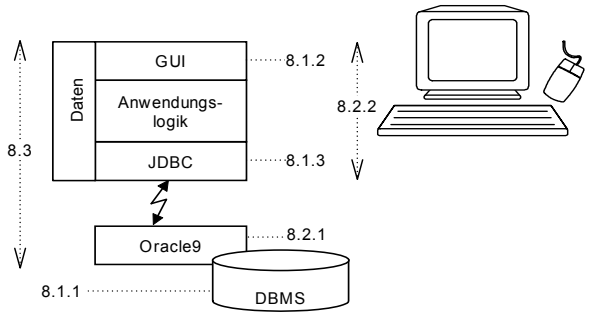
8 Ein Anwendungsbeispiel

Die in den Kapiteln 2 und 3 behandelten Themen sollen nun beispielhaft in einer Anwendung "Kursverwaltung" zusammengefasst werden. Dabei werden für die GUI-Programmierung und das Datenmanagement professionelle Instrumente wie Oracle9 und Swing eingesetzt.
Als Grundlage dieser Datenbankanwendung ist eine Zweiebenenstruktur gewählt, und zwar mit Oracle9 auf der Server- und mit einer Java/JDBC-Applikation auf der Clientseite (Abb. 8-1 links). Die Anwendung ist aber so programmiert, dass eine Verlagerung der Anwendungslogik auf eine mittlere Ebene einigermaßen einfach möglich sein sollte.
Die Anbindung der Datenbank an die Anwendung erfolgt mit JDBC, und das GUI für die Datenanzeige und die Datenmanipulation beruht auf Swing-Klassen der Java Foundation Classes (JFC, vgl. Abschnitt 8.2.2).
Die Swingklassen unterstützen die systematische Trennung von Präsentation (GUI) und Daten. Vor allem stellen sie effiziente Verfahren zur Verfügung, um die Daten graphisch wiederzugeben und umgekehrt Eingaben in Änderungen der Daten zu transformieren. Das Swing-Analogon zu Datenbanktabellen, die Klasse `JTable`, bietet sich einer solchen Verwendung geradezu an.
Als Datenbanksystem kommt Oracle9 zum Einsatz, das bereits im Zusammenhang mit vorbereiteten Anweisungen und gespeicherten Prozeduren verwendet wurde (`PreparedStatement` und `CallableStatement`, vgl. Abschnitt 3.3). Als Tabellen werden die gleichen wie bisher verwendet, mit einigen Erweiterungen wie die Hinzunahme von Tabellen für Kunden (`Personen`) und `Teilnehmer`.
In den folgenden Abschnitten wird zunächst die Konzeption der Anwendung vorgestellt. Dazu gehören vor allem die Datenmodellierung, die Festlegung der Funktionalität und der GUIs und die Strukturierung der JDBC-Anbindung. Dem schließt sich die Darstellung der benötigten Arbeitsmittel an, wozu insbesondere das Oracle9-DBMS, die MVC-Methodik der Swing-Klassen sowie einige der Swing-Klassen selbst gehören (vgl. Abschnitt 6.2.2). Im letzten Abschnitt des Kapitels ist beispielhaft eine Komponente der Anwendung programmiert und eingehend kommentiert.

Das folgende Bild dient dem Geleit durch die Abschnitte des Kapitels und steht dort jeweils an erster Stelle nach den Überschriften, mit grauer Hinterlegung derjenigen Blöcke, die in den betreffenden Abschnitten relevant sind. Beispielsweise ist der JDBC-Teil des Programms in Abschnitt 8.1.3 enthalten.



8.1 Konzeption

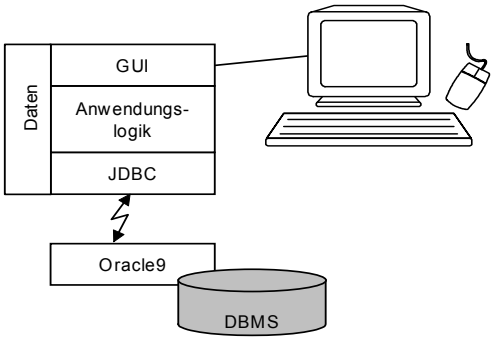
8.2 Arbeitsmittel

8-3 Der Client

8.1 Konzeption

Ziel ist ein einfaches Kursverwaltungssystem auf der Grundlage der in den Beispielen immer wieder verwendeten Tabellen der Datenbank "Kurse". Sie besteht aus Tabellen für Kurse (`Kurse`), Referenten (`Dozenten`) und Kunden (`Personen`), die an Kursen teilnehmen. Diese Tabellen enthalten die Basisbestände an Personal, Kunden und Kursen und sind somit der tragende Teil der Anwendung, d.h. sie beschreiben dessen *Statik*. Solche Daten werden auch als *Bestands- oder Stammdaten* bezeichnet. In der Anwendung sollen diese Bestände gepflegt, d.h. geändert, Neuzugänge registriert und ggf. veraltete Daten gelöscht werden können.
In der Tabelle `Teilnehmer` werden Zuordnungen von Personen zu Kursen vorgenommen. Diese Art der Zuordnung beschreibt gleichsam die *Dynamik* des Systems. Dementsprechend werden die damit verbundenen Daten auch *Bewegungsdaten* genannt.

8.1.1 Datenmodell



Die Datenbank besteht aus den vier Tabellen

- `Dozenten(dcode, nachname, vorname)`
- `Personen(pcode, nachname, vorname)`
- `Kurse(kcode, dcode, typ, bezeichnung, datum, zeit)`
- `Teilnehmer(pcode, kcode)`

Die Schlüssel in den Tabellendefinitionen sind hrvorgehoben, und zwar die Primärschlüssel durch Unterstreichen und die Fremdschlüssel durch Schrägstellen.
Die Beziehungen zwischen den Tabellen können der Abb. 8-1 entnommen werden. Im einzelnen sind das:

- Ein *Dozent* hält `N` *Kurse*, und ein *Kurs* wird von einem *Dozenten* gehalten.
- Eine *Person* repräsentiert `N` *Teilnehmer* an `N` verschiedenen *Kursen*, und umgekehrt wird ein *Kurs* von `M` *Teilnehmern*, d.h. `M` *Personen* besucht.

Die Tabelle Teilnehmer ist erforderlich, um die `N:M`-Beziehung zwischen den Tabellen Personen und Kurse in eine `1:N`- und eine `1:M`-Beziehung aufzulösen. Denn sowohl in der Tabelle `Personen` als auch in der Tabelle `Kurse` müsste jeweils der Primärschlüssel der anderen Tabelle als Fremdschlüssel verwendet werden, was ohne Mehrdeutigkeiten mit Tabellen fester Spaltenzahl nicht möglich ist. (Wären mehrere Dozenten für einen Kurs vorgesehen - was durch die gewählte Modellierung ausgeschlossen wurde! -, müsste auch diese Beziehung entsprechend aufgelöst werden.)

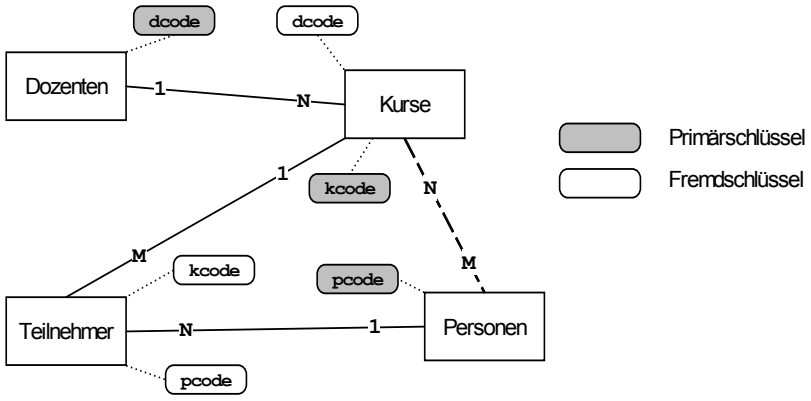
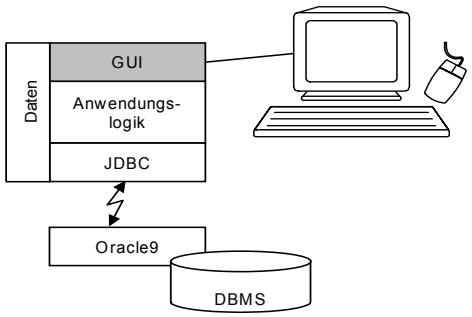


Abb. 8-1: Datenmodell der Anwendung

Die Primär- und Fremdschlüssel, über die die Tabellen in Beziehung gesetzt werden, sind in den Kästchen mit runden Ecken angegeben. Primärschlüssel sind außerdem grau hinterlegt. Die aufgelöste N:M-Beziehung ist gestrichelt eingezeichnet.

8.1.2 GUIs



Als Grundelemente des GUI werden vor allem Swing-Tabellen verwendet, die, wie in Abb. 8-2 gezeigt, oben mit Schaltknöpfen für das Erzeugen neuer Zeilen, das Löschen und Ändern vorhandener Zeilen und zum Erneuern der Ansicht (refresh) berandet sind. Über einen Knopf "Kursteilnehmer" beispielsweise wird ein GUI aktiviert, in dem Personen als Teilnehmer einem zuvor gewählten Kurs zugeordnet werden können. (Im Programm in Abschnitt 8.3 wird dieser Knopf allerdings dazu missbraucht, die Datenbanktabellen wieder in ihren Urzustand zu bringen.)
Datenänderungen werden entweder direkt in den Tabellenzellen oder durch Werteauswahl in Popup-Menüs vorgenommen.

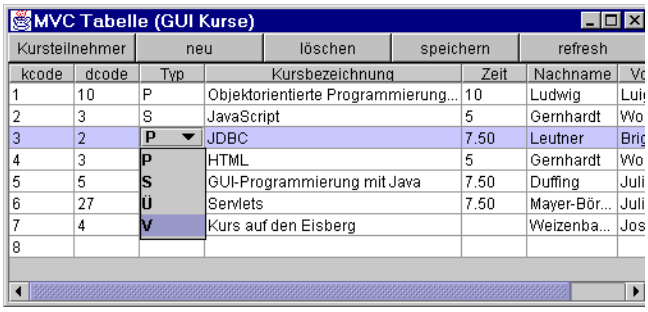


Abb. 8-2: GUI (Swing-Tabelle) für die Manipulation der Tabelle `Kurse`

Änderungen sowie neue Zeilen sind zunächst noch widerrufbar, und mit "refresh" kann der alte Zustand wiederhergestellt werden. Erst durch Betätigen von "speichern" werden die Änderungen in der gerade markierten Zeile dauerhaft (persistent).
Die GUI-Struktur des gesamten Anwendungssystems ist in Abb. 8-3 skizziert. Die GUIs für die Tabellenanzeige und -manipulation sind alle ähnlich wie die eingeblendete Swing-Tabelle aus Abb. 8-2 aufgebaut.

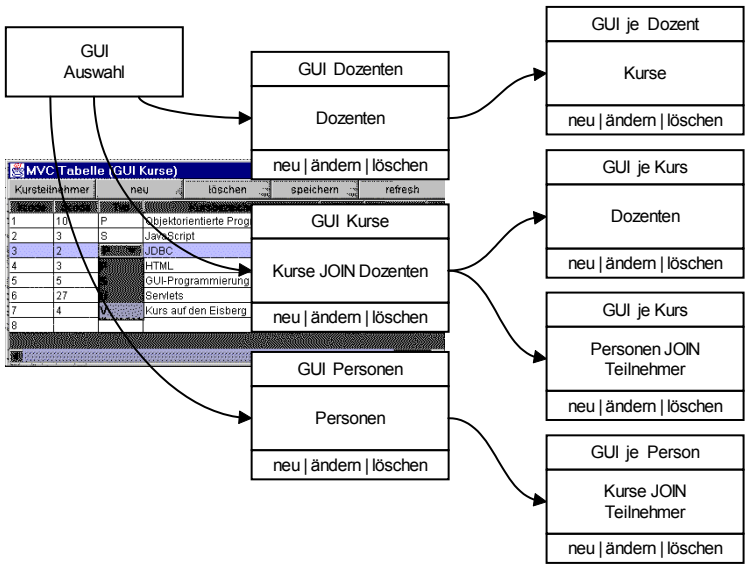


Abb. 8-3: GUI-Struktur

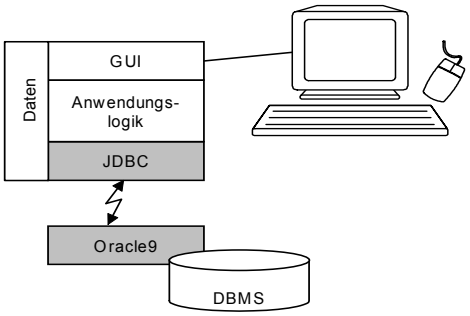
Die Funktionsauswahl besteht im einfachsten Falle aus drei adäquat beschrifteten Knöpfen, die beim Betätigen zu den in der Bildmitte symbolisierten GUIs führen. In diesen GUIs können die Stammdatentabellen bearbeitet werden, und dort kann nach Wahl einer Tabellenzeile und nach Klicken eines entsprechenden Knopfes eine Zuordnung von Daten aus den anderen Tabellen erfolgen. Diese Funktionalität ist durch die vier Kästchen auf der rechten Bildseite symbolisiert:

1. *Ein* Dozent kann einen oder *mehrere* Kurse leiten.
2. *Einem* Kurs kann genau *ein* Dozent zugeordnet werden.
3. *Ein* Kurs kann ein oder *mehrere* Teilnehmer haben.
4. *Eine* Person kann an einem oder *mehreren* Kursen teilnehmen.

In jedem der Kästchen ist oben die gewählte Zeile aus der betreffenden Tabelle angegeben, z.B. "GUI je Kurs" oder "GUI je Person". Dem Mittelteil kann jeweils entnommen werden, aus welcher Tabelle oder welcher View, z.B. aus der Verknüpfung von `Personen` und `Teilnehmer` in `Personen JOIN Teilnehmer`, Zeilen zugeordnet werden können.

Bei jedem der Kästchen ist außer den angegebenen Funktionen "neu | ändern | löschen" noch das Berichtswesen in Form von Übersichten, Bestätigungen, Rechnungen etc. hinzuzudenken.

8.1.3 JDBC-Anbindung



Als Erstes sind einige Produktfestlegungen erforderlich, nämlich welches Datenbanksystem eingesetzt und mit welchen JDBC-Treibern dieses Datenbanksystem an Java gebunden werden soll. Sorgfalt ist dabei angebracht, denn die Bindungen an die gewählten Produkte sind in der Regel derart, dass ein Wechsel des Datenbanksystems oder des Treibers oder von beidem einigen Aufwand erfordert.
Fast jedes Datenbanksystem hat seinen eigenen SQL-Dialekt. So hat Oracle9 seine eigene Syntax für Outer Joins, die in anderen Datenbanksystemen in der Regel so nicht vorkommt. Aber selbst bei gleichen Datenbanksystemen sind die Reaktionen auf gleiche SQL-Anweisungen unter Umständen unterschiedlich, je nachdem, über welche Schnittstelle diese Anweisungen gegeben wurden. Solche Abhängigkeiten können zwar minimiert werden, beispielsweise durch Verzicht auf gespeicherte Prozeduren oder Verwendung nur eines eingeschränkten Sprachumfanges von SQL; die fast zwangsläufige Folge solcher Einschränkungen sind aber oft erhebliche Effizienzeinbußen.
Als Datenbank wird Oracle9 verwendet, und zwar als sogenannte Oracle9*i*Database oder Oracle9*i*Personal. Die Personal ist die Einzelnutzerversion von Database und mit dieser kompatibel, d.h. alle getätigten „Personal"-Entwicklungen sind auf Oracle9*i*Database übertragbar.
Für die JDBC-Anbindung ist der „Thin"-Treiber von Oracle (Typ 4) geeignet. Er ist in der Oracle9-Distribution enthalten. Vorsichtig sein sollte man mit ODBC-Treibers, die zumindest mit der JBDC-Brücke von Sun Schwierigkeiten bereiten können, beispielsweise im Zusammenhang mit gespeicherten Prozeduren.
Die Teil-GUIs sind zwar für verschiedene Datenbanktabellen bzw. Views zuständig, haben aber immer das Holen, Ändern, Löschen und Einfügen von Tabellenzeilen als gemeinsame Funktionen. Die entsprechenden SQL-Anweisungen können vorbereitet werden, so dass zum Zeitpunkt des Aufrufes gegebenenfalls nur noch die aktuellen Schlüssel bzw. Tabellenwerte eingesetzt werden müssen. Für das Programm, das der Abb. 8-2 zugrunde liegt, und für die Tabellen `Kurse` und `Dozenten` sind das die im Folgenden beschriebenen SQL-Anweisungen.

- `SELECT` wird ohne weitere Parametrisierung verwendet, d.h., dass die Tabelle `Dozenten` und die View `Kurse LEFT JOIN Dozenten` jeweils immer als Ganzes von der Anwendung gelesen werden. Für diese `SELECT`s kann also ein gewöhnliches, mit dem

Standardkonstruktor erzeugtes `Statement`-Objekt angelegt und bei Bedarf entweder durch Aufruf von `execute(`*`sqlAnweisung`*`)` oder mit `executeQuery(`*`sqlAnweisung`*`)` ausgeführt werden.

Die Tabellen `Kurse` und `Dozenten` werden im folgenden Ausdruck durch einen *Left* Join verknüpft, der sich in Oracle-SQL syntaktisch durch das geklammerte Pluszeichen (+) auf der *rechten* Seite des Vergleichsoperators = ausdrückt:

```
SELECT kcode, dcode, typ, bezeichnung, zeit,
     nachname, vorname FROM Kurse, Dozenten
     WHERE Kurse.dcode = Dozenten.dcode (+)
```

Die `SELECT`-Operation auf die Tabelle `Dozenten` dient der Beschaffung aller Werte, insbesondere derer von `dcode`. Aus ihnen wird der Wertebereich (Domäne) gebildet, der für Eingaben in der entsprechenden GUI-Tabellenzelle der Spalte `dcode` in der Tabelle `Kurse` von Relevanz ist. Da alles gelesen wird, kann, wie zuvor, ohne Parametrisierung des SQL-Ausdrucks verfahren werden:

```
SELECT dcode,nachname,vorname FROM Dozenten
```

- `UPDATE` bezieht sich auf einen einzigen Datensatz, der über das Primärschlüsselfeld `kcode` bestimmt wird. Der Wert dieses Schlüsselfeldes und die vorzunehmenden Änderungen ergeben sich zur Laufzeit und beziehen sich auf alle Spalten der Tabelle `Kurse`, die Primärschlüsselspalte allerdings ausgenommen. Dieser `UPDATE` ist also parametrisiert und wird infolgedessen als `PreparedStatement`-Objekt angelegt, bei Bedarf mit den aktuellen Parametern versehen und mit `execute()` bzw. `executeUpdate()` (mit leeren Parameterklammern!) aufgerufen. Die SQL-Anweisung, die bei der Instanziierung des `PreparedStatement`-Objektes verwendet wird, ist

  ```
  UPDATE Kurse
     SET dcode=?, typ=?, bezeichnung=?, zeit=? WHERE kcode=?
  ```

  Die Fragezeichen stehen an denjenigen Stellen, die vor Ausführung der Anweisung mit aktuellen Werten versehen werden müssen.
- `DELETE` bezieht sich wie `UPDATE` auf einen einzigen Datensatz, der über den Primärschlüssel der ausgewählten Tabellenzeile festgelegt wird. Auch diese SQL-Anweisung wird parametrisiert und als `PreparedStatement`-Objekt abgelegt, um bei Bedarf mit `execute()` bzw. `executeUpdate()` aufgerufen zu werden:

  ```
  DELETE FROM Kurse WHERE kcode=?
  ```
- `INSERT` wird über eine benutzerdefinierte gespeicherte Prozedur realisiert. Aufgerufen wird sie mit der Anweisung

  ```
  {? = call InsertKurse(?, ?, ?, ?)}.
  ```

  Das erste Fragezeichen steht für einen Rückgabewert, nämlich den Primärschlüsselwert der eingefügten Tabellenzeile. Über die restlichen Fragezeichen sind die Spalteninhalte der einzufügenden Zeile definiert. Die Codierung der gespeicherten Prozedur selbst ist in Programm 8-3c gezeigt und wird im vorangehenden Teil b des Programms verwendet.

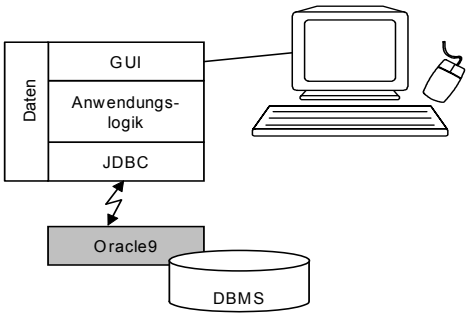
  Ein Grund für die Verwendung einer gespeicherten Prozedur besteht darin, dass in zwei aufeinanderfolgenden Schritten erst ein neuer Primärschlüsselwert aus allen bereits vorhandenen bestimmt wird (Maximum plus 1) und dieser Schlüsselwert sodann mit der neuen Zeile in die Tabelle eingefügt wird. Im Falle der Verwendung

von gewöhnlichen JDBC-Statement-Objekten müssten die beiden Aktionen einzeln über die JDBC-Schnittstelle und das Internet ausgeführt werden. Die Verwendung einer gespeicherten Prozedur reduziert diese Aktionen auf eine einzige, ist in der Regel also mit einem Gewinn an Effizienz verbunden und erleichtert die Meidung von Konkurrenzproblemen bei Mehrfachzugriffen.

8.2 Arbeitsmittel

Neben JDBC werden für die Realisierung der Anwendung zusätzliche Arbeitsmittel benötigt, und zwar das Oracle9-Datenbanksystem und die Swingklassen, vor allem deren MVC-Funktionalität. Sie werden in den beiden folgenden Abschnitten im erforderlichen Umfang erläutert.

8.2.1 Oracle9



Oracle9 lässt sich mit unterschiedlichem Umfang installieren, etwa in der Gestalt von Oracle9iDatabase als Datenbank-Server oder Oracle9iPersonal als Einzelplatzsystem. Das Anwendungsprogramm sollte mit allen Fassungen, vielleicht mit Ausnahme von Oracle Lite, zufrieden sein.
Oracle9*i*Personal kann alles, was auch Oracle9*i*Database kann, ausgenommen natürlich die Serverfunktionen. Mit Oracle9*i*Personal entwickelte Datenbankanwendungen lassen sich ohne Weiteres in Server-Umgebungen transportieren.
Oracle9 ist für Standardanwendungsfälle vorkonfiguriert. Dazu zählt auch eine Startdatenbank mit Tabellen und Benutzern für Übungszwecke, die man unbedingt mit installieren sollte. Die Installation unter Windows XP ist unproblematisch, und nach abgeschlossener Installation können unmittelbar eigene Datenbanken in Betrieb genommen werden, sei es durch Import aus anderen Datenbanksystemen wie z.B. MS Access oder durch komplette Neuentwicklung.
Für das Management Für die Verbindungsaufnahme über JDBC und für administrative Aufgaben wird neben dem Benutzernamen noch ein Passwort benötigt. Nach Installation einer Datenbank gelten Voreinstellungen, nämlich (Groß-/Kleinschreibung wird nicht unterschieden):

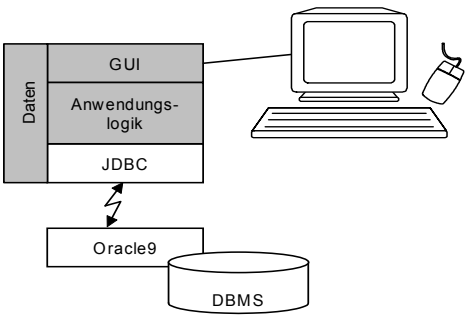
`SYSTEM` als Benutzername (UID) und `MANAGER` als Passwort.

Mit der folgenden Programmanweisung kann die Verbindung mit der Musterdatenbank (in Oracle ein Benutzername) Kurse über JDBC aufgenommen werden:

```
DriverManager.getConnection(
    "jdbc:oracle:thin:@orcl.vsite.de:1521:Kurse",
    "system","manager")
```

Benutzername und Passwort werden auch benötigt, wenn mittels Oracle SQL*Plus direkt an der SQL-Schnittstelle des Datenbanksystems gearbeitet werden soll.

8.2.2 MVC und Swing-Tabellen



Swing bzw. die *Java Foundation Classes* (*JFC*), zu denen Swing gehört, bringen die GUI-Programmierung in Java auf ein dem Stand der Technik einigermaßen entsprechendes Niveau. Die Swing-Klassen sind Bestandteil des `javax.swing`-Packages, die seit JDK 1.2 als sogenannte Standarderweiterungen[1] des JDK geführt werden (zu den Standarderweiterungen gehören auch alle anderen mit `javax` beginnenden Packages, z.B. die Servlet- und JSP-Packages).

Swing umfasst bereits in den frühen Versionen (1.1) ca. 1200 Klassen, also mehr als doppelt so viele wie die erste Version des Java Development Kits enthielt, von denen allerdings nur einige wenige hier benötigt werden. Von Wichtigkeit ist die Architektur, mit der Ordnung in diese Klassen gebracht wird, und wie aus einem Grundverständnis heraus Extrapolationen auf das Gesamtpaket möglich sind. Vollständig beschrieben sind die Swing-Klassen in der Dokumentation der JDK bzw. SDK.

Die Model-View-Controller-Architektur (MVC) ist Gestaltungsprinzip bei allen Swing-Komponenten. In dieser Architektur ist jede GUI-Komponente in die folgenden drei Teile zerlegt:

- *Model* (Datenmodell)
  Im *Model* ist der Zustand der Komponente festgehalten. Unterschiedliche Komponententypen wie Rollleiste oder Tabelle haben unterschiedliche Models. Das Model einer Rollleiste beispielsweise enthält die aktuelle, minimale und maximale Schieberposi-

[1] heute kurz *Optionen* genannt

tion. Das Model selbst ist unabhängig von der graphischen Ausgestaltung der Komponente.

- *View* (Aussehen)
  Die *View* bestimmt das Aussehen der Komponente auf dem Bildschirm. Sie stellt die Daten des Models graphisch dar, etwa die Rollleiste im folgenden Beispiel.
- *Controller* (Steuerung)
  Der *Controller* legt fest, wie die GUI-Komponente auf Ereignisse, z.B. auf Mausklicks und Tastatureingaben, reagiert.

View und Controller bestimmen das sogenannte *Look & Feel* des GUI, etwa das der Windows XP- oder das einer Motiv-Oberfläche.
In Abb. 8-4 ist als Beispiel eine Rollleiste gezeigt.

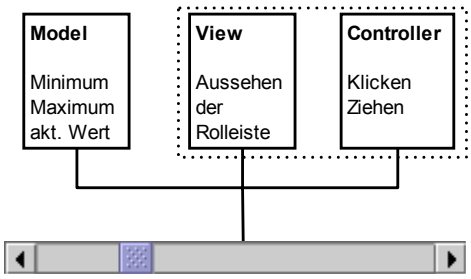


Abb. 8-4: Rollleiste (Scrollbar) und MVC

Das Model enthält außer dem aktuellen Stellwert des Rollleistenschiebers auch dessen Wertebereichsgrenzen. Die View sorgt für die Ausgabe der Leiste, wie im Bild gezeigt, und der Controller sorgt dafür, dass Verstellungen durch Mausklicken oder Mausziehen im Model festgehalten werden und zur richtigen Anzeige führen.
Die MVC-Elemente interagieren wie in Abb. 8-5 gezeigt. Trifft ein Ereignis ein, beispielsweise durch das Abschließen eines Eintrags in einer Tabellenzelle durch Betätigen der Enter-Taste, so sorgt der Controller dafür, dass das Model entsprechend geändert wird, und das Model veranlasst die View, dass diese Änderungen unmittelber angezeigt werden. Die View ihrerseits legt fest, welche Ereignisse an den Controller gehen können.

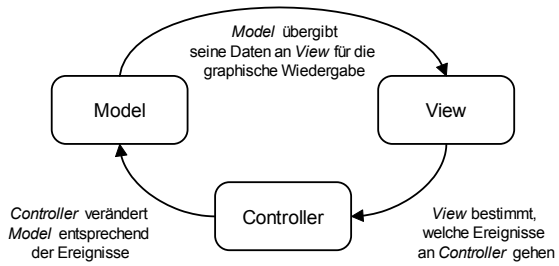


Abb. 8-5: Kommunikation in der MVC-Architektur

Swing verwendet eine vereinfachte Architekturvariante von MVC. In dieser sind View und Controller zu einem Element zusammengezogen, in dem sowohl die Anzeige der GUI-Komponente als auch die Behandlung der Ereignisse erfolgt (Abb. 8-6). Dieses Element heißt auch *UI Delegate*.

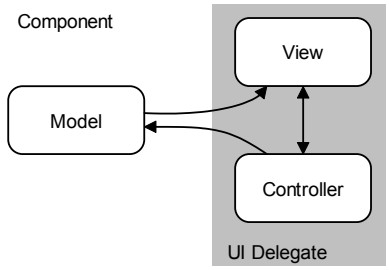


Abb. 8-6: MVC in Swing: Model und UI Delegate

In zwei Beispielen soll das Zusammenspiel von Model und UI Delegate gezeigt werden.

Das *erste Beispiel* zeigt ein GUI mit einer Rollleiste (Swing-Klasse `JScrollBar`) und einem Ausgabefeld. Es wurde deshalb ausgewählt, weil mit knappem Java-Code das Wichtigste aufgezeigt werden kann. (Das zweite Beispiel ist dagegen stärker an dem Anwendungsbeispiel dieses Kapitels orientiert.)
In Abb. 8-7 ist das GUI gezeigt, nämlich eine Rollleiste und ein Anzeigefeld, in dem die Position des Schiebers in der Leiste als Dezimalbruch zwischen `0.0` und `1.0` angezeigt wird. Rechts in der gleichen Abbildung sind die Klassen angegeben, die zuständig für die Daten und das GUI sind, also die Architektur beschreiben.

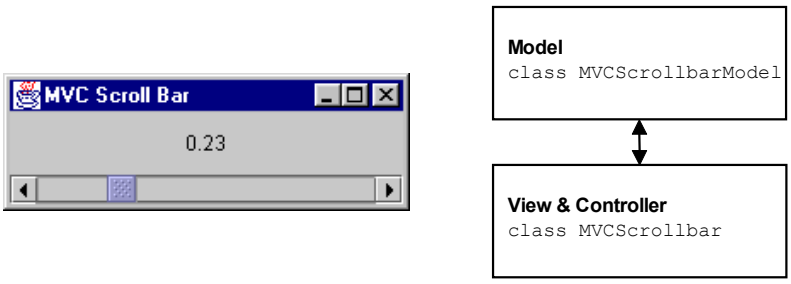


Abb. 8-7: GUI mit Rollleiste und Anzeigefeld (links) sowie seiner Architektur (rechts)

Das Programm beginnt mit den Importen der erforderlichen AWT- und Swing-Packages. Dem schließen sich zwei Klassendeklarationen für Model und UI Delegate an.
Die UI Delegate-Klasse `MVCScrollbar` ist eine Erweiterung der Swing-Klasse `JFrame` und kennzeichnet somit das Programm als Java-Applikation. Aus ihr wird in der `main()`-Methode, also beim Start, ein Objekt erzeugt, in dem beim Instanziieren das Datenmodell die GUI-Elemente Rollleiste (`JScrollBar`) und Ausgabefeld (`Label`) erzeugt und in einem `BorderLayout` plaziert. Im Unterschied zu Swing-Klassen wie `JTable` oder `JList` wird nicht automatisch ein Datenmodell mit definiert. Das Datenmodell muss explizit an das `JScrollBar`-Objekt gebunden werden (`jbar.setModel(model)`). Es

kann entweder das in Swing enthaltene Standardmodell sein oder ein eigenes, das z.B. auf der Erbschaft des Standardmodells beruht. Außerdem wird ein Event-Listener in einer anonymen Klasse definiert. In ihm werden die Veränderungen des Schiebers als Fließkommazahl auf dem `Label`-Objekt `anzeige` ausgegeben.

Programm 8-1: Java-Code für den GUI in Abb. 8-7 (Rollleiste)

```
// Programm 8-1:  BSP_ROOT/meinewebapp/WEB-INF/classes/oraundswing/MVCScrollbar.java
import java.awt.*;         import javax.swing.*;
import java.awt.event.*;

// UI Delegate (View und Controller)
class MVCScrollbar extends JFrame {
  MVCScrollbarModel model = new MVCScrollbarModel();
  JScrollBar jbar;
  Label anzeige = new Label("", Label.CENTER);
```

```
  public MVCScrollbar() {
    super("MVC Scroll Bar");
    jbar = new JScrollBar(JScrollBar.HORIZONTAL);
    jbar.setModel(model);
    jbar.addAdjustmentListener(new AdjustmentListener() {
      public void adjustmentValueChanged(AdjustmentEvent e) {
        anzeige.setText("" + model.getDouble()); } });
    getContentPane().setLayout(new BorderLayout());
    getContentPane().add("Center", anzeige);
    anzeige.setText("" + model.getDouble());
    getContentPane().add("South", jbar);
    setSize(300, 100);
    setVisible(true);
  }
  public static void main(String[] args) {
    new MVCScrollbar();
  }
}                                  // Ende class MVCScrollbar
```

Das Datenmodell `MVCScrollbarModel` basiert auf dem Standardmodell `DefaultBoundedRangeModel`. Als Methode zusätzlich zu den geerbten wird `getDouble()` deklariert. In ihr wird die aktuelle Schieberstellung in eine Fließkommazahl zwischen `0.0` und `1.0` abgebildet. Die dazu erforderlichen Werte werden dem Standardteil des Datenmodells entnommen (`getValue()`, `getMinimum()` und `getMaximum()`).

```
// Model
class MVCScrollbarModel extends DefaultBoundedRangeModel {
  double getDouble() {
    return ((double) getValue())/(getMaximum() - getMinimum());
  }
}                              // Ende class MVCScrollbarModel
```

Im zweiten Beispiel wird eine Swing-Tabelle gezeigt (Abb. 8-8). Wieder ist das Programm in zwei Klassen geteilt, eine für das Datenmodell und die andere für den UI Delegate (Controller & View). Das Beispiel ist zugleich Modell für die danach folgende Datenbankanwendung.

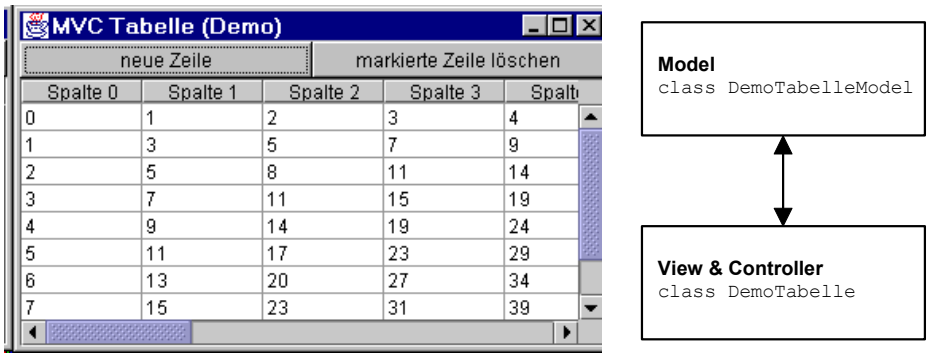


Abb. 8-8: Beispiel für eine Swing-Tabelle

Auch dieses Beispielprogramm ist als Java-Applikation konzipiert, erkenntlich an der Erbschaft von `JFrame`. Nach den nötigen Klassenimporten wird ein Datenmodell erzeugt und in der Instanzvariablen `m` festgehalten, danach wird dieses Datenmodell an die Tabelleninstanz `table` gebunden. Nach Aufbau der GUI-Elemente (Panel, Knöpfe, Tabelle) werden einige Tabelleneigenschaften festgelegt, nämlich dass keine Spalten selektiert werden können, dass genau eine Zeile markierbar ist und dass die Zellenbreiten nicht an die Fensterbreite angepasst werden (`table.`*`methode`*`()` in Fettschrift).

Programm 8-2: Java-Code für den MVC-GUI in Abb. 8-8 (Tabelle)

```
// Programm 8-2:  BSP_ROOT/meinewebapp/WEB-INF/classes/oraundswing/DemoTabelle.java
import java.awt.*;       import javax.swing.*;
import java.awt.event.*; import javax.swing.table.*;

// UI Delegate (View und Controller)
class DemoTabelle extends JFrame {
  DemoTabelleModel m = new DemoTabelleModel();
  JTable table = new JTable(m);
```

Es folgen die Definitionen von Event-Listenern für die Knöpfe "neue Zeile" und "markierte Zeile löschen". Der erste Listener sorgt für den Aufruf der Methode `neueZeile()` im Datenmodellobjekt m und der zweite für den Aufruf von `löscheZeile()`. In beiden Listenern wird die Zeilenmarkierung auf die letzte Zeile eingestellt, und zwar durch Aufruf der Methode `setRowSelectionInterval()`, mit `getRowCount()-1` als Parameter zur Positionierung der Markierung.
Der dritte Event-Listener sorgt für eine geordnete Terminierung der Anwendung, falls das Fenster über dessen Navigationsschaltflächen geschlossen wird. (Im vorangehenden Rollleistenbeispiel ist dafür noch nicht gesorgt. Die Folge ist, dass bei Klicken in die Navigationsschaltflächen zwar das Fenster geschlossen, aber das Programm nicht terminiert wird.)

```
  public DemoTabelle() {
    super("MVC Tabelle (Demo)");
    Panel oben = new Panel(new GridLayout(0, 2));
    Button neuZeile = new Button("neue Zeile");
    Button delZeile = new Button("markierte Zeile löschen");
    getContentPane().setLayout(new BorderLayout());
    getContentPane().add("North", oben);
    oben.add(neuZeile);
    oben.add(delZeile);
    getContentPane().add("Center", new JScrollPane(table));
    table.setColumnSelectionAllowed(false);
    table.setSelectionMode(ListSelectionModel.SINGLE_SELECTION);
    table.setAutoResizeMode(JTable.AUTO_RESIZE_OFF);
    neuZeile.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent e) {
        m.neueZeile();
        table.setRowSelectionInterval(m.getRowCount()-1, m.getRowCount()-1); }});
    delZeile.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent e) {
        m.löscheZeile(table.getSelectedRow());
        table.setRowSelectionInterval(m.getRowCount()-1, m.getRowCount()-1); }});
    addWindowListener(new WindowAdapter() {
      public void windowClosing(WindowEvent e) { System.exit(0); } });
    setSize(500, 400);
    setVisible(true);
  }
```

Wie üblich bei Java-Applikationen, dient `main()` dem Start des GUI:

```
  public static void main(String[] p) {  new DemoTabelle(); }
}
```

Das Datenmodell des Programms basiert auf dem Standardmodell für Swing-Tabellen, nämlich der Klasse `DefaultTableModel`. Von diesem Standardmodell werden die Methoden `isCellEditable()` und `setValueAt()` überschrieben. In `isCellEditable()` kann für jede individuelle Zelle festgelegt werden, ob ihr Inhalt verändert werden darf oder nicht. Aufrufer ist die MVC-Komponente, hier die Tabelle selbst. Ihr wird auf Aufruf hin durch die Rückgabe von `true` ein Ja und von `false` ein Nein signalisiert, ob die gewählte Zelle editierbar ist oder nicht. Im Beispiel wird so die erste Spalte für nicht editierbar erklärt.

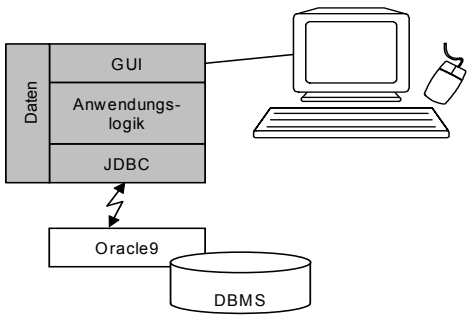
```
// Model
class DemoTabelleModel extends DefaultTableModel {
  int zähler;
  static final int cols = 16, rows = 10;
  public DemoTabelleModel() {
    super(0, cols);
    for (int i = 0; i < rows; i++) neueZeile();
    Object[] namen = new Object[cols];
    for (int i = 0; i < cols; i++) namen[i] = "Spalte " + i;
    setColumnIdentifiers(namen);
  }
```

```
  public boolean isCellEditable(int row, int col) {
    if (col == 0) return false;
    else return super.isCellEditable(row, col);
  }
  public void setValueAt(Object o, int row, int col) {
    try {
      Integer.parseInt((String) o);
      super.setValueAt(o, row, col);
    }
    catch(NumberFormatException e) {}
  }
```

Zum Datenmodell gehören auch zwei Methoden zum Anfügen und zum Löschen einer wählbaren Zeile. Die Methode `neueZeile()` wird im Event-Listener des Knopfes mit der Aufschrift "neue Zeile" aufgerufen, `löscheZeile()` in dem des Knopfes "markierte Zeile löschen". Neue Zeilen werden am Ende der Tabelle angefügt. Jede neue Zeile wird mit aufsteigenden Werten versehen, die mit dem aktuellen Wert der Variablen `zähler` beginnen. Mit jeder neuen Zeile wird diese Variable um `1` erhöht. Die Zeilen der ersten Spalte sind also mit der `1` beginnend durchnumeriert.

```
  void neueZeile() {
    Object[] temp = new Object[cols];
    for (int i = 0; i < cols; i++)
      temp[i] = String.valueOf((zähler + 1) * (i + 1) - 1 );
    addRow(temp);
    zähler++;
  }
  void löscheZeile(int row) {
    if (row < 0 || row >= getRowCount()) return;
    removeRow(row);
  }
}                              // Ende class DemoTabelleModel
```

8.3 Der Client



Das vollständige Clientprogramm entsprechend Abb. 8-3 auf Seite 414 umfasst bei moderatem Bedienkomfort und hinreichender Behandlung von Ausnahmesituationen und Fehlern ca. 800 Codezeilen (Kommentarzeilen nicht eingerechnet). Es besteht aus den sieben dort gezeigten MVC-Komponenten, die sich allerdings sehr stark ähneln. Es genügt also vollauf, das Programm einer einzelnen Komponente stellvertretend für alle auszuführen. Abb. 8-9 zeigt das GUI dieser Musterkomponente, und Abb. 8-10 deren Architektur. In der angezeigten Tabelle kann eine Zeile ausgewählt und können ihre Zellen bearbeitet werden, die erste und die beiden letzten ausgenommen. Die Zellen in den Spalten "Kursbezeichnung" und "Zeit" können direkt, die in den Spalten "dcode" und "Typ" durch Auswahl aus einem Popup-Menü geändert werden. Persistent werden die Daten dann nach Drücken des Knopfes "speichern"; bis dahin können die Änderungen jederzeit wieder rückgängig gemacht werden, und zwar durch Drücken des "refresh"-Knopfes. Neue Zeilen werden mit "neu" am Ende der Tabelle angelegt, und eine markierte Zeile wird mit "löschen" gelöscht. Persistent wird auch eine neue Zeile erst dann, wenn sie gespeichert wird, während Löschungen nach dem Bejahen einer Kontrollfrage unmittelbar in der Datenbanktabelle ausgeführt werden.
In einer vollständigen Fassung würde Drücken von "Kursteilnehmer" zu einem GUI führen, in dem zum Kurs in der markierten Zeile Kursteilnehmer aus der Tabelle Personen ausgewählt werden könnten. Im Beispielprogramm wird dieser Knopf für die Wiederherstellung des Urzustandes in allen Tabellen verwendet (Klasse `InitialisiereTabellen`, siehe `http://vsite.de`).

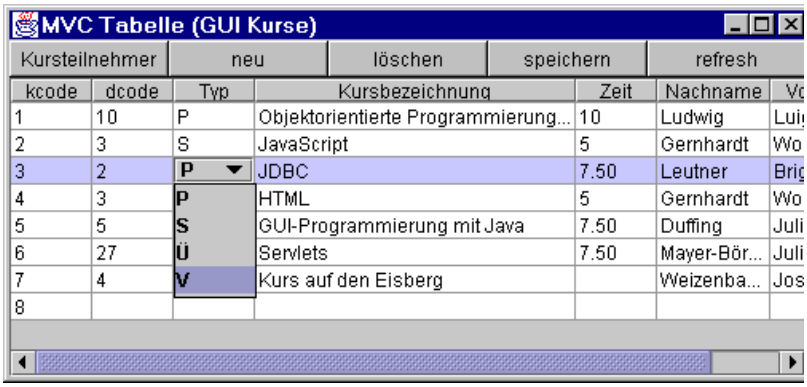


Abb. 8-9: Das Clientfenster

Das folgende Bild zeigt neben der Gliederung des Programms in die beiden MVC-Bestandteile vor allem auch die Aufteilung des Datenmodells (Model) in einen verbindungsunabhängigen Teil und einen "Connector".

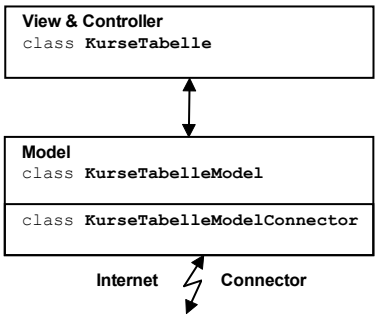


Abb. 8-10: MVC-Architektur des Client

Der Connector beinhaltet die Kommunikation mit der Datenbank sowie Rudimente einer Applikationslogik. Er könnte ohne allzu große Probleme in eine Middleware verlagert werden.
Der UI Delegate des Clientprogramms hat die folgende Struktur:

```
// UI Delegate (View & Controller)
class KurseTabelle extends JFrame
{
   public KurseTabelle(            // Konstruktor
      GUI zusammenstellen und starten (AWT- und Swing-Komponenten)
      Eventlistener einrichten für:
      - neue Zeile anfügen
      - eine Zeile löschen
      - Anzeige aktualisieren
      - Änderungen in einer Zeile speichern
      - Client schließen
   Die Instanzmethoden dafür sind:
      void neueZeile(Object[] o) {}      Array als Zeile am Ende einfügen
      void neueZeile() {}                dito Leerzeile
      void löscheZeile(int row) {}       die angegebene Zeile löschen
      void speichereZeile(int row) {}    neue Werte einer Zeile speichern
      void refresh() {}                  Tabellenwerte aktualisieren
   Startmethode:
      public static void main(String[] args) {}
}
```

Der Java-Code des UI Delegate beginnt mit den erforderlichen Importen der AWT-, JDBC- und Swing-Klassen,

Programm 8-3a: GUI des Anwendungsprogramms (View & Control)

```
// Programm 8-3a:  BSP_ROOT/meinewebapp/WEB-INF/classes/oraundswing/KurseTabelle.java
import java.awt.*;          import javax.swing.*;
import java.awt.event.*;    import javax.swing.table.*;
import java.sql.*;          import javax.swing.event.*;
```

gefolgt von der Deklaration der UI Delegate-Klasse und Instanzvariablen für Datenmodell und Swing-Tabelle.

```
// UI Delegate (View und Controller)
class KurseTabelle extends JFrame {

  KurseTabelleModel m = new KurseTabelleModel();

  JTable table = new JTable(m);
```

Im Konstruktor `KurseTabelle()` des Delegate wird der GUI mit AWT- und Swing-Elementen aufgebaut,

```
  public KurseTabelle() {
    super("MVC Tabelle (GUI Kurse)");
    Panel oben = new Panel(new GridLayout(1, 5));
    Button reset = new Button("Reset");
    Button neuZeile = new Button("neu");
    Button delZeile = new Button("löschen");
    Button speicher = new Button("speichern");
    Button refresh  = new Button("refresh");
    getContentPane().setLayout(new BorderLayout());
    getContentPane().add("North", oben);
    oben.add(reset);
    oben.add(neuZeile);
    oben.add(delZeile);
    oben.add(speicher);
    oben.add(refresh);
    getContentPane().add("Center", new JScrollPane(table));
    table.setColumnSelectionAllowed(false);
```

und die Tabelle wird so konfiguriert, dass nicht mehr als eine Tabellenzeile angewählt werden kann, die Zellenbreiten sich nicht automatisch an die sichtbare Fensterbreite anpassen und die Tabellenzellen individuelle Breiten erhalten.

```
    table.setSelectionMode(ListSelectionModel.SINGLE_SELECTION);
    table.setAutoResizeMode(JTable.AUTO_RESIZE_OFF);
    for (int i = 0; i < m.cols; i++)
      table.getColumn(m.colnames[i]).setPreferredWidth(m.colwidths[i]);
```

Sodann wird der im Datenmodell instanziierte Datenbank-Connector initialisiert, das Datenmodell ein erstes Mal mit den aktuellen Datenbankwerten versehen und die Spalten mit den Beschriftungen „dcode“ und „Typ“ mit Popupmenüs (Comboboxen) für die Dateneingabe aus Domänen versehen.

```
    m.con.init();
    refresh();
    combobox(m.dozenten, m.colnames[1]);
    combobox(m.typen, m.colnames[2]);
```

Für die Controller-Funktionen werden insgesamt sechs Event-Listener definiert, fünf für die Knöpfe in der Kopfleiste des GUI und einer (an letzter Stelle), um die Anwendung geordnet abzuschließen. Dabei bewirkt der Knopf mit der Objektreferenz

- `neuZeile` eine neue Tabellenzeile;
- `reset` die Wiederherstellung aller Tabellen in den Originalzustand;
- `delZeile` die Löschung der markierten Zeile der Tabelle;
- `refresh` die Aktualisierung der angezeigten Tabelle mit Daten aus den Datenbanktabellen `Kurse` und `Dozenten` sowie
- `speicher` die Speicherung von Änderungen, die in der markierten Zeile vorgenommen wurden, in der Datenbanktabelle `Kurse`.

```
    neuZeile.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent e) {
        neueZeile(); }});
    reset.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent e) {
        new InitialisiereTabellen();
        refresh(); }});
    delZeile.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent e) {
        löscheZeile(table.getSelectedRow()); }});
```

```
    refresh.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent e) {
        refresh(); }});
    speicher.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent e) {
        speichereZeile(table.getSelectedRow()) ; }});
    addWindowListener(new WindowAdapter() {
      public void windowClosing(WindowEvent e) { System.exit(0); }});
    setSize(500, 400);
    setVisible(true);
  }
```

In der Methode `combobox()` werden `JComboBox`-Elemente mit Einträgen aus einem als Parameter mitgeteilten Array aufgebaut und mit einem passenden Zelleneditor versehen.

```
  private void combobox(Object[] items, String spalte) {
    JComboBox box = new JComboBox();
    for (int i = 0; i < items.length; i++)
      box.addItem(items[i]);
    table.getColumn(spalte).setCellEditor(
                            new DefaultCellEditor(box));
  }
```

Es folgt eine Gruppe von Methoden, die fast immer von Event-Listenern aufgerufen werden, wenn ein Ereignis eintritt, d.h. auf einen der Knöpfe gedrückt wird.

```
  void neueZeile(Object[] o) {
    m.addRow(o);
    table.setRowSelectionInterval(m.getRowCount()-1, m.getRowCount()-1);
  }
  void neueZeile() {
    Object[] temp = new Object[m.cols];
    for (int i = 0; i < m.cols; i++)
      temp[i] = "";
    neueZeile(temp);
  }
  void löscheZeile(int row) {
    if (row < 0 || row >= m.getRowCount())
      return;
    if (JOptionPane.showOptionDialog(this, "Wirklich kcode=" +
             m.getValueAt(row, 0) + " löschen?",
             "KurseTabelle", JOptionPane.DEFAULT_OPTION,
             JOptionPane.WARNING_MESSAGE, null,
             new Object[]{"Ja", "Nein"}, "Nein") != 0) return;
    m.con.delete(m.getValueAt(row, 0));
    m.removeRow(row);
    table.setRowSelectionInterval(m.getRowCount()-1, m.getRowCount()-1);
  }
  void speichereZeile(int row) {
    if ("".equals(m.getValueAt(row, 0)))
      m.con.insert(m, row);
    else
      m.con.update(m, row);
    refresh();
  }
```

`refresh()` sorgt dafür, dass die Swing-Tabellenzellen mit aktuellen Daten aus der Datenbank versorgt und außerdem die Combobox-Einträge für die „dcode“-Spalte erneuert werden.

```
  void refresh() {
    Object[] temp = new Object[m.cols];
    m.con.select();
    m.setNumRows(0);
    while ((temp = m.con.nächste()) != null)
      neueZeile(temp);
    combobox(m.dozenten, m.colnames[1]);
  }
```

Wie üblich, wird das GUI über die `main()`-Methode gestartet.

```
  public static void main(String[] args) {
    new KurseTabelle();
  }
}                                   // Ende Class KurseTabelle
```

Auch für das Datenmodell folgt erst eine Übersicht.

Das Modell ist in zwei Teile gegliedert. Im ersten Teil wird das Standardtabellenmodell `DefaultTableModel` erweitert, und es werden dort die Tabellenwerte als zweidimensionales, die Spaltenüberschriften als eindimensionales Array etc. festgehalten. Mit diesem Teil eng verbunden ist ein zweiter Teil, in dem alle datenbankbezogenen Eigenschaften und Operationen enthalten sind, d.h. der gleichsam als Verbindungsstück zur Datenbank dient. (Dieser Teil könnte geschlossen in eine Middleware verlagert werden.)

```
// Datenmodell (Model)
class KurseTabelleModel extends DefaultTableModel
{
 Tabellendaten einschließlich Spaltenwerten und -überschriften, Editierbarkeit etc.
   public KurseTabelleModel()
Instanzen: Eingelesene Tabellen, Tabellenüberschriften, Editierbarkeit etc.
   public boolean isCellEditable(int row, int col)
Editierbarkeit einer Zelle überprüfen/festlegen
}
```

```
// Datenbank-Connector
class KurseTabelleModelConnector
{
   static {}          statischer Konstruktor zur DB-Verbindungsaufnahme
   public KurseTabelleModelConnector(KurseTabelleModel m)
                      Konstruktor des Datenbank-Connectors
   void select()      Tabelle/View selektieren und Resultset speichern
   Object[] next()    nächste Zeile aus Resultset lesen und im Model speichern
   void update(KurseTabelleModel m, int row)
                      vorbereitetes UPDATE mit Model-Werten versehen und ausführen
   void insert(KurseTabelleModel m, int row)
                      vorbereiteten gespeicherten Prozeduraufruf mit Parameterwerten versehen und ausführen
   void delete(Object kcode)
   void init()        SELECT, UPDATE, INSERT und DELETE vorbereiten
}
```

Das Datenmodell `KurseTabelleModel` ist eine Erweiterung des Standardtabellenmodells `DefaultTableModel`. Als Instanzvariable ist dort eine Art Spiegelbild der View `Kurse LEFT JOIN Dozenten` (`Kurse = Dozenten (+)` bei Oracle) gehalten, außerdem sind dort die Spaltennamen für Überschriften und die Datenbank, Informationen über die Editierbarkeit von Spalten, die Domäne für die Spalte "Typ" usw. zu finden.

Programm 8-3b: Datenmodell des Anwendungsprogramms (Model)

```
// Programm 8-3b  (gleiche Datei wie Teil a)
// Model
class KurseTabelleModel extends DefaultTableModel {
  KurseTabelleModelConnector con;
  String[] colnames = new String[] {"kcode","dcode","Typ",
                                    "Kursbezeichnung","Zeit","Nachname","Vorname"};
  String[] columns = new String[]  {"kcode","Kurse.dcode","typ",
                                     "bezeichnung","zeit","nachname","vorname"};
  boolean[] editable = {false, true, true, true, true, false, false};
  int[] colwidths = {50, 50, 50, 200, 50, 70, 70};
  static final int cols = 7;
  Object[] dozenten;
  Object[] typen = new Object[] {"P", "S", "Ü", "V"};
```

Im Konstruktor des Datenmodells wird der Datenbank-Connector erzeugt und die Referenz auf ihn in der Instanzvariablen `con` festgehalten; danach wird die Tabelle beschriftet (`setColumnIdentifiers(colnames)`).
Die Methode `isCellEditable()` überschreibt die aus dem Standardmodell geerbte. Damit werden über die `false`-Werte im Array `editable` Spaltenwerte als nicht veränderlich erklärt.

```
  public KurseTabelleModel() {
    super(0, cols);
    con = new KurseTabelleModelConnector(this);
    setColumnIdentifiers(colnames);
  }
  public boolean isCellEditable(int row, int col) {
    if (!editable[col]) return false;
    else return super.isCellEditable(row, col);
  }
}                                // Ende Class KurseTabelleModel
```

Die Klasse `KurseTabelleModelConnector` beginnt mit Klassenvariablen für die Daten der Datenbankverbindung, gefolgt von Instanzvariablen für SQL-Anweisungen in Form von Strings und JDBC-`Statement`-Objekten.

```
// Datenbank-Connector
class KurseTabelleModelConnector {
  static String treiber = "oracle.jdbc.driver.OracleDriver";
  static String jdbcurl = "jdbc:oracle:thin:@orcl.vsite.de:1521:Kurse";
  static String uid = "system", pwd = "manager";
  static Connection c;
  String sqlsel; Statement sel;
  String sqlupd; PreparedStatement upd;
  String sqldel; PreparedStatement del;
  String sqlins; CallableStatement ins;
  String sqldoz; Statement doz;
  ResultSet rsel, rdoz;
  KurseTabelleModel m;
```

Im statischen Initialisierer `static {}` wird nach dem Laden der Klasse die Verbindung zur Datenbank hergestellt, und bei der Erzeugung einer Instanz wird im Konstruktor das Model-Objekt `m` festgehalten, womit auch in diese Richtung die Verbindung hergestellt ist.

```
static {                          // statischer Initialisierer
  try {
    Class.forName(treiber);
    c = DriverManager.getConnection(jdbcurl, uid, pwd);
  }
  catch (Exception ex) { ex.printStackTrace(); System.exit(0); }
}
public KurseTabelleModelConnector(KurseTabelleModel m) { this.m = m; }
```

In `select()` werden die Daten der View `Kurse LEFT JOIN Dozenten` (bei Oracle `Kurse = Dozenten (+)`) in einem `ResultSet`-Objekt bereitgestellt, die dann mit der folgenden Methode `nächste()` Zeile für Zeile eingelesen werden können (`nächste()` wird in der Methode `refresh()` verwendet). In `select()` werden außerdem alle Dozentenschlüssel als Wertebereich für die `dcode`-Spalte in der View eingelesen.

```
void select() {
  System.out.println("SQL " + sqlsel);
  try {
    rsel = sel.executeQuery(sqlsel);
    rdoz = doz.executeQuery(sqldoz);
    m.dozenten = null;
    while (rdoz.next()) {
      int x = m.dozenten == null ? 0 : m.dozenten.length;
      Object[] temp = new Object[x + 1];
      for (int i = 0; i < x; i++) {
        temp[i] = m.dozenten[i]; }
      temp[x] = rdoz.getString(1);
      m.dozenten = temp;
    }
  }
  catch (Exception ex) { ex.printStackTrace();}
}
Object[] nächste() {
  Object[] temp = new Object[m.cols];
  try {
    if (!rsel.next()) return null;
    for (int i = 0; i < m.cols; i++)
      temp[i] = rsel.getString(i + 1);
    return temp;
  }
  catch (Exception ex) { ex.printStackTrace();}
  return null;
}
```

In der `update()`-Methode wird die vorbereitete SQL-Anweisung mit Parametern versehen (`setString()`), die der markierten Zeile entsprechend dem Datenmodell entnom-

men (`getValueAt()`) und danach ausgeführt werden. (Zur Vorbereitung der SQL-Anweisungen siehe weiter unten bei `init()`.)

```
void update(KurseTabelleModel m, int row) {
  try {
    System.out.println("SQL " + sqlupd);
    upd.setString(5, (String) m.getValueAt(row, 0));
    upd.setString(1, (String) m.getValueAt(row, 1));
    upd.setString(2, (String) m.getValueAt(row, 2));
    upd.setString(3, (String) m.getValueAt(row, 3));
    upd.setString(4,((String) m.getValueAt(row, 4)).replace('.', ','));
    upd.execute();
  }
  catch (Exception ex) { ex.printStackTrace(); }
}
```

Zu beachten und etwas gewöhnungsbedürftig bei der Verwendung des GUI ist, dass Änderungen, die noch nicht durch Eingabetaste o.ä. bestätigt wurden, mit `getValue()` nicht verfügbar sind, d.h. `getValue()` liefert den vorangehenden, noch unmodifizierten Wert. Für den Bediener heißt das, dass er Eingaben über die Tastatur erst mit der Eingabe- oder Tabulatortaste bestätigen muss, bevor er mittels des Knopfes „speichern“ die Änderungen persistent machen kann.

Ähnlich wird in `insert()` verfahren. Die Einfügung wird durch eine gespeicherte Funktion vorgenommen, die den Schlüssel `dcode` festlegt und als Ergebnis zurückliefert. Vor dem Aufruf wird der Typ des entsprechenden Rückgabeparameters mit `registerOutParameter()` registriert.

```
void insert(KurseTabelleModel m, int row) {
  try {
    System.out.println("SQL " + sqlins);
    ins.registerOutParameter(1, Types.INTEGER);
    ins.setString(2, (String) m.getValueAt(row, 1));
    ins.setString(3, (String) m.getValueAt(row, 2));
    ins.setString(4, (String) m.getValueAt(row, 3));
    ins.setString(5, (String) m.getValueAt(row, 4));
    ins.execute();
  }
  catch (Exception ex) { ex.printStackTrace(); }
}
```

Mit `delete()` wird die markierte Tabellenzeile aus der Datenbank gelöscht,

```
void delete(Object kcode) {
  try {
    System.out.println("SQL " + sqldel);
    del.setInt(1, Integer.parseInt((String)kcode));
    del.execute();
  }
  catch (Exception ex) { ex.printStackTrace(); }
}
```

und mit `init()` werden alle SQL-Anweisungen vorbereitet: Die `SELECT`s als einfache Anweisungen, d.h. `Statement`-Objekte, `UPDATE` und `DELETE` als vorbereitete Anweisungen (`PreparedStatement`-Objekte) und `INSERT` als gespeicherte Funktion, d.h. als `CallableStatement`-Objekt.

```
  void init() {
    try {
   // SELECT
      sel = c.createStatement();
      sqlsel = "SELECT ";
      for (int i = 0; i < m.cols; i++)
        sqlsel += (i == 0 ? "" : ",") + m.columns[i];
      sqlsel += " FROM Kurse, Dozenten " +
                " WHERE Kurse.dcode = Dozenten.dcode (+)" +
                " ORDER BY kcode";
      doz = c.createStatement();
      sqldoz = "SELECT dcode,nachname,vorname FROM Dozenten";

   // UPDATE
      sqlupd = "UPDATE Kurse " +
               "SET dcode=?, typ=?, bezeichnung=?, zeit=? " +
               "WHERE kcode=?";
      upd = c.prepareStatement(sqlupd);

   // INSERT
      String proz =
          "CREATE OR REPLACE FUNCTION InsertKurse \n" +
          "(dc IN INTEGER, typ IN STRING, " +
          " tit IN STRING, zeit IN INTEGER) \n" +
          " RETURN INTEGER IS \n t NUMBER; \n" +
          " BEGIN\n SELECT max(kcode) INTO t FROM Kurse;\n" +
          " IF t IS NULL THEN t := 0; END IF;" +
          " INSERT INTO Kurse(kcode, dcode, typ," +
          " bezeichnung, zeit)\n" +
          " VALUES(t+1,dc,typ,tit,zeit); \n" +
          " RETURN t+1; \n END;";
      c.createStatement().execute(proz);
      sqlins = "{? = call InsertKurse(?, ?, ?, ?)}";
      ins = c.prepareCall(sqlins);

   // DELETE
      sqldel = "DELETE FROM Kurse WHERE kcode=?";
      del = c.prepareStatement(sqldel);
    }
    catch (Exception ex) {
      ex.printStackTrace(); System.exit(0);
    }
  }
}                    // Ende Class KurseTabelleModelConnector
```

Der besseren Lesbarkeit wegen ist die gespeicherte Funktion für die Einfügeoperation als Programm 8-3c wiedergegeben. Es ist in PL/SQL geschrieben.
Die Funktion beginnt mit der Deklaration ihres Kopfes, d.h dem Namen der Funktion und ihrer formalen Parameter. Für jeden der formalen Parameter sind der Name, die Art der Übergabe und der Datentyp definiert. Es folgen die Typfestlegung für den Rückgabewert

und die Deklaration lokaler Variabler. Der Funktionskörper ist in einem `BEGIN...END;`-Block eingeschlossen, in dem die eigentliche Funktionalität enthalten ist.

Programm 8-3c: Gespeicherte Funktion zum Einfügen einer Zeile

```
-- Programm 8-3c:  Oracle9 Stored Function
FUNCTION InsertKurse (dc IN INTEGER,
        typ IN STRING, tit IN STRING, zeit IN INTEGER)
RETURN INTEGER IS
t NUMBER;
BEGIN
  SELECT max(kcode) INTO t FROM Kurse;
  IF t IS NULL THEN t := 0; END IF;
  INSERT INTO Kurse(kcode, dcode, typ, bezeichnung, zeit)
         VALUES    (t+1,dc,typ,tit,zeit);
  RETURN t+1;
END;
```