer Anwendung, Schließen aller Fenster, aber das war es meist auch schon.)

Kommunikation zwischen Model und Delegate in Java

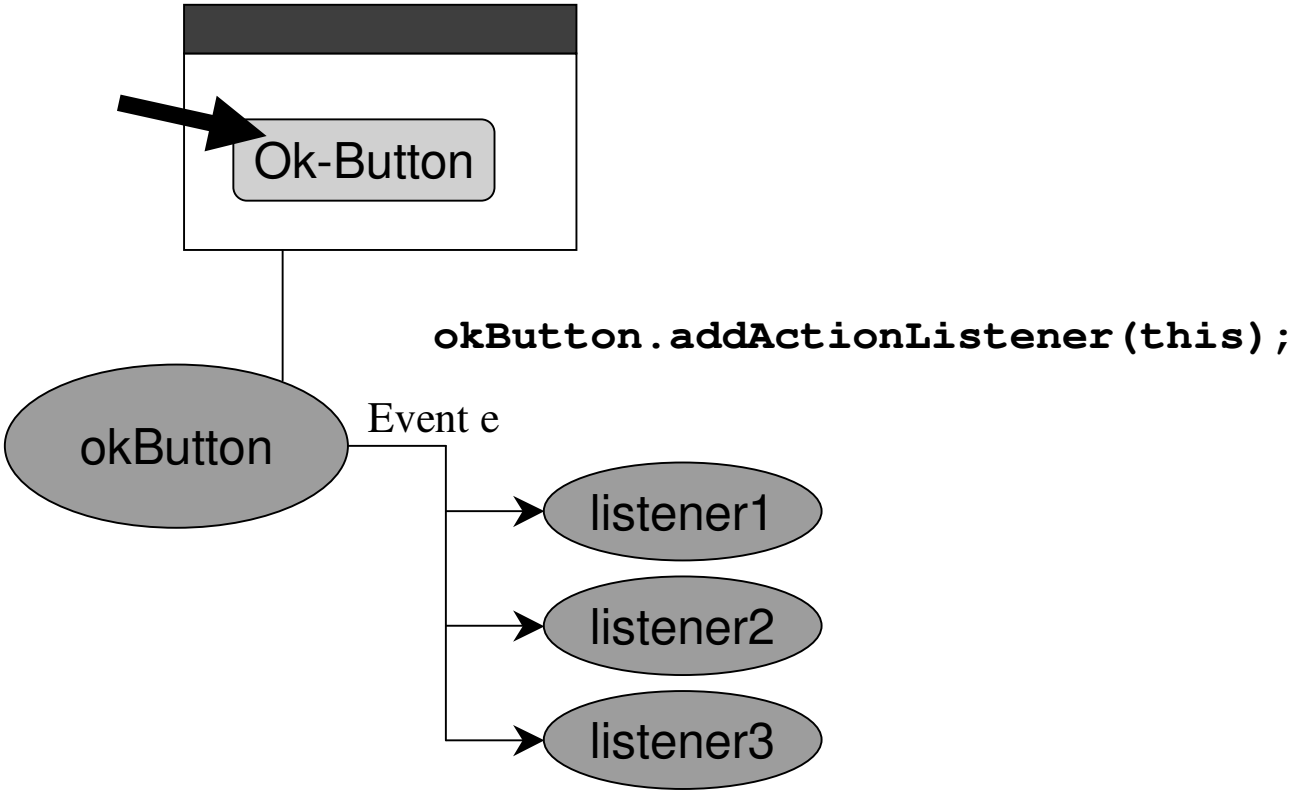
Bisher kennen wir lediglich Methodenaufrufe als Kommunikationsinstrument zwischen zwei Objekten. Wollte nun ein Delegate das Model bspw. darüber informieren, daß ein bestimmter Button gedrückt wurde, müßte es wissen, welche Methode welchen Model-Objektes zur Behandlung dieses *Ereignisses* notwendig ist. Unter Umständen müßten mehrere Model-Objekte mit unterschiedlichen Interfaces informiert werden.

Zur Vereinfachung dieses Problems implementiert Java das *Observer - Observable Muster*.

Observer - Observable Muster

Dieses Muster realisiert eine 1:n Verknüpfung zwischen einem Objekt (Observable) und weiteren davon abhängigen Objekten (Observers). Eine Zustandsänderung des Objektes hat dabei automatisch eine Benachrichtigung der anderen Objekte zur Folge.

Die Verknüpfung zwischen Observable und Observers kann unter anderem über das Ereignis-Konzept erfolgen.



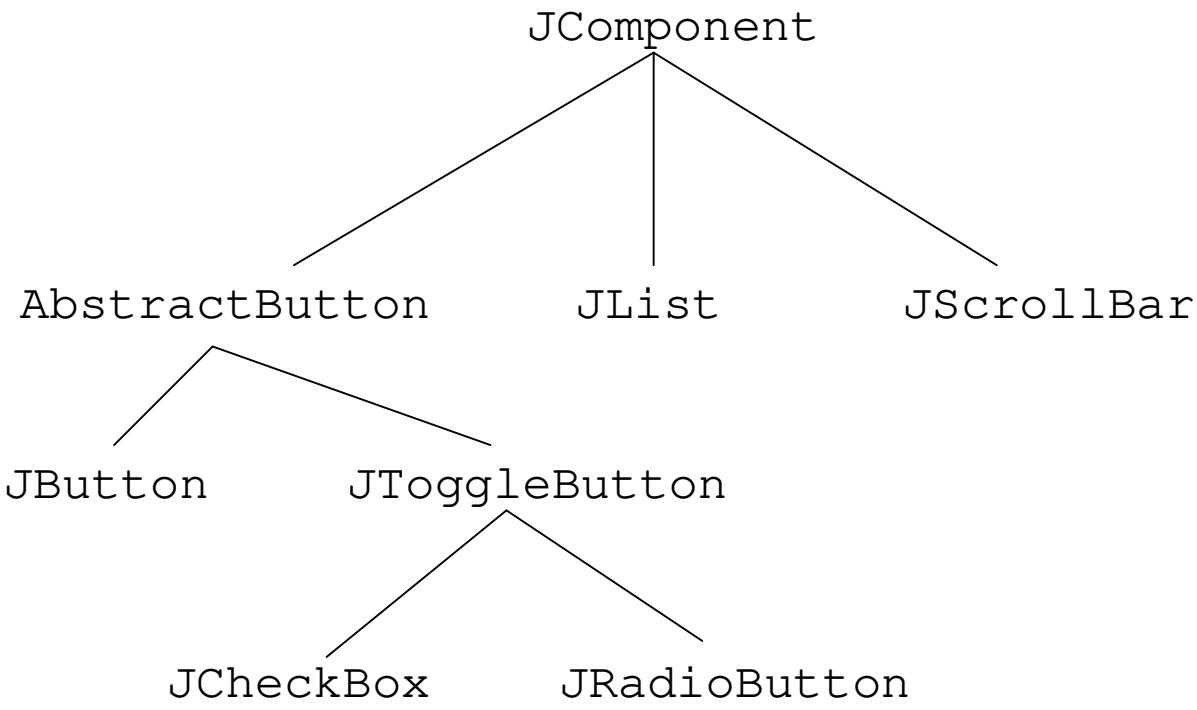
Ereigniskonzept

Wesentliche Begriffe in der Programmierung von Benutzeroberflächen mit Swing sind die Begriffe: Component, Event und EventListener.

Unter Components werden Objekte verstanden, die GUI-Elemente (Widgets) repräsentieren. Ein Beispiel für ein Widget ist ein Button. Interagiert der Benutzer mit einem Widget, erzeugt dieses Events. Verschiedene Benutzerinteraktionen lösen verschiedene Events aus. Jede Component verwaltet Listen von EventListeners für verschiedene Event-Typen. Ein EventListener ist ein Objekt, das sich für ein Event eines bestimmten Typs interessiert. Um Events einer bestimmten Component zu empfangen, muß sich dieses bei der Component anmelden.

Überblick Components

Auszug aus der Ableitungshierarchie der Swing Components:



Zur Registrierung von EventListenern verfügen Components über Methoden

```
addXYZListener(XYZListener l);
```

Beispiel JButton

Objekte der Klasse JButton repräsentieren ein Button Widget. Dieses verwaltet eine Liste von ActionEventListener Objekten, die ein ActionEvent erhalten, wenn der Benutzer den Button klickt.

```
class JButton
  extends AbstractButton
{
 public void
  addActionListener(ActionListener l);
}
```

Beispiel ActionEventListener

Das Drücken eines Buttons löst ein Ereignis vom Typ ActionEvent aus. ActionEvents können von ActionListener Objekten verarbeitet werden.

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

Beispiel Anwendung

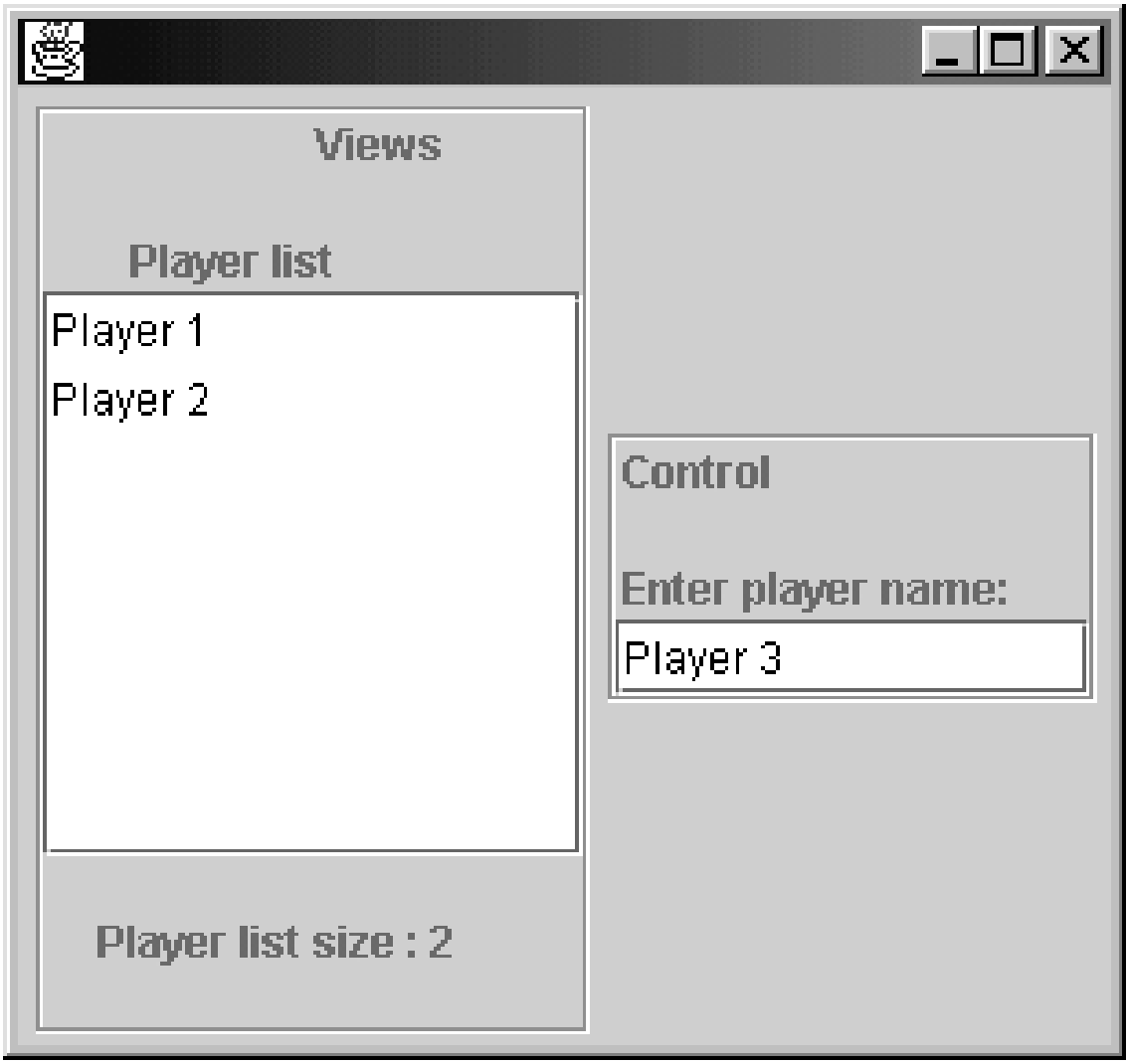
Nachfolgender Codeauszug zeigt, wie ein Objekt aus dem Problembereich mit einem Delegate verbunden wird.

```
class ButtonCtrl implements ActionListener
{
  public void
       actionPerformed(ActionEvent e)
    { System.out.println("ok."); }
}

public class MyClass
{
  public static ButtonCtrl bl;

  public static void main()
  {
    bl              = new ButtonCtrl();
    JButton    b    = new JButton("Ok");

    b.addActionListener(bl);
  }
}
```

Beispiel: „PlayerList“

Erstellung eines Dialogfensters zur:

1. Anzeige der Einträge und der Grösse einer Spielerliste und
2. Eingabe von weiteren Spielernamen



Komponenten des Beispiels

„Modell“

Das Datenmodell umfasst eine Liste in der die Spielernamen gespeichert werden.

„View 1“

Innerhalb der ersten Ansicht wird die Liste der eingetragenen Spielernamen angezeigt.

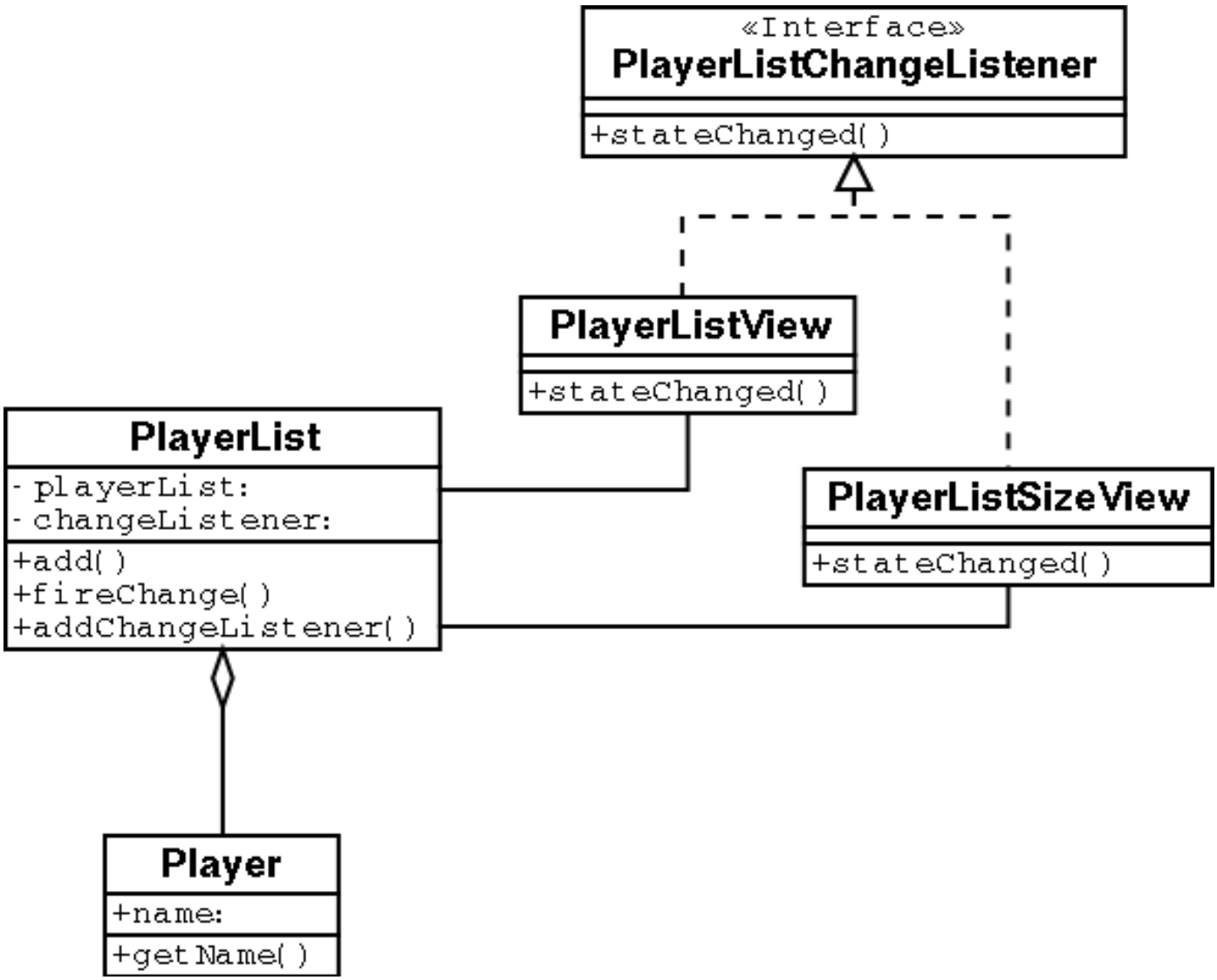
„View 2“

Die zweite Ansicht umfasst die Anzeige der aktuellen Grösse der Spielerliste.

„Controller“

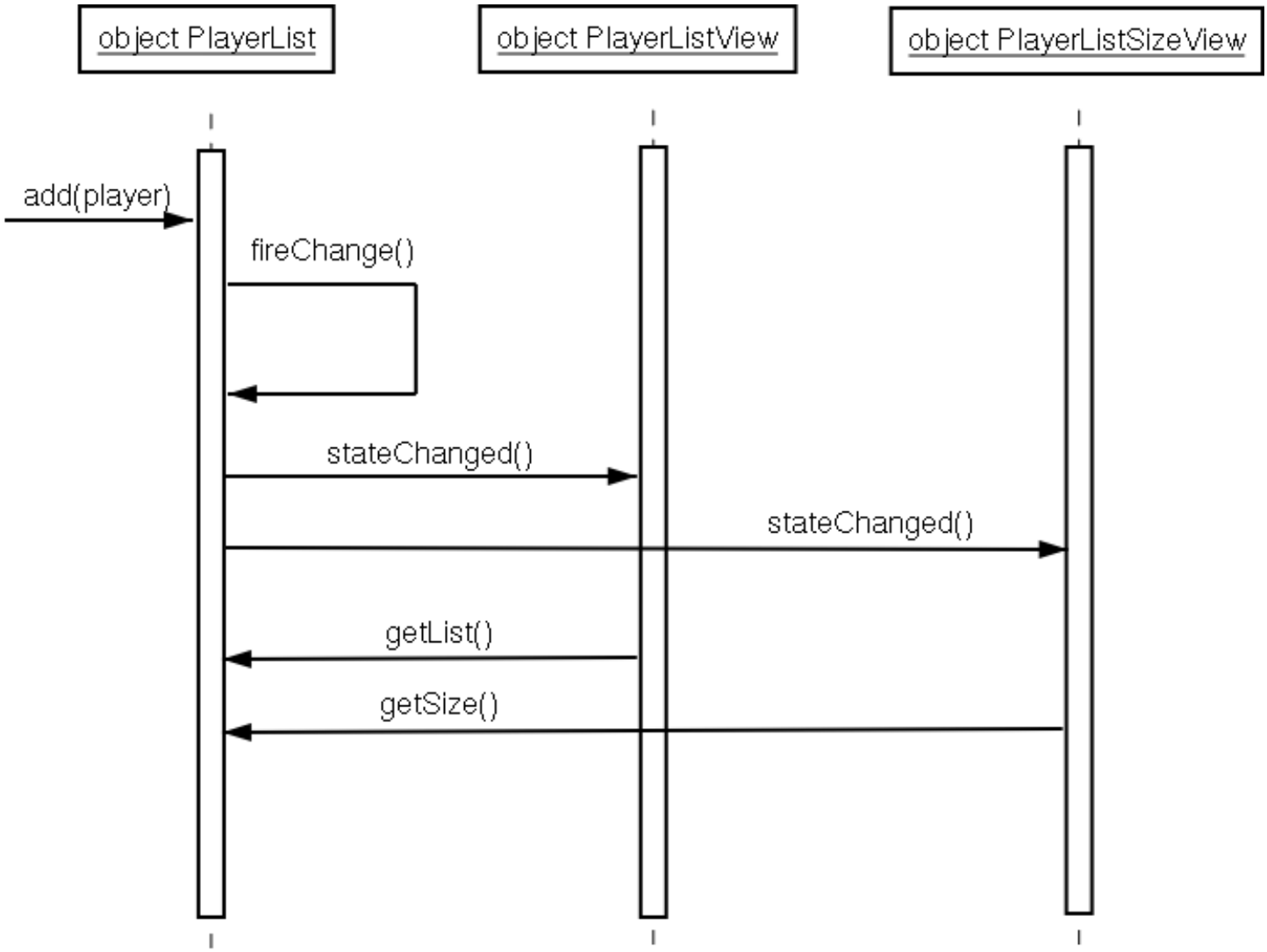
Die Benutzereingaben werden über das Textfeld angenommen und entsprechend ausgewertet.

Klassendiagramm



Sequenzdiagramm

Ereignisfolge beim Einfügen eines neuen Spielernamens:



Implementierung der Views (1)

Allgemeiner „Listener“ für Änderungen der Spielerliste:

```
public interface PlayerListChangeListener
{
   // this method is called when the player list changes
   public void stateChanged();
}
```

Implementierung der Views (2)

Ansicht zur Anzeige der Listengrösse:

```
public class PlayerListSizeView extends JPanel
                implements PlayerListChangeListener
{
   // the observable
   PlayerList playerList;
   // the label that shows the size
   JLabel label = new JLabel();

   public PlayerListSizeView(PlayerList playerList)
   {
      this.playerList = playerList;

      // set the Swing component
      add(label);

      // update the label
      stateChanged();
   }

   public void stateChanged()
   {
      label.setText("Player list size : " + playerList.getSize());
   }
}
```

Implementierung der Views (3)

Ansicht zur Anzeige der Spielernamen:

```
public class PlayerListView extends JPanel
                implements PlayerListChangeListener
{
   JList list = new JList(new DefaultListModel());

   // player list that is observed
   PlayerList playerList;

   // construction
   public PlayerListView(PlayerList playerList)
   {
      // set the observable
      this.playerList = playerList;

      setLayout(new BoxLayout(this, BoxLayout.Y_AXIS));

      // set the Swing components
      add(new JLabel("Player list"));
      JScrollPane scrollPane = new JScrollPane(list);
      add(scrollPane);

      // update the JList
      stateChanged();
   }
```

Implementierung der Views (4)

Ansicht zur Anzeige der Spielernamen (Fortsetzung):

```
public void stateChanged()
{
      // get the current list model
      DefaultListModel dlm = (DefaultListModel) list.getModel();
      // and clear all entries
      dlm.clear();

      // get all names from the observer player list
      ListIterator it = playerList.getList().listIterator();
      while (it.hasNext()) {
         Player current = (Player) it.next();
         // add them to the list model
         dlm.addElement(current.getName());
      }
   }
} // class PlayerListView
```

Implementierung des Modells

Verwaltung der Spielernamen:

```
public class PlayerList
{
   private ArrayList playerList = new ArrayList();
   private ArrayList changeListener = new ArrayList();

    public void add(Player p) {
        playerList.add(p);
        // notify listeners
        fireChange();
    }

    public ArrayList getList() { return playerList; }
    public int getSize()       { return playerList.size(); }

    public void fireChange() {
        // notify all listeners that the player list has changed
        ListIterator it = changeListener.listIterator();
        while (it.hasNext()) {
            PlayerListChangeListener current =
                    (PlayerListChangeListener) it.next();
            current.stateChanged();
        }
    }

    public void addChangeListener(PlayerListChangeListener c)
    {  changeListener.add(c); }
}
```

Integration der Komponenten (1)

Erzeugung des Hauptfensters:

```
public class Main
{
   // the data model
   static PlayerList playerList = new PlayerList();

   // the input field for adding new players
   static JTextField textField = new JTextField();

   public static void main(String[] args)
   {
      // create the application window
      JFrame mainFrame = new JFrame();
      Container c = mainFrame.getContentPane();
      c.setLayout(new FlowLayout());
      ....
```

Integration der Komponenten (2)

Erzeugung der Views:

```
/* ---- create views ---- */

 // create the first view ...
 PlayerListView plv = new PlayerListView(playerList);
 // ... and add it to the listener list of the model
 playerList.addChangeListener(plv);

 // create the second view ...
 PlayerListSizeView plsv = new PlayerListSizeView(playerList);
 // ... and add it to the listener list of the model
 playerList.addChangeListener(plsv);

 // integrate the views into the application frame
 JPanel viewPanel = new JPanel();
 viewPanel.setBorder(BorderFactory.createEtchedBorder());
 viewPanel.setLayout(new BoxLayout(viewPanel,
                                   BoxLayout.Y_AXIS));
 viewPanel.add(new JLabel("Views"));
 viewPanel.add(Box.createVerticalStrut(15));
 viewPanel.add(plv);
 viewPanel.add(plsv);
 c.add(viewPanel);
```

Integration der Komponenten (3)

Erzeugung des Controllers:

```
    /* ---- create controller ---- */

    // create the controller (textField + ActionListener)
    textField.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
               playerList.add(new Player(textField.getText()));
               textField.setText("");
            }
        });

    // integrate the controller into the application frame
    JPanel controlPanel = new JPanel();
    controlPanel.setBorder(BorderFactory.createEtchedBorder());
    controlPanel.setLayout(new BoxLayout(controlPanel,
                                         BoxLayout.Y_AXIS));
    controlPanel.add(new JLabel("Control"));
    controlPanel.add(Box.createVerticalStrut(15));
    controlPanel.add(new JLabel("Enter player name:"));
    controlPanel.add(textField);
    c.add(controlPanel);

    mainFrame.pack();
    mainFrame.setVisible(true);
  }
}
```