Tutorium zur Programmierung in Java

1. Überblick und Lernziele

Zusammenfassung

Dieses Tutorium verfolgt den Zweck, Java zur Programmierung von einfachen Anwendungen anzuwenden. Dabei wird Grundwissen über die objektorientierte Programmierung und Grundkonzepte der Sprache **Java** vorausgesetzt. Leser, die schon über gute Java-Kenntnisse verfügen, können dieses Tutorium als eine Wiederholung verstehen und in ihr Lernprogramm aufnehmen. Trifft dies nicht zu, ist der Aufwand zum Durcharbeiten des Tutoriums aufgrund der Übungsorientierung sehr zeitaufwändig. Es ist unerlässlich, dass der Leser konkret am Rechner das Fallbeispiel durcharbeitet. Auf der Webseite ist zwar der gesamte Programmcode mitgeliefert, eine Beschränkung auf ein rein lesendes Durcharbeiten bringt jedoch wenig Lernerfolg. Dieser Programmcode sollte ausschließlich als Referenz dienen bzw. als Quelle, um ausgewählte Code-Teile zu übernehmen. In einer ersten Stufe wird die Installation der **Entwicklungsumgebung** *NetBeans* und des **Datenbankmanagementsystems** *MySQL* beschrieben. Auf dieser Basis wird die Fachlogik mit Java umgesetzt und ohne grafische Oberfläche und Datenbankanbindung getestet. In einem zweiten Schritt wird eine grafische Oberfläche mit dem **Swing-Framework** entwickelt, dabei wird auf die Funktionalität eines grafischen Editors von *NetBeans* zurückgegriffen. Im dritten Schritt wird die **Datenhaltung** mit dem relationalen Datenbankmanagementsystem *MySQL* implementiert. Dabei wird zuerst auf die grundsätzliche Fragestellung der Verbindung der Java-Umgebung mit einer relationalen Datenbank eingegangen. In diesem Zusammenhang werden Java Database Connectivity (**JDBC**) und Datenbank-Treiber angesprochen. Danach werden sowohl das Lesen von der Datenbank und das Schreiben auf die Datenbank einschließlich einfacher Möglichkeiten der Transaktionssteuerung am Anwendungsbeispiel angewendet. Mit dem letzten Abschnitt soll insbesondere unter Aspekten des **Architektur-Entwurfs** gezeigt werden, dass eine strikte Trennung der einzelnen Schichten sehr viel Sinn machen kann. Es wird unterstellt, dass ein Teil der Anwendungsfälle, die bisher nur als Desktop-Anwendung zur Verfügung standen, nun auch über das Internet ausgeführt werden können. Dabei zeigt sich, dass an den Fachklassen und der Datenhaltung grundsätzlich nichts geändert werden muss. Bei dieser Gelegenheit lernen Sie die **Web-Technologien** Java Server Pages (JSPs) und Servlets kennen und wenden diese praktisch an. Hintergrundwissen zu diesen Technologien wird grundsätzlich nicht vermittelt, sondern ist bei Bedarf in einschlägigen Fachbüchern nachzulesen. (Quellenverweise beziehen sich auf das zugehörige Buch bzw. Publikationen, die im Literaturverzeichnis des Buches aufgeführt sind.)

Wichtige Teilgebiete sind:

- Installation der Entwicklungsumgebung und eines relationalen Datenbankmanagementsystems sowie deren grundsätzliche Charakterisierung
- Anwendung von Java zur Programmierung einfacher Klassen
- Anwendung des Java-Swing-Frameworks zur Erstellung einer grafischen Benutzeroberfläche
- Anwendung des Datenbank-Frameworks Java Database Connectivity (JDBC) zur Anbindung eines relationalen Datenbankmanagementsystems
- Anwendung von Java Server Pages (JSPs) und Servlets zur Entwicklung einer einfachen Web-Anwendung

Lernziele

Der Leser:

- kann die integrierte Entwicklungsumgebung *NetBeans* zum Programmieren relativ einfacher Programme einsetzen
- kann einfache Sprachelemente von Java zur Implementierung von Klassen anwenden
- kann einfache Dialogschnittstellen mit Swing programmieren
- kann eine relationale Datenbank zur Datenhaltung von Objekten verwenden
- bekommt eine erste Vorstellung von Technologien zum Entwickeln einer Web-Anwendung
- kann eine einfache Web-Anwendung mit JSPs und Servlets entwickeln

2. Grundlagen zur Entwicklungsumgebung

Für alle in diesem Buch durchzuführenden Programmierübungen wollen wir eine integrierte Entwicklungsumgebung (IDE – Integrated Development Environment) verwenden. Im Java-Umfeld dominieren derzeit zwei Open-Source Entwicklungsumgebungen. Das sind zum einen *Eclipse* (vgl. http://www.eclipse.org/) und zum anderen ***NetBeans*** (vgl. http://www.netbeans.org). Daneben gibt es eine Vielzahl kommerziell vertriebener IDEs. Im Rahmen dieses Buches werden wir *NetBeans* verwenden, das von der Firma *Sun*, dem Entwickler von Java, stammt. Damit soll keine Wertung hinsichtlich der Qualität der IDEs vorgenommen werden (ein Produktvergleich findet sich bei Wunderlich, 2004, S. 33 ff.).

Von der Sun-Website (http://java.sun.com/javase/downloads/index.jsp) ist "JDK 6 with NetBeans 5.5" herunterzuladen. Es handelt sich um die ausführbare Datei (jdk-6-nb-5_5-win.exe). Diese beinhaltet sowohl das Java 2 Standard Edition (J2SE)

Development Kit in der Version 6.0 als auch die Entwicklungsumgebung NetBeans in der Version 5.5. Die Rechnervoraussetzungen sind ein Pentium III Prozessor mit 500 MHz und 512 MB Arbeitsspeicher, auf der Festplatte sollten 850 MB zur Verfügung stehen. Die Installation wird einfach durch Doppelklick auf den Dateinamen im Explorer gestartet, die Lizenzbedingungen sind zu akzeptieren, ansonsten sollten die vorgeschlagenen Standardeinstellungen akzeptiert und jeweils durch das Betätigen der Schaltfläche Next bestätigt werden. Die Installation benötigt zwischen fünf und zehn Minuten und wird mit einem Klick auf die Schaltfläche Finish beendet. Auf dem Arbeitsplatz (Desktop) wird automatisch eine NetBeans-Schaltfläche installiert. Durch Doppelklick wird *NetBeans* gestartet. Auf der Willkommensseite werden allgemeine Hilfestellungen angeboten. Zur Vorbereitung der Entwicklung von Java-Programmen muss ein so genanntes Projekt angelegt werden (File → New Project). Im Dialog New Project wird die Kategorie General und die Projektart Java Application ausgewählt. Über die Next-Schaltfläche gelangen Sie in den Schritt 2, in dem Name und Speicherort festgelegt werden. Als Projektnamen geben Sie JavaTutorial ein, im Eingabefeld Project Location geben Sie c:\JavaProjekte ein. Mit der Schaltfläche Finish beenden Sie den Dialog. Im linken Teil des Dialogs finden Sie ein Explorer-Fenster mit dem Projekteintrag JavaTutorial. Durch Erweitern der Baumstruktur (JavaTutorial → Source Packages → javatutorial →Main.java) gelangen Sie zu einer automatisch generierten Klasse mit dem Namen Main. Diesen Inhalt finden Sie natürlich nun auch im Windows-Dateisystem. Mit dem Windows-Explorer finden Sie im Laufwerk c unter JavaProjekte den Ordner JavaTutorial mit der Unterordnerstruktur src, nbproject, test und build. Im Ordner src finden Sie den Ordner javatutorial und dort den Quellcode in der Datei Main.java.

Durch einen Klick auf Main im *NetBeans* Explorer sehen Sie im rechten Teil des Dialogs (Editor) auch schon den Quellcode der Klasse *Main*, welche auch eine *main*-Methode beinhaltet. Die Klasse *Main* wurde automatisch dem Java-Paket *javatutorial* zugewiesen. Im Editor sehen Sie auf der linken Seite einen blauen Längsbalken. Positionieren Sie die Maus in diesen Balken, mit rechter Maustaste wird Ihnen angeboten *Show Line Numbers*. Dies zu aktivieren ist sehr hilfreich. Wenn Sie nun im Editor in der *main*-Methode nach dem Kommentar `//TODO code application logic here` die Anweisung:

`System.out.println("Mein erster Programmierversuch mit NetBeans!");` eingeben, haben Sie schon das erste Java-Programm mit der *NetBeans* IDE geschrieben. Sie haben bei der Eingabe vielleicht gleich gemerkt, wie Sie *NetBeans* unterstützt. Bei der Eingabe von `System.` wird Ihnen durch die Funktion *code completion* vorgeschlagen, was eingebbar ist, also z.B. `out`. Weiterhin merken Sie, dass der Editor

gleich eine dynamische Code-Überprüfung durchführt. Wenn ein Fehler in der Anweisung ist, wird die Zeile mit einer roten Wellenlinie unterstrichen und am linken Rand wird ein rotes Fehlersymbol angezeigt. Wenn Sie mit dem Cursor auf dieses Symbol oder die fehlerhafte Zeile gehen wird auch ein Fehlerhinweis angezeigt. Probieren Sie es einfach mal aus, indem Sie das abschließende Semikolon vergessen.

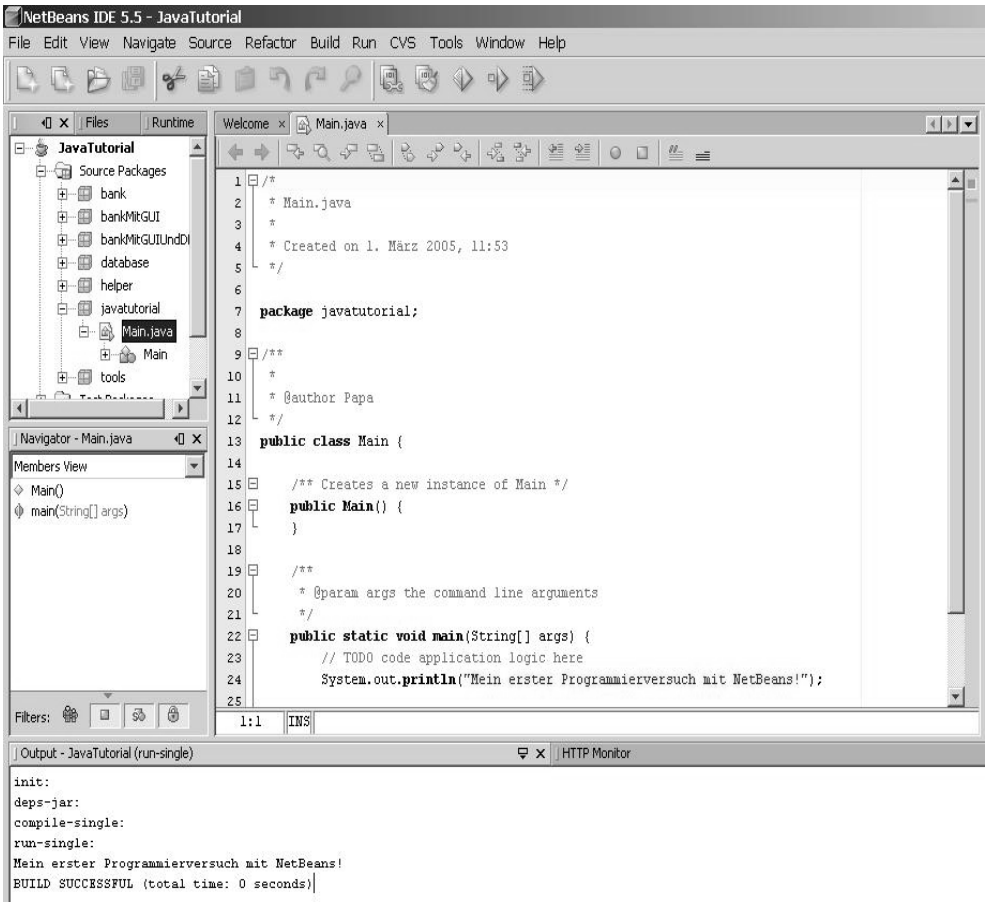

Abb. 1: NetBeans-IDE

Wenn Sie nun im *NetBeans* Explorer auf den Eintrag *Main.java* gehen und mit der rechten Maustaste das Umgebungsmenü anzeigen, dann können Sie die Klasse übersetzen (*Compile File*). Dadurch wird eine *Main.class*-Datei erzeugt. Den Speicherort können Sie im Output-Fenster (Konsole), das im unteren Teil des Dialogs

aufgeht, ablesen. Wenn Sie die Menü-Option *Run File* auswählen, wird die Klasse ausgeführt und es erscheint im Output-Fenster die Ausgabe unseres ersten Programms. Wollen Sie das Programm außerhalb der IDE ausführen, kann dies über eine *.bat*-Datei geschehen. Die *Main.class*-Datei befindet sich im Ordner *c:\JavaProjekte\JavaTutorial\build\classes\javatutorial*. Legen Sie beispielsweise im Ordner *c:\JavaProjekte\JavaTutorial\build* eine Datei *Start.bat* mit folgendem Inhalt an:

```
@echo off
JAVA –classpath ..\build\classes javatutorial.Main
pause
```

und führen diese *.bat*-Datei aus, so erscheint die Programmausgabe im Kommando Fenster von *Windows*.

Um aus der IDE direkten Zugriff auf die **Java-Dokumentation** zu haben, ist es notwendig, diese einzubinden. Hierzu ist von der Sun-Website http://java.sun.com/javase/downloads/index.jsp) die Datei *jdk-6-doc.zip* herunterzuladen (unter Punkt: Java SE 6 Documentation). Diese sollte in den Ordner *C:\Programme\Java\jdk1.6.0* kopiert werden, in dem sich auch die J2SE befindet. Im *NetBeans* ist die Option *Java Platform Manager* im Menü *Tools* auszuwählen. Im Register *Javadoc* ist die Schaltfläche *Add ZIP/Folder...* auszuwählen und über den nächsten Dialog ist die Datei *jdk-6-doc.zip* hinzuzufügen. Wenn Sie nun in der *main*-Methode der Klasse *Main* mit dem Cursor auf die Zeichenfolge *System* gehen und aus dem Umgebungsmenü *Show Javadoc* auswählen, wird die Javadoc der Klasse *System* im Browser angezeigt.

In späteren Übungen werden wir auch mit einer **relationalen Datenbank** arbeiten. Hierfür ist von der MySQL-Website (http://dev.mysql.com/downloads/mysql/5.0.-html#downloads) die Datei *mysql-5.0.27-win32.zip* herunterzuladen. Unter Windows XP muss diese Datei einfach im Windows Explorer angeklickt werden, mit einem Doppelklick auf *Setup* wird die Installation der MySQL-Datenbank gestartet. Im Dialog *Setup Type* sollten Sie *Complete* auswählen, ansonsten bestätigen Sie die vorgeschlagenen Einstellungen mit einer Ausnahme. Es wird der Port 3306 vorgeschlagen, hier ändern Sie auf 3307. Beim *Sign-Up*-Dialog können Sie *Skip Sign-Up* auswählen. Führen Sie auch die Server Konfiguration aus, wählen Sie dabei *Detailed Configuration*, *Developer Machine*, *Multifunctional Database* und bei den *InnoDB Tablespace Settings* statt dem *Installation Path* die Option *\MySQL Datafiles*. Ansonsten übernehmen Sie die Vorschlagswerte. Vergeben Sie kein *root password*, danach übernehmen Sie wieder die vorgeschlagenen Einstellungen. Um auf den MySQL-Server zugreifen zu können, sollten Sie noch das *HeidiSQL* Opensource Tool zur Verwaltung von MySQL Datenbanken installieren. Hierzu befindet sich

auf der Webseite die Datei *HeidiSQL_3.0RC4_Setup.exe*. Mit einem Doppelklick starten Sie die Installation. Sie sollten die Lizenzbedingungen akzeptieren, ansonsten übernehmen Sie die vorgeschlagenen Einstellungen. Auf dem Windows-Arbeitsplatz müsste ein Link *HeidiSQL* eingefügt sein, ein Doppelklick öffnet einen Dialog, in dem die Verbindungsparameter zum MySQL-Server zu spezifizieren sind (Voraussetzung ist, dass der MySQL-Server gestartet ist, siehe Dienste unter *Systemsteuerung*, *Verwaltung*, *Dienste*). Bei *Server Host* ist *localhost*, bei Port *3307*, bei *Username root* und bei *Password* nichts einzutragen. Mit dieser Anwendung sind Sie in der Lage neue Datenbanken (Schemata), neue Tabellen anzulegen und SQL-Befehle auszuführen. Als kleine Übung wollen wir eine neue Datenbank *javatutorial* mit einer Tabelle *Kunde* anlegen. Öffnen Sie das Menü *Tools* und wählen Sie *Create Database*. Nachdem die Datenbank *javatutorial* angelegt ist, wählen Sie aus dem Menü *Tools* die Funktion *Create Table*. Der Dialog ist weitgehend selbsterklärend und erlaubt auch das Anlegen der Spalten *Kundennummer* und *Name*. Um Daten einzugeben, muss die Tabelle markiert sein und über das Register *Data* erscheint die Tabelle im Editiermodus, so dass unmittelbar Werte eingegeben werden können.

Damit von einem Java-Programm auch auf die Datenbank zugegriffen werden kann, ist es notwendig, dass der passende **JDBC** (Java Database Connectivity)-Treiber zur Verfügung steht. Hierzu ist auf der Webseite eine jar-Datei mit dem Namen *mysql-connector-java-5.0.4-bin.jar* enthalten. Kopieren Sie diese am einfachsten in den Ordner *C:\Programme\MySQL*. In *NetBeans* sollten Sie das Projekt *JavaTutorial* selektieren und aus dem Umgebungsmenü die Option *Properties* auswählen. Wählen Sie dort *Libraries* → *Run* aus und fügen Sie die Treiber-Datei mittels der Schaltfläche *Add JAR/Folder ...* hinzu. Damit können wir später eine Verbindung zur Datenbank aufbauen. Falls Sie im Detail Hilfe zu SQL bzw. zu *MySQL* brauchen, empfehle ich folgende Originalquelle: http://dev.mysql.com/doc/refman/5.0/en/index.html.

3. Grundlagen von Java am Beispiel

Ein **Java-Programm** besteht aus einer oder mehreren Klassen. Diese Klassen können mehrere Methoden haben, über welche die geforderte Funktionalität des Pro-

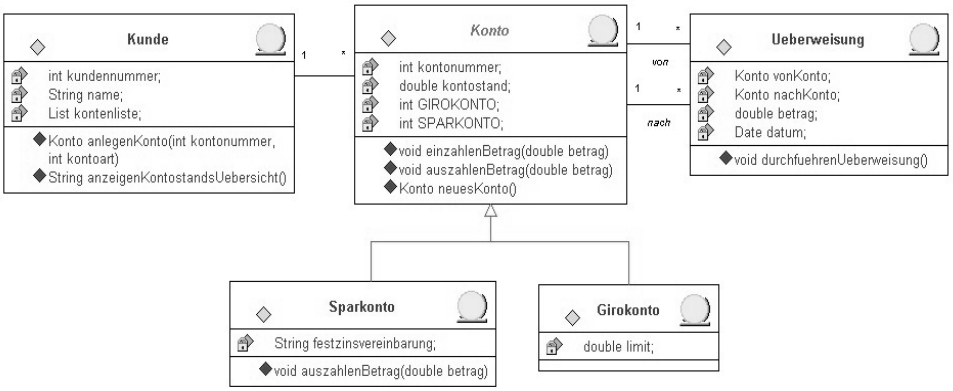

Abb. 2: Fachkonzeptmodell der Beispielbank für Programmierbeispiel

gramms erfüllt wird. Wir wollen ein einfaches und gemessen an der Realität sehr vereinfachtes Problem mit unserem Programm umsetzen. In unserer Beispielbank sollen auf Konten Ein- und Auszahlungen sowie Überweisungen durchgeführt werden können. Im ersten Schritt wollen wir an die unmittelbare Umsetzung der Fachkonzeptschicht gehen und folgen dabei dem **Domain Model** (vgl. Abb. 7.3 im Buch).

Wir starten *NetBeans* mit dem Projekt *JavaTutorial*. Im Ordner *Source Packages* legen wir ein neues **Paket** an (selektieren von *Source Packages* →*Umgebungsmenü* → *New* → *Java Package*). Im Eingabefeld *Package Name* tragen Sie bitte *bank* ein und lösen die Schaltfläche *finish* aus. Nun legen wir die Java-**Klasse** *Kunde* an. Selektieren Sie das Paket *bank* → *Umgebungsmenü* → *New* → *File/Folder...*, im rechten Teil den *File Type Java Class* auswählen, über den *Next Button* gelangen Sie zum Dialog, in dem Sie *Kunde* als Klassennamen eingeben und mit dem *Finish Button* gelangen Sie wieder in die IDE zurück und sehen im Editor bereits den Quellcode der Klasse *Kunde* mit Klassendeklaration und **Standardkonstruktor**. Nun definieren Sie die Attribute des Klassenmodells als **Felder** der Klasse. Im Baum des Explorers erweitern Sie den Knoten *Kunde* und selektieren *Fields*, im Umgebungsmenü wählen Sie *Add Field...* Im folgenden Dialog tragen Sie den Feldnamen, z.B. *kundennumer* ein und

wählen den Typ aus bzw. geben ihn ein. Statt mittels Wizard können Sie die Eingabe natürlich auch direkt im Editor vornehmen. Das Attribut *kontenliste* implementiert die **Assoziation** zur Klasse *Konto* (vgl. Abb. 2) und ist vom Typ *java.util.List*. Die Deklaration der Methode *anzeigenKontostandsUebersicht()* funktioniert analog. Beim Übersetzen werden Sie merken, dass Sie in der Methode *anzeigenKontostandsUebersicht()* noch eine *return*-Anweisung einfügen müssen. Auf den Inhalt gehe ich ein, sobald wir die Klasse *Konto* implementiert haben (vgl. Zeilen 8-13 in Listing 1). Sie können einstweilen einfach *return null;* einfügen. Die Methode *anlegenKonto()* schickt die Nachricht *neuesKonto(kontonummer, kontoart)* an die Klasse *Konto* und bekommt entweder eine Referenz auf ein *Sparkonto*- oder ein *Girokonto*-Objekt zurück. In Zeile 23 wird die Kontonummer zusammengesetzt aus der Kundennummer, zwei Stellen mit der Kontoart und eine fortlaufende Nummer bezogen auf den jeweiligen Kunden. Objekte beider Klassen sind aufgrund der **Vererbungsbeziehung** ja auch vom Typ *Konto*. Aus Zeile 24 (vgl. Listing 1) wird ersichtlich, dass es sich bei der Methode *neuesKonto()* um eine **statische Methode** handelt, d.h. die Nachricht wird an die Klasse *Konto* geschickt. Diese Referenz wird in das Listen-Objekt *kontenliste* mit der *add()*-Methode aufgenommen (Zeile 25). Zur Erzeugung von Objekten der Klasse *Kunde* steht bisher der Standardkonstruktor zur Verfügung. Wir wollen einen **überladenden** Konstruktor hinzufügen. Mit *NetBeans* kann dies auf zwei Arten erfolgen. Entweder wir schreiben den Code einfach im Editor oder wir lassen uns durch die IDE unterstützen. Im zweiten Fall selektieren wir im Explorer in der Klasse *Kunde* den Knoten *Constructors* im Umgebungsmenü wählen wir *add* → *Contructor*. Da der Name des Konstruktors immer gleich dem Klassennamen ist und der Rückgabetyp immer void ist, kann dies auch gar nicht geändert werden. Im unteren Fenster Parameters fügen wir über *Add* die Parameter *kundennummer* und *name* mit den Typen *int* und *String* hinzu. Nach der *OK*-Bestätigung können wir den generierten Code im Editor ergänzen. In den Zeilen 28 und 29 (vgl. Listing 1) werden die Objektattribute mit den Parameterwerten initialisiert. In der Zeile 30 wird dem Feld *kontenliste* eine Referenz auf ein neu erzeugtes *ArrayList*-Objekt zugewiesen. In dieser Anweisung wird eine Neuerung in Java verwendet, die erst seit der Version 5.0 (manchmal auch mit 1.5 bezeichnet) verfügbar ist (vgl. zu einem kompakten Überblick zu Spracherweiterungen in Java 5.0 Trapp, 2004, S. 10 ff.). Ursprünglich waren **Collection**-Objekte generisch, d.h. ein *ArrayList* konnte Objekte jeder beliebigen Klasse enthalten. Aus der Sicht der *Collection* handelte es sich eben um *Object*-Objekte. Dies hatte zur Konsequenz, dass bei einem Zugriff auf Objekte in einer Kollektion eine explizite **Typumwandlung** stattfinden musste. Dies kann zu Laufzeitfehlern führen, da die *Collection*-Objekte und andere Container-Klassen nicht typsicher waren. Ab der Version 5.0 stehen die so genannten **Generics** zur Verfügung. In spitzen Klammern wird der Typ der Objekte im Container spezifiziert (vgl. Zeile 30 in Listing .1). Damit kann die *ArrayList kontenliste* nur für *Konto*-Objekte verwendet werden. Beim Auslesen wird dadurch auch die Typumwandlung überflüssig. Natürlich können die Container-Klassen auch nach wie vor ohne die Typspezifikation verwendet werden.

Bevor wir die nächste Klasse anlegen, können wir für die beiden Felder noch *getr-* und *set*-Methoden generieren lassen: Selektieren Sie die Klasse *Kunde* → *Umgebungsmenü* → *Refactor* → *Encapsulate Fields...* und bestätigen Sie mit dem *next* Button, im Output Fenster bestätigen Sie den Button *Do Refactoring*.

Listing 1: Klasse *Kunde*

```
 1 package bank;
 2 public class Kunde {
 3     public Kunde() {}
 4     private int kundennummer;
 5     private String name;
 6     private java.util.List<Konto> kontenliste;
 7     public String anzeigenKontostandsUebersicht() {
 8         String out="Kontostandsübersicht von Kontoinhaber "+name+"\n";
 9         out+="Kontonummer   Kontostand \n";
10         for(Konto konto : kontenliste){
11           out+=konto.getKontonummer()+" "+konto.getKontostand()+"\n"; }
12         return out;
13     }
14     public int getKundennummer() {
15         return kundennummer; }
16     public String getName() {
17         return name; }
18     public void setKundennummer(int kundennummer) {
19         this.kundennummer = kundennummer; }
20     public void setName(String name) {
21         this.name = name; }
22     public Konto anlegenKonto(int kontoart) throws Exception{
23          int kontonummer=(((kundennummer*100)+kontoart)
              *1000)+kontenliste.size()+1;
24         Konto konto=Konto.neuesKonto(kontonummer, kontoart);
25         kontenliste.add(konto);
26         return konto;}
27     public Kunde(int kundennummer, String name) {
28         this.kundennummer=kundennummer;
29         this.name=name;
30         kontenliste=new java.util.ArrayList<Konto>(); }}
```

Bei der Anlage der Klasse *Konto* sollten Sie nur darauf achten, dass Sie die Klasse als abstrakt spezifizieren. Ansonsten ist das Vorgehen analog zur Klasse *Kunde*. Die beiden Attribute *GIROKONTO* und *SPARKONTO* sind als Konstanten mit den

Eigenschaften *static* und *final* zu versehen, sie bekommen die Werte 0 bzw. 1 (vgl. Zeilen 5 und 6 in Listing 2a). Wenn Sie die Methoden über den Wizard anlegen, so können Sie auch dort gleich die Parameter mit ihren Typen spezifizieren. Im Dialog *Add New Method* wählen Sie im Register *Parameter Add*, der Rest ist selbsterklärend. Damit auf den Kontostand auch durch die Unterklassen zugegriffen werden kann, wurde dieser mit dem Zugriffsmodifikator *protected* definiert (vgl. Zeile 4). Wir können auch gleich den Inhalt der beiden Methoden *einzahlenBetrag()* und *auszahlenBetrag()* programmieren, in dem der Kontostand erhöht bzw. reduziert wird (vgl. Zeilen 12-17). Die Methode *neuesKonto()* liefert entweder ein *Sparkonto*- oder *Girokonto*-Objekt zurück und fungiert als eine Art **Fabrikmethode** (vgl. 8.3.1). In dieser Methode wird das *switch-case-default*-Konstrukt verwendet, mit dessen Hilfe man in verschiedene Alternativen verzweigen kann. In der *default*-Marke wird eine Ausnahmesituation durch eine *Exception* repräsentiert. Diese *Exception* wird geworfen, wenn eben die Kontoart nicht 0 oder 1 ist, was theoretisch vorkommen könnte. Da die *Exception* in der Methode nicht abgefangen wird, steht im Methodenkopf die *throws*-Klausel, d.h. die *Exception* wird weitergeworfen. Ansonsten wurden noch die *get*- und *set*-Methoden generiert.

Listing 2a: Abstrakte Klasse *Konto*

```
 1 package bank;
 2 public abstract class Konto {
 3     private int kontonummer;
 4     protected double kontostand;
 5     public static final int GIROKONTO = 0;
 6     public static final int SPARKONTO = 1;
 7     public Konto() {}
 8     public Konto(int kontonummer) {
 9         this.kontonummer=kontonummer;
10         this.kontostand=0;
11     }
12     public void einzahlenBetrag(double betrag) {
13         kontostand+=betrag;
14     }
15     public void auszahlenBetrag(double betrag) throws Exception {
16         kontostand-=betrag;
17     }
18     public int getKontonummer() {
19         return kontonummer;
20     }
```

Listing 2b: Abstrakte Klasse *Konto*

```
21      public void setKontonummer(int kontonummer) {
22          this.kontonummer = kontonummer;
23      }
24      public double getKontostand() {
25          return kontostand;
26      }
27      public void setKontostand(double kontostand) {
28          this.kontostand = kontostand;
29      }
30      public static Konto neuesKonto(int
          kontonummer, int kontoart) throws Exception{
31          Konto konto=null;
32          switch(kontoart){
33              case GIROKONTO:
34                  konto = new Girokonto(kontonummer);
35                  break;
36              case SPARKONTO:
37                  konto = new Sparkonto(kontonummer);
38                  break;
39              default:
40                  throw new Exception("Falsche Kontoart "+kontoart);
41          }
42          return konto;
43      }
44  }
```

Die Klasse *Girokonto* ist eine Unterklasse der Klasse *Konto* und unterscheidet sich nur durch ein zusätzliches Attribut *limit*. Zu erwähnen ist noch die Besonderheit im Konstruktor, der mit *super(kontonummer)* den Konstruktor der Oberklasse aufruft (vgl. Zeile 7 in Listing 3).

Listing 3: ***Girokonto***

```
 1 package bank;
 2 public class Girokonto extends Konto {
 3     private double limit;
 4     public Girokonto() {
 5     }
 6     public Girokonto(int kontonummer) {
 7         super(kontonummer);
 8     }
 9     public double getLimit() {
10         return limit;
11     }
12     public void setLimit(double limit) {
13         this.limit = limit;
14     }
15 }
```

Die Klasse *Sparkonto*, die ebenfalls von der Klasse *Konto* erbt, weist die Besonderheit auf, dass die Methode *auszahlenBetrag()* der Oberklasse in der Unterklasse **überschrieben** wird. Damit wird sichergestellt, dass durch eine Auszahlung der Kontostand eines Sparkonto-Objektes nicht negativ wird. Aus den Zeilen 9-12 im Listing 4 wird ersichtlich, dass im Falle, dass der Auszahlungsbetrag größer als der aktuelle Kontostand ist, eine *Exception* geworfen wird. Dies wird auch wieder in der Methodendeklaration deutlich. Zu beachten ist, dass damit auch die Methodendeklaration der Oberklasse *Konto* die Klausel *throws Exception* beinhaltet (vgl. Zeile 15 in Listing 2a). Die *else*-Klausel beinhaltet noch eine kleine Besonderheit. Reicht der Kontostand zur Auszahlung aus, dann wird einfach die *auszahlen-Betrag()*-Methode der Oberklasse ausgeführt. In unserem Beispiel, bei der die Methode der Oberklasse ja nur eine Zeile aufweist, scheint dies keinen besonderen Vorteil zu bringen. Denken Sie jedoch daran, dass wenn sich diese Methode mal ändert, diese Änderung eben nur einmal durchgeführt werden muss, was ein grundsätzlicher Vorteil der Vererbung darstellt.

Listing 4: Klasse *Sparkonto*

```
 1 package bank;
 2 public class Sparkonto extends Konto {
 3     public Sparkonto() {
 4     }
 5     private String festzinsvereinbarung;
 6     public Sparkonto(int kontonummer) {
 7         super(kontonummer);
 8     }
 9     public void auszahlenBetrag(double betrag) throws Exception{
10         if(betrag > kontostand){
11             throw new Exception("Kontostand zu
                 gering, keine Auszahlung möglich!");
12         }
13         else{
14             super.auszahlenBetrag(betrag);
15         }
16     }
17     public String getFestzinsvereinbarung() {
18         return festzinsvereinbarung;
19     }
20     public void setFestzinsvereinbarung(String festzinsvereinbarung) {
21         this.festzinsvereinbarung = festzinsvereinbarung;
22     }
23 }
```

Nun können wir auch noch einmal auf die Methode *anzeigenKontostandsUebersicht()* der Klasse *Kunde* zurückkommen (vgl. Zeilen 7-13 in Listing 1). Grundsätzlich wird in einfachster Weise eine Zeichenkette aufgebaut. Hierzu wird eine **Schleifenkonstruktion** verwendet, die erst seit Java 5 zur Verfügung steht. Die so genannte **erweiterte foreach-Schleife** hat die allgemeine Syntax:

```
for (FormalerParameter : Ausdruck)
```

Dabei besteht *FormalerParameter* aus dem Datentyp und Variablennamen, in unserem Fall *Konto konto*. *Ausdruck* ist ein Array oder ein Objekt des Typs *java.lang.Iterable*, z.B. ein *Collection*-Objekt, in unserem Fall das *ArrayList*-Objekt *kontenliste*. Das bedeutet die Schleife durchläuft nacheinander alle Elemente der *kontenlis-*

te und stellt in der Variablen *konto* die Referenz auf das aktuelle *Konto*-Objekt bereit, so dass *konto* kann im Schleifenrumpf angesprochen werden. Das Listing 5 zeigt die gleichwertige Lösung mit der klassischen **for-Schleife** ohne Verwendung von Generics.

Listing 5: Klassische for-Schleife

```
1 for(int i=0; i<kontenliste.size(); i++){
2     Konto konto=(Konto)kontenliste.get(i);
3     out+=konto.getKontonummer()+"   "+konto.getKontostand()+"\n";}
```

Die Klasse *Ueberweisung* implementiert mit den beiden Referenzattributen *vonKonto* und *nachKonto* die beiden Assoziationen zwischen *Konto* und *Ueberweisung* (vgl. Zeilen 3 und 4 in Listing 6). Die Methode *durchfuehrenUeberweisung()* verwendet die beiden Methoden *auszahlenBetrag()* und *einzahlenBetrag()* der Klasse *Konto* bzw. der konkreten Unterklassen. Folgerichtig wird auch wieder die *Exception* der Methode *auszahlenBetrag()* weitergeworfen.

Listing 6: Klasse Ueberweisung

```
 1 package bank;
 2 public class Ueberweisung {
 3     private Konto vonKonto;
 4     private Konto nachKonto;
 5     private double betrag;
 6     private java.util.Date datum;
 7     public Ueberweisung() {}
 8     public Ueberweisung(
         Konto vonKonto,Konto nachKonto,
         double betrag, java.util.Date datum) {
 9         this.vonKonto=vonKonto;
10         this.nachKonto=nachKonto;
11         this.betrag=betrag;
12         this.datum=datum;
13     }
14     public void durchfuehrenUeberweisung() throws Exception {
15         vonKonto.auszahlenBetrag(betrag);
16         nachKonto.einzahlenBetrag(betrag);
17 }}
```

Die Klasse *BankTest* beinhaltet eine *main()*-Methode und ist damit ausführbar. Diese verwenden wir, um die Funktionalität unserer Klassen zu testen. Die einzelnen Anweisungen sind weitgehend selbsterklärend (vgl. Listing 7). Die Eingaben von Kundennummer und Kundennamen erfolgt über kleine Nachrichtenfenster. Die Methoden hierzu sind in dem Paket *Tools* in der Klasse *IOTool* als statische Methoden implementiert. Eine nähere Erläuterung erfolgt an dieser Stelle nicht. Der Quellcode ist auf der Webseite zu finden. Zum näheren Verständnis sind in Abbildung 3 die Ausgaben der Testklasse wiedergegeben.

Listing 7: Testklasse *Banktest*

```
 1 package bank;
 2 import bank.tools.IOTool;
 3 public class BankTest {
 4     public BankTest() {}
 5     public static void main(String[] args) {
 6     try{
 7         int kundennummer=IOTool.readInteger("Bitte KundenNr eingeben ")
 8         String kundenname=IOTool.readLine(
             "Bitte Kundenname eingeben ")
 9         Kunde arm=new Kunde(kundennummer, kundenname)
10         Konto konto1=arm.anlegenKonto( Konto.GIROKONTO)
11         Konto konto2=arm.anlegenKonto( Konto.SPARKONTO)
12         System.out.println("(1) "+arm.anzeigenKontostandsUebersicht())
13         konto1.einzahlenBetrag(1000)
14         System.out.println("(2) "+arm.anzeigenKontostandsUebersicht())
15         konto2.einzahlenBetrag(100)
16         System.out.println("(3) "+arm.anzeigenKontostandsUebersicht())
17         konto1.auszahlenBetrag(400)
18         System.out.println("(4) "+arm.anzeigenKontostandsUebersicht())
19         konto2.auszahlenBetrag(10)
20         System.out.println("(5) "+arm.anzeigenKontostandsUebersicht())
21         Ueberweisung ueb=new Ueberweisung(
             konto1,konto2, 25, new java.util.Date())
22         ueb.durchfuehrenUeberweisung();
23         System.out.println("(6)"+arm.anzeigenKontostandsUebersicht());}
24         catch(Exception e){
25         System.out.println("Fehler: "+e.getMessage());
26 }}}
```

(1) Kontostandsübersicht von Kontoinhaber Arm Kontonummer Kontostand 100001 0.0 101002 0.0 (2) Kontostandsübersicht von Kontoinhaber Arm Kontonummer Kontostand 100001 1000.0 101002 0.0 • (3) Kontostandsübersicht von Kontoinhaber Arm Kontonummer Kontostand 100001 1000.0 101002 100.0	(4) Kontostandsübersicht von Kontoinhaber Arm Kontonummer Kontostand 100001 600.0 101002 100.0 (5) Kontostandsübersicht von Kontoinhaber Arm Kontonummer Kontostand 100001 600.0 101002 90.0 (6) Kontostandsübersicht von Kontoinhaber Arm Kontonummer Kontostand 100001 575.0 101002 115.0

Abb. 3: Ausgaben der Testklasse *BankTest*

4 Beispiel für grafische Oberflächen mit dem Swing-Framework

Im bisherigen Bank-Beispiel erfolgte die Interaktion mit dem Benutzer über das Konsolfenster bzw. über einfache Eingabe-Fenster. Nun wollen wir eine einfache grafische Dialogschnittstelle bzw. Benutzungsschnittstelle (Graphical User Interface – GUI) entwickeln, mit welcher der Anwender interagieren kann. Java bietet hierzu Klassenbibliotheken in Form eines Frameworks an. Diese **Java Foundation Classes** (JVC) lassen sich in folgende vier Gruppen von Komponenten einteilen (vgl. Ratz u.a., 2006, S. 132):

- **Grundkomponenten**, z.B. Beschriftungen (labels), Knöpfe (buttons), Auswahlfelder usw.
- **Container**, sind Komponenten, die selbst wieder Komponenten enthalten können, z.B. *JOptionPane, JFrame* usw.
- **Layout-Manager, Farben und Fonts**, die für die Anordnung und Gestaltung einzelner Komponenten zuständig sind.

– **Ereignisse und Listener**, die für die Interaktion der Komponenten mit den Anwendern benötigt werden.

Die einzelnen Klassen befinden sich in den Paketen *java.awt* (**AWT**-Bibliothek - Abstract Window Kit) und *javax.swing* (**Swing**-Bibliothek). Die neueren Swing-Klassen ersetzen die alten AWT-Grundkomponenten und –Container, während die beiden letzten Gruppen der obigen Aufzählung weiterhin aus dem AWT-Paket verwendet werden. Abbildung 4 gibt einen Überblick in Ausschnitten.

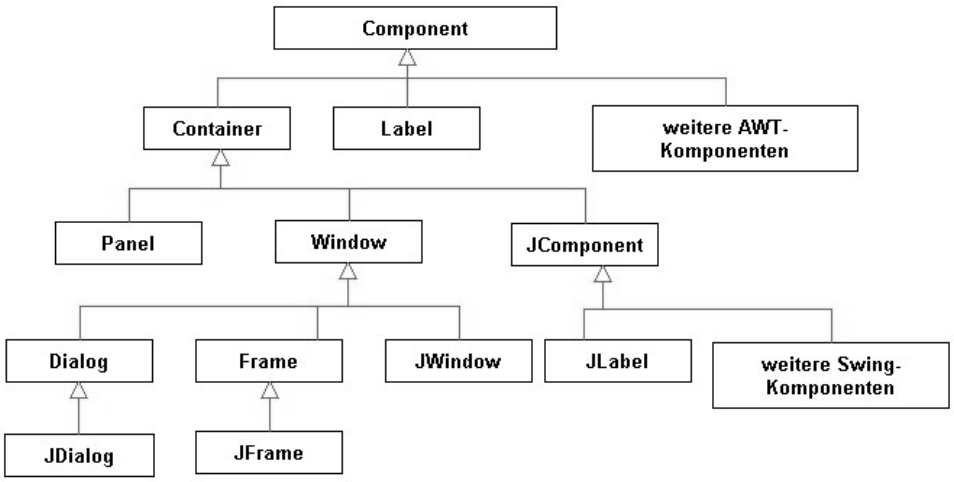

Abb. 4: AWT- und Swing-Klassen-Hierarchie (vgl. Ratz u.a., 2006, S. 140)

Unsere kleine Anwendung erlaubt einen Kunden anzulegen, ein Konto anzulegen, Ein- bzw. Auszahlungen zu verbuchen, Überweisungen zu tätigen und eine Kontostandsübersicht anzuzeigen. Hierzu entwickeln wir mit unserer IDE als erstes einen Anwendungsframe mit einem Anwendungsmenü (vgl. Abb. 5).

Hierzu legen Sie bitte ein neues Java Package mit dem Namen *bankMitGUI* an. In dieses kopieren Sie die Klassen aus dem Paket *bank*. Als erstes entwickeln wir den Anwendungsdialog als Unterklasse von *JFrame* mit einem Menübalken, einem Menü *Anwendungen* mit den Menüpunkten *Kunde anlegen*, *Konto anlegen*, *Ein-/Auszahlungen durchführen* und *Überweisung durchführen* (vgl. Abb. 5).

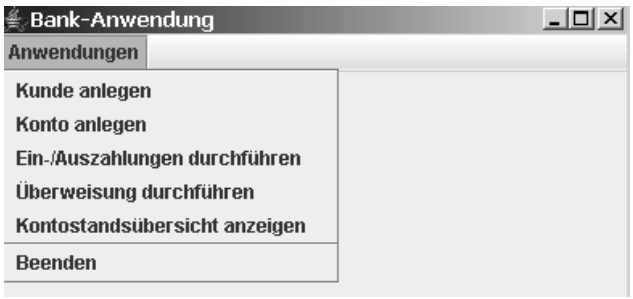

Abb. 5: Anwendungs-Dialog: Bank Anwendung

Im *NetBeans* Explorer selektieren Sie das Paket *bankMitGUI* → *Umgebungsmenü* → *New* → *File/Folder...* → im Dialog *New File* → Selektion von *Java GUI Forms* → im rechten Dialogteil: Selektieren von *JFrame Form* → *Next* → Class Name: *DlgBank* → *Finish*. Nun öffnet sich der GUI-Editor von *NetBeans*. Im rechten Fenster *Palette* → *Swing* selektieren Sie *JMenuBar* und klicken auf das *JFrame Objekt* im linken Editor-Fenster. Dort werden nun ein Menübalken und ein Menü eingefügt. Im linken Editor-Fenster selektieren Sie das gerade angelegte Objekt *jMenuBar1* und über das Umgebungsmenu gelangen Sie zur Option *Change Variable Name*. Im *Rename*-Fenster nennen Sie das Objekt *jMenuBarBank*. In gleicher Weise verfahren Sie mit *jMenu1* und nennen es in *jMenuAnwendungen* um. Im Umgebungsmenü von *jMenuAnwendungen* führen Sie die Option *Edit Text* aus. Geben Sie statt *Menu* den Text *Anwendungen* ein. Nun müssen noch Menüoptionen zum Menü hinzugefügt werden. Im Umgebungsmenü von *jMenuAnwendungen* → *Add* → *JMenuItem*. Damit wird eine Menüoption eingefügt. Nennen Sie diese um in *jMenuItemKundeAnlegen*. Selektieren Sie diesen Knoten im Inspector und ändern Sie im darunter liegenden Eigenschafts-(Properties-)Fenster den Wert des Attributs *text* in *Kunde anlegen*. Verfahren Sie in analoger Weise mit den anderen Menü-Optionen (vgl. Abb. 5). Selektieren Sie zum Schluss nochmals *JFrame* im Inspector und ändern Sie den Wert des Attributs *title* in *Bank Anwendung* und selektieren Sie im Register *Code* für die Eigenschaft *Form Size Policy* die Option *Generate Resize Code*. Damit wird das Fenster in der Größe 400x300 in der Mitte der Monitoranzeige platziert. Sie können nun die Klasse *DlgBank* ausführen und bekommen den Dialog angezeigt. Lassen Sie noch für die *jMenuItem*-Felder Standard-*get*-Methoden generieren. Damit haben wir schon unsere erste grafische Oberfläche entwickelt. Wenn Sie im Quellcode-Editor den Code anschauen, sehen Sie, dass wir einiges geschafft haben. Im Quellcode ist der größte Teil mit einem Blauton unterlegt, was bedeutet, dass dieser Teil nicht im Quellcode-Editor verändert werden kann.

Als nächstes entwickeln wir den Dialog zum Anlegen eines Kunden. Die Vorgehensweise mit *NetBeans* ist ähnlich. Statt *JFrame Form* wählen Sie *JDialog Form* und vergeben den Klassennamen *DlgKundeAnlegen*. Die Festlegung von *title* und die Positionierung des Objektes auf dem Monitor ist wieder gleich wie bei der *DlgBank*-Klasse. Ab der Version 5.0 von *NetBeans* ist es nicht mehr notwendig einen Layout-Manager zu spezifizieren. Standardmäßig ist *Free Design* mit dem *GroupLayout Manager* unterstellt. Damit können Sie die einzelnen Komponenten durch drag-and-drop auf der Oberfläche beliebig platzieren. Im oberen Fenster *Palette* im Register *Swing* sind die Swing-Komponenten bereitgestellt. Durch Links-Klick kann eine Komponente ausgewählt werden und durch einen weiteren Klick im *JDialog*-Container wird diese Komponente zu einem Bestandteil der Benutzungsoberfläche. Durch übliche Maus-Manipulationen lassen sich die Komponenten positionieren und in ihrer Größe verändern. In der Abbildung 6 sehen, Sie dass zwei *JLabel*-, zwei *JTextField*- und zwei *JButton*-Komponenten eingefügt wurden. Dies können Sie selbständig durchführen. Vergessen Sie nicht die Objekte umzubenennen, z.B. statt *jLabel1*, *jLabelKundennummer* und statt *jTextField1 jTextFieldKundennummer*. Über die Option *Edit Text* können Sie die angezeigten Inhalte festlegen bzw. bei den Textfeldern den Standardtext löschen.

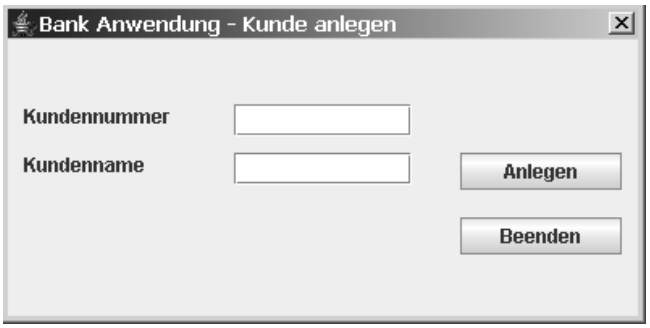

Abb. 6: Anwendungs-Dialog: Bank Anwendung – Kunde anlegen

Der nächste Schritt besteht nun darin, dass einerseits unsere Anwendung gestartet wird und die Verbindung zwischen Anwendungsdialog und dem Dialog zum Kunden erfassen hergestellt wird. Hierzu entwickeln wir eine einfache Start-Klasse *Bank* mit einer *main()*-Methode, eine **Input-Controller**-Klasse (vgl. 7.4.1), die wir *CtrlDlgBank*, eine **Fassaden**-Klasse (vgl. 8.4.1) *BankHandler* als Schnittstelle zu den Fachklassen und eine Klasse *Kunden* zur Verwaltung von Kundenobjekten. Einen Überblick liefert das Klassendiagramm in Abb. 7.

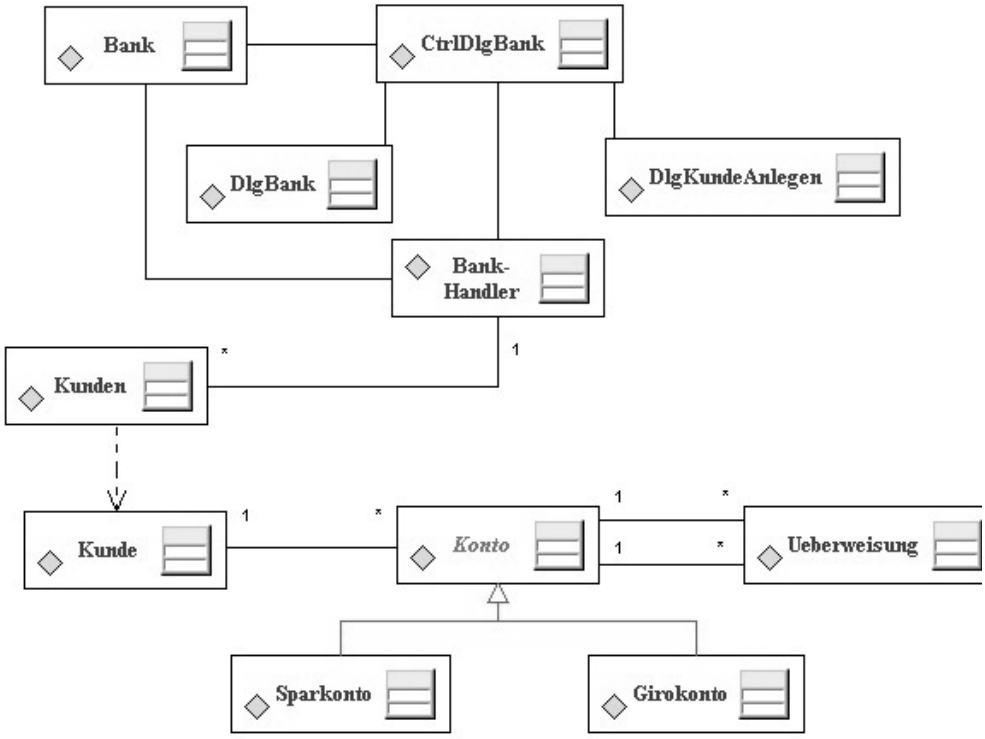

Abb. 7: Klassendiagramm der Swing-Implementierung des Bank-Beispiels

Die Klasse *Bank* (vgl. Listing 8) beinhaltet die *main()*-Methode und instanziiert ein Objekt *ctrlDlgBank* der Input-Controller-Klasse sowie ein Objekt *bankHandler* der Fassaden-Klasse und sendet die Nachricht *startDlgBank()* an das Input-Controller-Objekt. Aufgrund der Einfachheit des Beispiels wurden keine unterschiedlichen **Anwendungsfall-Controller** differenziert.

Listing 8: Klasse *Bank*

```
 1 package bankMitGUI;
 2 public class Bank {
 3     private CtrlDlgBank ctrlDlgBank;
 4     private BankHandler bankHandler;
 5     public Bank() {}
 6     public static void main(String[] args) {
 7         new Bank().start();
 8     }
 9     private void start(){
10         ctrlDlgBank=new CtrlDlgBank();
11         bankHandler=new BankHandler();
12         ctrlDlgBank.startDlgBank(bankHandler);
13     }
14 }
```

Die Klasse *CtrlDlgBank* ist schon ein wenig umfangreicher. Die Methode *startDlgBank()* erzeugt ein *DlgBank*-Objekt (vgl. Abb. 5) und registriert bei dem *JMenuItemKundenAnlegen*-Objekt des *DlgBank*-Objektes einen Listener. Ein **Listener** ist der Empfänger eines Ereignisses (event). Beide Klassenkategorien werden im Swing-Framework zum ereignisgesteuerten Ablauf von Programmen mit grafischen Oberflächen eingesetzt. Das **Ereignis** wird durch den Benutzer einer grafischen Oberfläche ausgelöst, z.B. durch berühren mit der Maus, durch Auswahl eines Menü-Eintrages oder das Betätigen der Maustaste. Eine Ereignisquelle können alle Komponenten einer grafischen Oberfläche sein, z.B. ein Knopf (Button oder Schaltfläche). Bei solch einer Komponente können Ereignisempfänger bekannt gemacht werden. Die konkrete programmtechnische Umsetzung kann grundsätzlich unterschiedlich erfolgen (vgl. Ratz u.a., 2006, S. 205 ff.):

- Die Listener Klasse wird als **innere Klasse** realisiert.
- Die Listener Klasse wird als **anonyme Klasse** realisiert.
- Die **Container Klasse** (z.B. JFrame) wird selbst zur Listener Klasse.
- Die Listener Klasse wird als **separate Klasse** realisiert.

In unserem Fall haben wir eine Kombination gewählt. Unsere *CtrlDlgBank*-Klasse fungiert als separate Steuerungsklasse und in ihr haben wir den Listener als eine **anonyme Klasse** realisiert und beim Menü-Eintrag registriert (vgl. Listing 9a, Zeilen 10-14). Diese Vorgehensweise ermöglicht eine strikte Trennung von Oberfläche und Ereignisverarbeitung. Im Prinzip könnte die Steuerungsklasse auch selbst als

Listener verwendet werden, dann müsste in unserem Fall diese Klasse eine *actionPerformed()*-Methode implementieren, die auf die Ereignisse entsprechend reagieren würde. Durch die Verwendung der inneren Klasse wird direkt die in der nicht weiter bezeichneten anonymen Klasse definierten *actionPerformed()*-Methode verwendet, die in unserem Beispiel die Methode *jMenuItemKundenAnlegenActionPerformed()* aufruft (vgl. Zeile 12). In dieser Methode wird der nächste Dialog-Objekt zum Anlegen eines Kunden erzeugt und die Schaltflächen *Anlegen* und *Beenden* (vgl. Abb. 6) werden in gleicher Weise mit Listener-Objekten versehen (vgl. Zeilen 17-26 in den Listings 9a und 9b). Eine interessante Methode ist noch *jButtonKundeAnlegenActionPerformed()*. Diese übernimmt die Rolle des **Event-Handlings** und wird manchmal auch als Aktions-Methode bezeichnet. Es werden die Werte aus den Eingabefeldern des *DlgKundeAnlegen*-Objektes eingelesen. Die Anweisung in Zeile 31 stellt sicher, dass bei der Kundennummer auch ein Zahlenwert eingegeben wird. Wenn dies nicht der Fall ist, wirft die Methode *parseInt()* der Basisklasse *Integer* eine *NumberFormatException*, die im *catch()*-Block (vgl. Zeilen 37-39) zu einer Mitteilung an den Benutzer führt. In der Zeile 33 wird über die Verwendung des *BankHandler*-Objektes ein neues *Kunde*-Objekt erzeugt. Das *BankHandler*-Objekt hat lediglich Delegationsfunktion (vgl. Listing 10).

Listing 9a: Klasse *CtrlDlgBank*

```
 1 package bankMitGUI;
 2 public class CtrlDlgBank {
 3     private DlgBank dlgBank;
 4     private DlgKundeAnlegen dlgKundeAnlegen;
 5     private BankHandler bankHandler;
 6     public CtrlDlgBank() { }
 7     public void startDlgBank(BankHandler bankHandler){
 8         this.bankHandler=bankHandler;
 9         dlgBank=new DlgBank();
10         dlgBank.getJMenuItemKundenAnlegen()
          .addActionListener(new java.awt.event.ActionListener() {
11           public void actionPerformed(java.awt.event.ActionEvent evt){
12                jMenuItemKundenAnlegenActionPerformed();}};
13         dlgBank.setVisible(true);
14     }
15     private void jMenuItemKundenAnlegenActionPerformed(){
16        dlgKundeAnlegen=new DlgKundeAnlegen(dlgBank, true);
17        dlgKundeAnlegen.getJButtonKundeAnlegen()
         .addActionListener(new java.awt.event.ActionListener() {
```

Listing 9b: Klasse *CtrlDlgBank*

```
18            public void actionPerformed(java.awt.event.ActionEvent evt){
19                jButtonKundeAnlegenActionPerformed();
20              }
21          };
22          dlgKundeAnlegen.getJButtonKundeAnlegenBeenden()
          .addActionListener(new java.awt.event.ActionListener() {
23            public void actionPerformed(java.awt.event.ActionEvent evt){
24                jButtonKundeAnlegenBeendenActionPerformed();
25              }
26          };
27          dlgKundeAnlegen.setVisible(true);
28      }
29      private void jButtonKundeAnlegenActionPerformed(){
30          try{
31              int kundennummer=Integer.parseInt(
              dlgKundeAnlegen.getJTextFieldKundennummer().getText());
32              String kundenname=dlgKundeAnlegen
                .getJTextFieldKundenname().getText();
33              Kunde neuerKunde=bankHandler
                  .anlegenKunde(kundennummer, kundenname);
34              new javax.swing.JOptionPane().showMessageDialog(
              dlgKundeAnlegen,"Kunde:"+neuerKunde.getName()+" angelegt.");
35              clearDlgKundeAnlegen();
36          }
37          catch(NumberFormatException e){
38              new javax.swing.JOptionPane().showMessageDialog(
              dlgKundeAnlegen,"Bitte Zahl als Kundennummer eingeben.");
39          }
40      }
41      private void clearDlgKundeAnlegen(){
42          dlgKundeAnlegen.getJTextFieldKundenname().setText("");
43          dlgKundeAnlegen.getJTextFieldKundennummer().setText("");
44      }
45      private void jButtonKundeAnlegenBeendenActionPerformed(){
46          dlgKundeAnlegen.dispose();
47  }}
```

Listing 10: Klasse *BankHandler*

```
 1 package bankMitGUI;
 2 public class BankHandler {
 3     private Kunden kunden;
 4     public BankHandler() {
 5         kunden=new Kunden();
 6     }
 7     public Kunde anlegenKunde(
     int kundennummer, String kundenname){
 8         return kunden.anlegenKunde(
         kundennummer, kundenname);
 9     }
10 }
```

Die konkrete Erzeugung des *Kunde*-Objektes erfolgt in der Klasse *Kunden* (vgl. Listing 11). Diese Klasse verfügt über ein *HashMap*-Objekt *kunden,* in dem die Referenzen der erzeugten *Kunde*-Objekte hinterlegt werden. Als Schlüssel-Objekt wird die Kundennummer verwendet (vgl. Zeilen 3, 5 und 9).

Listing 11: Klasse Kunden

```
 1 package bankMitGUI;
 2 public class Kunden {
 3     private java.util.HashMap<Integer, Kunde>  kunden;
 4     public Kunden() {
 5         kunden=new java.util.HashMap<Integer, Kunde>();}
 6     public Kunde anlegenKunde(int kundennummer, String kundenname){
 7         Kunde kunde=new Kunde(kundennummer, kundenname);
 8         kunden.put(new Integer(kundennummer), kunde);
 9         return kunde;
10     }
11 }
```

Die Anweisung in Zeile 34 (vgl. Listing 9b) der *CtrlDlgBank*-Klasse gibt dem Benutzer die Rückmeldung, dass ein neues *Kunde*-Objekt angelegt wurde. Für den *Beenden*-Button wurde auch eine Event-Handling-Methode geschrieben (vgl. Zeilen 45-47 in Listing 9b). Über die *dispose()*-Methode wird das *JDialog*-Objekt zerstört.

Entsprechend der Abb. 5 sind noch die Funktionalitäten *Konto anlegen*, *Ein-/-Auszahlungen durchführen*, *Überweisung durchführen* und *Kontostandsübersicht anzeigen* in analoger Weise umzusetzen. Hierzu beschreiben wir nicht alle Details, sondern greifen nur einzelne Spezialitäten heraus. Der gesamte Quellcode ist auf der Webseite. In Abb. 8 ist der Benutzerdialog für das Anlegen eines Giro- oder Sparkontos wiedergegeben. Dabei kommen so genannte Radio-Buttons zum Einsatz. In Swing handelt es sich um die Klasse *JRadioButton*, um zu gewährleisten, dass nur eine Option ausgewählt werden kann, wird noch die Klasse *ButtonGroup* benötigt. Im GUI-Editor von *NetBeans* ist folgendes Vorgehen nötig: Es ist ein *ButtonGroup*-Objekt auszuwählen und in den *JDialog* einzufügen. In unserem Fall wurde dieser der Name *ButtonGroupKontoArt* gegeben. Die *JRadioButton*-Objekte sind in gleicher Weise einzufügen, wie *JButton*-Objekte oder andere. Allerdings haben die *JRadioButton*-Objekte ein Attribut (property) *buttonGroup* und diesem Attribut ist dann bei beiden *JRadioButton*-Objekten das Objekt *ButtonGroupKontoArt*, das in einer Auswahlliste erscheint, zuzuordnen. Weiterhin ist es auch sinnvoll, bei einem der beiden *JRadioButton*-Objekte das Attribut *selected* anzuhaken, damit ist diese Option standardmäßig als ausgewählt gekennzeichnet. Damit entfällt die Plausibilitätsprüfung, ob eine der beiden ausgewählt wurde. Die weitere Oberflächenlogik ist vergleichbar mit dem Dialog *Kunde anlegen*. In der Klasse *CtrlDlgBank* sind die Methoden *jMenuItemKontoAnlegenActionPerformed()*, *jButtonKontoAnlegenActionPerformed()* und *jButtonKontoAnlegenBeendenActionPerformed()* hinzugekommen.

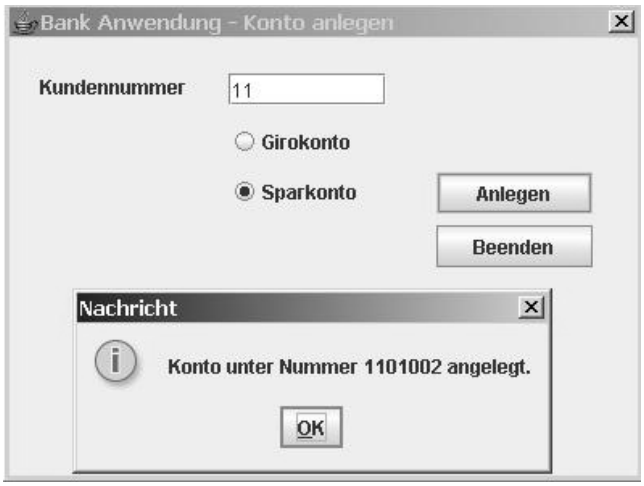

Abb. 8: Anwendungs-Dialog: Bank Anwendung – Konto anlegen

In Abb. 9 ist der Dialog für die Ein- und Auszahlungen abgebildet. Die grafische Oberfläche verwendet bekannte Komponenten und wird daher nicht im Detail

erläutert. Die Klasse *CtrlDlgBank* wurde um die Methoden *jMenuItemEinAuszahlungenActionPerformed()*, *jButtonKontostandAltActionPerformed()*, *jButtonAuszahlungActionPerformed()*, *jButtonEinzahlungActionPerformed()*, *showNeuerKontostand(double kontostand)* sowie *jButtonEinAuszahlungBeendenActionPerformed()* erweitert. In diesem Zusammenhang musste auch die Klasse *BankHandler* noch um die Methoden *getKontostand(int kontonummer)*, *auszahlenBetrag(int kontonummer, double betrag)* und *einzahlenBetrag(int kontonummer, double betrag)* ergänzt werden. Das sind die wesentlichen Änderungen, welche Sie unmittelbar im Code mit Ihrem Lösungsweg vergleichen können.

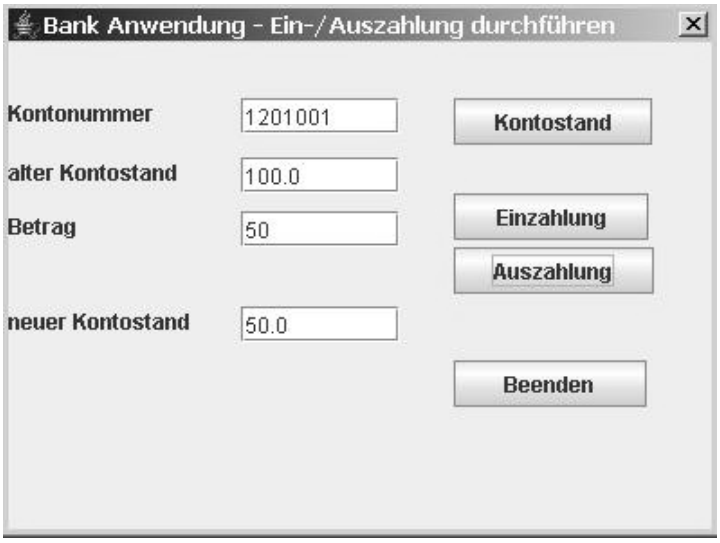

Abb. 9: Anwendungs-Dialog: Bank Anwendung – Ein-/Auszahlungen durchführen

Als nächste Transaktion schauen wir uns die Überweisung an, wobei wir in unserem Beispiel vereinfachend davon ausgehen, dass nur zwischen Konten der Kunden der gleichen Bank Überweisungen durchgeführt werden sollen. Abb. 10 zeigt den Anwendungs-Dialog. Die Besonderheit bei diesem Dialog ist die Möglichkeit, das Überweisungsdatum aus einem Kalender-Fenster auszuwählen und in das Textfeld Datum zu übernehmen. Die Details des Dialogs zur Datumsauswahl sollen hier nicht beschrieben werden. Hierfür sind im Paket *helper* die beiden Klassen *Kalender* und *KalenderView* implementiert. Dabei wird in hohem Maße auf die Funktionalität der abstrakten Klasse *java.util.Calendar* bzw. der konkreten Unterklasse *java.util.GregorianCalendar* zurückgegriffen. In der Klasse *CtrlDlgBank* wurden die Methoden *jMenuItemUeberweisungActionPerformed()*, *jButtonUeberweisenAc-*

tionPerformed(), *jButtonDatumAuswaehlenActionPerformed()* und *jButtonUeberweisungBeendenActionPerformed()* hinzugefügt. Die Klasse *BankHandler* wurde um die Methode *ueberweisen(int vonKontoNr, int nachKontoNr, double betrag, java.util.Date datum)* ergänzt.

Abb. 10: Anwendungs-Dialog: Bank Anwendung – Überweisung durchführen

Gemäß Abb. 5 setzen wir nun noch die letzte Transaktion *Kontostandsübersicht anzeigen* um. In Abb. 11 ist der Benutzerdialog abgebildet. Die Besonderheit ist die Tabellenanzeige. Hierfür verwenden wir die Swing-Klasse *JTable*. Ein *JTable*-Objekt stellt gemäß dem MVC-Muster (vgl. 7.4.1) die Daten eines *JTableModel*-Objektes dar. Hierfür haben wir eine eigene *JTableModel*-Klasse geschrieben (vgl. Listing 12), die von der Klasse *javax.swing.table.AbstractTableModel* abgeleitet ist. Die Methoden *getColumnCount(), getRowCount()* und *getValueAt(int rowIndex, int columnIndex*) sind in der abstrakten Oberklasse deklariert und werden in der Unterklasse implementiert. Zusätzlich werden die Methoden *getColumnName(int colNr)* und *setDaten(java.util.List<Konto> daten)* benötigt. Das *List*-Objekt *daten* verweist auf die Objekte, die als Zeilen in der Tabelle dargestellt werden. In unserem Fall sind dies Objekte vom Typ *Konto*. Der Array *col* wird für die Spaltenüberschriften verwendet. Die umfangreichste Methode ist *getValueAt()*, welche aus den *Konto*-Objekten die jeweiligen Attribute zurückgibt. Als nächstes wollen wir uns ansehen, wie wir das *JTableModel*-Objekt mit dem *JTable*-Objekt verbinden. Das Dialogbild in Abb. 11 setzt sich aus zwei Teilen zusammen. Bisher hatten wir die Interaktionselemente, wie *JLabel*- oder *JTextField*-Objekte direkt in die *Content-Pane* des *JDialog*-Objektes eingefügt. In unserem Fall fügen wir zuerst ein *JPanel*-Objekt ein. Im *JPanel*-Objekt

platzieren wir das *JLabel*- und *JTextField*-Objekt sowie die beiden *JButton*-Objekte. Anschließend fügen wir ein *JTable*-Objekt, das automatisch in ein *JScrollPane*-Objekt eingebettet ist, in das *JPanel*-Objekt ein. Ist dies geschehen so sind folgende Schritte zum Verbinden des *JTable*-Objektes mit dem *JTableModel*-Objekt notwendig: Selektieren des *JTable*-Objektes im Inspector, Auswahl der Eigenschaft *model* im Eigenschafts-Fenster und Links-Klick auf Editor-Symbol (rechts außen). Im angezeigten Fenster ist im *Select Mode* → *Form Connection* auszuwählen und der Radio-Button *user code* zu selektieren. Im Editor ist einzugeben *new JTableModel-KontoUebersicht()* (ohne Semikolon) und mit *ok* zu bestätigen. Wenn Sie die Klasse *DlgKontostandsUebersicht* ausführen erscheinen bereits die richtigen Spalten-Überschriften. In der Klasse *CtrDlgBank* wurde die Methode *startDlgBank()* um die Registrierung eines *ActionListener*-Objektes ergänzt, weiterhin wurden die Methoden *jButtonKontoUebersichtActionPerformed()* und *jButtonKontoUebersichtBeendenActionPerformed()* hinzugefügt. In Zeile 5 (vgl. Listing 13) wird eine Referenz auf das *JTableModel*-Objekt der Tabelle einer lokalen Variablen *model* zugewiesen. In Zeile 6 erhält das *List*-Objekt *daten* des *JTableModel*-Objektes eine Referenz auf das *List*-Objekt des ausgewählten Kunden, für den die Kontostands-Übersicht angezeigt werden soll. Über die Anweisung in Zeile 7 wird das *JTable*-Objekt darüber informiert, dass sich die in der Tabelle darzustellenden Daten geändert haben.

Abb. 11: Anwendungs-Dialog: Bank Anwendung – Kontostandsübersicht anzeigen

Listing 12: *JTableModelKontoUebersicht*-Klasse

```
1  package bankMitGUI;
2  public class JTableModelKontoUebersicht extends
     javax.swing.table.AbstractTableModel{
3      public JTableModelKontoUebersicht() {}
4      private java.util.List<Konto> daten;
5      private String[] col={"Kontoart", "Kontonummer", "Kontostand"};
6      public int getColumnCount() {
7          return col.length; }
8      public int getRowCount() {
9          if(getDaten()==null){
10             return 0; }
11         return getDaten().size();
12     }
13     public Object getValueAt(int rowIndex, int columnIndex) {
14         Konto konto=getDaten().get(rowIndex);
15         switch(columnIndex){
16             case 0:
17                 if(konto instanceof Girokonto){
18                     return "Giro";}
19                 else {
20                     return "Spar";}
21             case 1:
22                 return new Integer(konto.getKontonummer());
23             case 2:
24                 return new Double(konto.getKontostand());
25             default:
26                 return "Error";
27         }
28     }
29     public String getColumnName(int colNr){
30         return col[colNr]; }
31     public java.util.List<Konto> getDaten() {
32         return daten; }
33     public void setDaten(java.util.List<Konto> daten) {
34         this.daten = daten; }}
```

Listing 13: ***jButtonKontoUebersichtActionPerformed()*****-Methode in Klasse** ***CtrlDlgBank***

```
1 private void jButtonKontoUebersichtActionPerformed(){
2     try{
3         int kundennummer=Integer.parseInt(
            dlgKontostandsUebersicht.
            getJTextFieldKundennummer().getText());
4        Kunde kunde=bankHandler.getKunde(kundennummer);
5        JTableModelKontoUebersicht model=
           (JTableModelKontoUebersicht)
           dlgKontostandsUebersicht
           .getJTableKontoUebersicht().getModel();
6        model.setDaten(kunde.getKontenliste());
7        model.fireTableDataChanged();
8     }
9     catch(NumberFormatException e){
10        new javax.swing.JOptionPane()
            .showMessageDialog(dlgKontostandsUebersicht,
            "Bitte bei Kundennummer eine Zahl eingeben.");
11    }
12    catch(Exception e){
13        new javax.swing.JOptionPane().showMessageDialog(
            dlgKontostandsUebersicht, e.getMessage());
14        e.printStackTrace();
15 }}
```

Damit haben wir für unser vereinfachtes Bank-Beispiel eine grafische Oberfläche mit Mitteln des Swing-Frameworks erstellt. Nicht jeder Dialog war durch absolut neue Aspekte geprägt, damit hatten Sie auch die Möglichkeit des Übens im Umgang mit Java und dem GUI-Editor unserer IDE *NetBeans*. Im nächsten Schritt wollen wir unser Beispiel um die Anbindung an eine relationale Datenbank ergänzen.

5 Datenbankanbindung mit JDBC anhand eines Beispiels

Soll ein Java-Programm mit einer relationalen Datenbank arbeiten, so es notwendig, dass diese beiden Welten miteinander verbunden werden. Damit das Java-Programm unabhängig von dem konkreten Datenbank-Produkt entwickelt werden kann, steht dem Java-Programmierer mit **JDBC** (Java Database Connectivity) ein generisches Programmgerüst (framework) zur Verfügung. Im Paket *java.sql* sind

die Klassen und Methoden als so genannte JDBC API bereitgestellt. Neben der JDBC API ist ein datenbanksystemspezifischer **JDBC-Treiber** notwendig. Dieser Treiber versteht die JDBC-Befehle, übersetzt sie in Befehle des Datenbankmanagementsystems (DBMS) und leitet sie an das Datenbanksystem weiter. Die so genannte JDBC-ODBC-Bridge nimmt dabei eine Sonderstellung ein, da in diesem Fall ODBC (Open Database Connectivity) Treiber verwendet werden können, welche es für unterschiedliche Datenbanksysteme gibt. Damit ist es beispielsweise auch einfach möglich das Datenbanksystem *Microsoft Access* zu verwenden. Wir wollen, wie bereits in Abschnitt 2 ausgeführt, das DBMS *MySQL* verwenden, für das wir auch bereits den JDBC-Treiber unserem Projekt *javatutorial* zugeordnet haben. Abb. 12 zeigt den systematischen Zusammenhänge zwischen Java-Programm, JDBC-API, JDBC-Treiber und Datenbankmanagementsystem auf.

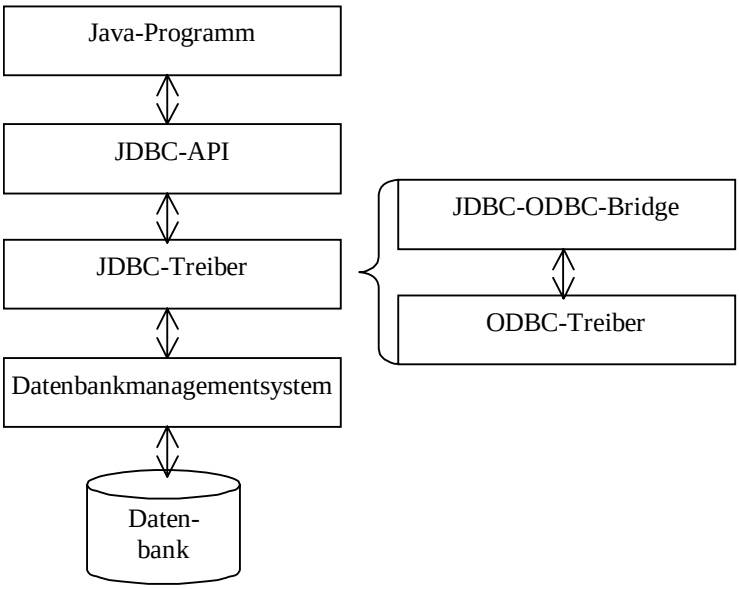

Abb. 12: Verbindung zwischen Java-Programm und relationaler Datenbank

In Abschnitt 2 haben wir mit Hilfe des Werkzeugs *Heidi*SQL bereits eine Datenbank *javatutorial* angelegt. In dieser Datenbank existiert auch schon eine Tabelle *Kunde* mit einem Datensatz. Auf diese Datenbank können wir mit dem Database Explorer von *NetBeans* direkt zugreifen. Im *NetBeans* Explorer ist das Register *Runtime* auszuwählen → *Databases* → *Drivers* → *Rechtsklick* → *Add Driver ...* → *Linksklick* → *Add ...* → wählen Sie aus dem Dateisystem Ihres Rechners die Datei *mysql-connector-java-5.0.4-bin.jar* mit dem MySQL-Driver aus → *ok*. Damit kann eine Verbindung aufgebaut werden: Selektieren von *MySQL* (MM.MySQL driver) →

Rechtsklick → *Connect Using ...* → im Dialog *New Database Connection* ist im Feld Database URL *jdbc:mysql://localhost:3307/javatutorial* und im Feld User Name *root* einzutragen. Falls Sie ein Passwort vergeben haben, ist dies ebenfalls einzugeben. Daraufhin wird ein Knoten für die Verbindung angezeigt, wenn Sie diesen erweitern, finden Sie unter *Tables* die Tabelle *Kunde*. Mit einem *Rechtsklick* können Sie die Funktion *View Data* aus dem Umgebungsmenu auswählen und Ihnen wird der Kunde 1 mit dem Namen Arm angezeigt. Wie Sie im Umgebungsmenü sehen, können auch Spalten und Tabellen hinzugefügt werden und beliebige SQL-Befehle ausgeführt werden.

Nun wollen wir eine Klasse mit einer *main()*-Methode schreiben, die uns ebenfalls den Inhalt der Tabelle *Kunde* anzeigt. Zuvor wollen wir jedoch noch einen weiteren Kunden hinzufügen und dessen Namen ändern. Damit wir die einfache *main()*-Methode mehrfach ausführen können, löschen wir nach der Anzeige den eingefügten Kunden wieder. Nachfolgend werden die wesentlichen Anweisungen der Listings 14a und 14b erläutert:

- **Treiber und Treiber-Manager:** In Zeile 15 wird ein Treiber instanziiert und das Treiber-Objekt wird durch Anweisungen im Konstruktor der Treiber-Klasse bei der Klasse *DriverManager* registriert. Der Treiber-Name ist in Zeile 7 definiert.
- **Verbindungsaufbau:** Zum Aufbauen einer Verbindung mit dem Datenbanksystem wird die statische Methode *getConnection()* der Klasse *DriverManager* verwendet. Dies geschieht in Zeile 17. In Zeile 8 ist die Zieldatenbank in URL-Schreibweise angegeben: *jdbc* ist das Protokoll, *mysql* ist das Subprotokoll, die Datenbank *javatutorial* befindet sich auf dem gleichen Rechner, daher *localhost*, als Parameter werden der *user* und das *password* mitgegeben.
- **Vorbereitete Anweisung (PreparedStatement):** *PreparedStatement*-Objekte werden dann verwendet, wenn eine Parametrisierung von SQL-Ausdrücken zweckmäßig ist. Es können dabei Werte, in unserem Fall von Kundennummer und Name, in die Insert-Anweisung (vgl. Zeile 18), übergeben werden. In Zeile 19 wird ein *PreparedStatement*-Objekt vom *Connection*-Objekt über die Methode *prepareStatement()* geholt. Dabei wird der SQL-Ausdruck als Zeichenketten-Parameter mitgegeben, der allerdings Fragezeichen als Platzhalter enthält. In den Anweisungen der Zeilen 20 und 21 werden die Parameterwerte mit typkonformen Methoden übergeben und in Zeile 22 wird die Ausführung der SQL-Insert-Anweisung veranlasst. In diesem Fall wäre die vorbereitete Anweisung nicht unbedingt notwendig gewesen. Insbesondere wird sie verwendet, wenn z.B. mehrere Zeilen einer Tabelle geschrieben werden, bei denen die Spaltenwerte jeweils unterschiedlich sind.
- **Einfache SQL-Anweisung**: Zur Ausführung einfacher SQL-Anweisungen werden *Statement*-Objekte verwendet. Da wir in Zeile 21 den Namen des

Kunden falsch geschrieben haben, wird in Zeile 23 ein SQL-Ausdruck zur Richtigstellung des Namens mittels einer SQL-Update-Anweisung definiert und in Zeile 24 ausgeführt. Als nächstes sollen die Zeilen der Tabelle *Kunde* angezeigt werden. Hierzu ist ein SQL-Ausdruck in Form einer Select-Anweisung notwendig (vgl. Zeile 25). Die Methode *executeQuery()* liefert eine Ergebnistabelle in einem *ResultSet*-Objekt zurück. Beim Umgang mit *ResultSet*-Objekten ist darauf zu achten, dass es zu einem *Statement*-Objekt höchstens ein *ResultSet*-Objekt gibt und dass auf jeden Wert eines *ResultSet*-Objektes höchstens einmal zugegriffen werden darf. Auf die Daten des *ResultSet*-Objektes kann mit Hilfe eines Cursors zugegriffen werden. Die Methode *next()* in Zeile 27 liefert beim erstmaligen Aufruf einen gültigen Cursor. Über jeden Aufruf von *next()* wird der Cursor um eine Position weitergeschaltet. Ist der Cursor am Ende der Ergebnismenge angekommen, wird *false* ansonsten *true* zurückgegeben. Über typkonforme *get*-Methoden können nun durch Übergabe des entsprechenden Spaltennamens der Datenbanktabelle die Werte übernommen werden (vgl. Zeilen 28 und 29). Die Zeile 30 soll nur andeuten, dass aus den eingelesenen Werten wieder ein *Kunde*-Objekt erzeugt werden kann. Zeile 31 gibt die Attributwerte aus dem *Kunde*-Objekt auf der Konsole aus. Die Zeilen 33 und 34 dienen dazu, dass die eingefügte Kunden-Zeile in der Datenbanktabelle wieder gelöscht wird. Dies hat einerseits den Zweck, auch ein Beispiel für die Delete-Anweisung zu zeigen und andererseits würde ein mehrmaliges Ausführen dieser Klasse sonst zu einem Fehler führen, da die Kundennummer als Primärschlüssel definiert ist und damit nicht mehrfach eine Zeile mit der Kundennummer 2 eingefügt werden könnte.

- **Schließen der Statement-, ResultSet- und Connection-Objekte:** In den Zeilen 40 ff. werden diese Objekte in einem *finally*-Block geschlossen. Der *finally*-Block wird auf jeden Fall ausgeführt, egal ob eine Ausnahmebedingung bei den SQL-Anweisungen aufgetreten ist oder nicht.

Listing 14a: Einfaches Beispiel für Datenbankanbindung

```
 1 package bankMitGUIUndDB;
 2 import java.sql.*;
 3 public class TestDbKunde {
 4     public TestDbKunde() {
 5     }
 6     public static void main(String[] args) {
 7         String treiber="com.mysql.jdbc.Driver";
 8         String
 9         Connection con=null;
10         Statement stmt=null;
11         PreparedStatement pstmt=null;
12         ResultSet rs=null;
13         String sql=null;
14         try{
15            Class.forName(treiber);
16             con=DriverManager.getConnection(url);
17             stmt=con.createStatement();
18             sql="Insert into Kunde (Kundennummer, Name) values (?,?);";
19             pstmt=con.prepareStatement(sql);
20             pstmt.setInt(1, 2);
21             pstmt.setString(2, "Riech");
22             pstmt.executeUpdate();
23             sql="Update Kunde set Name=\"Reich\"
                 where kundennummer=2;";
24             stmt.executeUpdate(sql);
25             sql="Select * from Kunde;";
26             rs=stmt.executeQuery(sql);
27             while(rs.next()){
28                 int kundennummer=rs.getInt("kundennummer");
29                 String name=rs.getString("name");
30                 Kunde kunde=new Kunde(kundennummer, name);
31                 System.out.println(kunde
                     .getKundennummer()+" "+kunde.getName());
32             }
```

Listing 14b: Einfaches Beispiel für Datenbankanbindung

```
33              sql="Delete from Kunde where Kundennummer=2;";
34              stmt.executeUpdate(sql);
35          }
36          catch(Exception e){
37              System.out.println("SQL-Fehler: "+e.getMessage());
38              e.printStackTrace();
39          }
40          finally{
41              try{
42                  stmt.close();
43                  pstmt.close();
44                  rs.close();
45                  con.close();
46              }
47              catch(SQLException e){
48                  System.out.println("SQL-Fehler: "+e.getMessage());
49                  e.printStackTrace();
50  }}}}
```

Anhand dieses einfachen Beispiels sollten die Grundlagen der Anbindung einer Datenbank an ein Java-Programm demonstriert werden.

In unserem Bank-Beispiel geht es jetzt darum, überall dort die Datenbank einzubinden, wo entweder Attributwerte von Objekten gespeichert bzw. gespeicherte Attributwerte in Objekten verwendet werden sollen. In der Datenbank javatutorial sind hierzu zusätzliche Tabellen anzulegen. Dies wird an den entsprechenden Stellen im Text erläutert. Eine Alternative besteht darin, das Skript *javatutorial.sql* von der Webseite zu nehmen und es entsprechend der Beschreibung unter 9.3.1 auszuführen und damit die Datenbank samt Beispieldaten anzulegen. Im *NetBeans* legen Sie ein neues Java-Projekt *bankMitGUIUndDB* an und kopieren die Klassen aus dem Paket *bankMitGUI*. In diesem Paket werden die notwendigen Ergänzungen und Änderungen vorgenommen. Beginnen wollen wir mit dem Anlegen eines neuen Kunden. Der Tabellenaufbau ist äußerst einfach und existiert ja bereits in unserer Datenbank *javatutorial*. Allerdings stellt sich die Frage, wie wir gewährleisten, dass die Kundennummer die Bedingungen des Primärschlüssels erfüllt. Bisher haben wir die Kundennummer durch den Benutzer eingeben lassen. Die Kundennummer hat in unserem Beispiel eine reine Identifikationsfunktion. Vor diesem

Hintergrund können wir die Nummer auch vom System vergeben lassen. Wir wollen dies mit Hilfe einer eigenen Schlüsseltabelle tun (vgl. 7.5.3).

Hierfür haben wir im Paket *database* eine Klasse *Sequence* geschrieben, die wir gemäß dem *Singleton*-Muster (vgl. 8.3.3) implementiert haben. Die Klasse *Sequence* verfügt über eine Methode *getNextIdFor(String table)*, welche für die im Parameter spezifizierte Tabelle den zuletzt vergebenen Schlüsselwert zurückliefert. Hierzu haben wir auch in der Datenbank eine Tabelle *sequence* mit den beiden Spalten *Name* und *Id* angelegt. Das Anlegen der Datenbanktabelle kann sehr einfach über den Database Explorer in *NetBeans* erfolgen. Wenn Sie sich diese *Sequence*-Klasse anschauen, werden Sie feststellen, dass wir noch ein paar weitere Hilfsklassen im Paket *database* definiert haben. Die Klasse *DbConnection* dient dazu, eine Datenbankverbindung aufzubauen, weiterhin können *Statement*-Objekte und *PreparedStatement*-Objekte erzeugt werden. Die Klasse *DbConnection* implementiert das Interface *Constants*, wobei dieses lediglich die Konstanten *DB_DRIVER* und *DB_URL* enthält und über die *implements*-Klausel der Klasse zur Verfügung stehen. Darüber hinaus wurden die beiden Hilfsklassen *DbReader* und *DbWriter* angelegt. Diese verfügen über einfache Methoden zum Ausführen von SQL-Anweisungen. Mit Hilfe dieser Klassen können die Datenbankzugriffe vereinfacht umgesetzt werden. Das *Sequence*-Objekt verfügt über eine eigene Datenbankverbindung, bei welcher der automatische *Commit*-Modus ausgeschaltet wird. Standardmäßig wird jede SQL-Anweisung als eine Transaktion behandelt. Wird der *Auto-Commit*-Modus auf *false* gesetzt, so können mehrere Anweisungen zu einer Transaktion dadurch zusammengefasst werden, dass nach den betreffenden Anweisungen die Transaktion durch ein explizites *commit* abgeschlossen wird. Im Fehlerfall erfolgt ein Zurücknehmen aller bereits abgearbeiteten Anweisungen mit Hilfe der *rollback()*-Operation. Die Anwendung der Transaktionslogik findet sich in der Methode *getNextIdFor()* in der Klasse *Sequence*. Damit fällt die Eingabe der Kundennummer weg, dies schlägt sich in den Klassen *DlgKundeAnlegen*, *CtrlDlgBank* und *BankHandler* nieder. Die Klasse *Kunden* haben wir durch die Klasse *Mapper* ersetzt. Die Aufgabe dieser Klasse ist die Kapselung aller Datenbankzugriffe unserer Fachklassen. Darüber hinaus nimmt sie auch die Funktion einer **Identity Map** wahr (vgl. 7.5.2) in dem das *HashMap*-Objekt *kunden* alle Referenzen auf die aktuell geladenen *Kunde*-Objekte aufnimmt. Die Methode *anlegenKunde()* verwendet die oben erläuterte *getNextIdFor()*-Methode (vgl. Zeile 2 in Listing 15). In Zeile 5 wird die *speichernKunde()*-Methode aufgerufen, welche in den Zeilen 8-21 implementiert ist. Die Implementierung entspricht weitestgehend den Anweisungen in den Zeilen 18 ff. in Listing 14.

Listing 15: Methoden *anlegenKunde()* und *speichernKunde()* in der Klasse *Mapper*

```
 1 public Kunde anlegenKunde(String kundenname){
 2     int kundennummer=Sequence.getSequence().getNextIdFor("Kunde");
 3     Kunde kunde=new Kunde(kundennummer, kundenname);
 4     kunden.put(new Integer(kundennummer), kunde);
 5     speichernKunde(kunde);
 6     return kunde;
 7 }
 8 public void speichernKunde (Kunde kunde){
 9     try{
10         String sql="Insert into Kunde
             (Kundennummer, Name) values (?,?); ";
11         java.sql.PreparedStatement pstmt=writer.insert(sql);
12         pstmt.setInt(1, kunde.getKundennummer());
13         pstmt.setString(2, kunde.getName());
14         pstmt.executeUpdate();
15     }
16     catch(Exception e){
17         new javax.swing.JOptionPane().showMessageDialog(
             null, "Datenbankfehler: "+e.getMessage()
             +". Datenbankadministrator benachrichtigen.");
18         e.printStackTrace();
19         System.exit(0);
20 }}
```

Das Anlegen eines neuen Kontos erfordert Anpassungen bei der Methode *anlegen-Konto()* der Klasse *BankHandler* (vgl. Listing 16), der Methode *getKunde()* der Klasse *Mapper* und die neuen Methoden *loadKunde()* und *speichernKonto()* in der Klasse *Mapper* (vgl. Listing 17a und b). Die Methode *getKunde()* der Klasse *Mapper* wird um einen Datenbankzugriff erweitert. Wenn das gesuchte *Kunde*-Objekt nicht in der HashMap *kunden* referenziert ist, so werden die Kundendaten von der Datenbank geladen (vgl. Zeile 4 in Listing 17a). Dies übernimmt die neue Methode *load-Kunde()* (vgl. Zeilen 12-21 in Listing 17a). Das *BankHandler*-Objekt übernimmt die Kontrolle über die Persistenz. In Zeile 4 (vgl. Listing 16) wird die Methode *speichernKonto()* der Klasse *Mappper* aufgerufen. Hinsichtlich der Abbildung der Vererbungsstruktur im Objektmodell haben wir uns für das **Single Table Inheritance**-Muster entschieden (vgl. Abb. 7.10). Für unser einfaches Beispiel haben wir auch keine extra **Mapper**-Klasse für die Klasse *Konto* erstellt, sondern im Sinne des **De-**

pendent Mapping (vgl. 7.5.3) werden die notwendigen Methoden in der Klasse *Mapper* implementiert.

Listing 16: Methode *anlegenKonto()* in der Klasse *BankHandler*

```
1 public Konto anlegenKonto(int
    kundennummer, int kontoart) throws Exception{
2     Kunde kunde=mapper.getKunde(kundennummer);
3     Konto konto= kunde.anlegenKonto(kontoart);
4     mapper.speichernKonto(konto, kundennummer);
5     return konto;
6 }
```

Listing 17a: Neue Methoden in Klasse *Mapper* zum Anlegen eines Kontos

```
1 public Kunde getKunde(int kundennummer) throws Exception{
2     Kunde kunde=kunden.get(new Integer(kundennummer));
3     if(kunde==null){
4         kunde=loadKunde(kundennummer);
5         if (kunde==null){
6             throw new Exception("Unter "+kunden-
                  nummer + " kein Kunde gefunden.");
7         }
8         kunden.put(new Integer(kundennummer), kunde);
9      }
10    return kunde;
11 }
12 private Kunde loadKunde(int kundennummer) throws Exception{
13    Kunde kunde=null;
14    String sql="Select * from Kunde where
          kundennummer="+kundennummer+";";
15    java.sql.ResultSet rs=reader.query(sql);
16    if(rs.next()){
17        String name=rs.getString("Name");
18        kunde=new Kunde(kundennummer, name);
19    }
20    return kunde;
21 }
```

Listing 17b: Neue Methoden in Klasse *Mapper* zum Anlegen eines Kontos

```
22 public void speichernKonto(Konto konto,
       int kundennummer) throws Exception{
23     String sql="Insert into Konto (kontonummer,
         kontostand, kategorie, kundennummer) values (?,?,?,?);";
24     java.sql.PreparedStatement pstmt=writer.insert(sql);
25     pstmt.setInt(1, konto.getKontonummer());
26     pstmt.setDouble(2, konto.getKontostand());
27     pstmt.setString(3, konto instanceof Girokonto ? "G" : "S");
28     pstmt.setInt(4, kundennummer);
29     pstmt.executeUpdate();
30 }
```

Listing 18: Methoden *getKontostand(), auszahlenBetrag(), einzahlenBetrag()* in Klasse *BankHandler*

```
 1 public double getKontostand(int kontonummer) throws Exception {
 2     int kundennummer=kontonummer/100000;
 3     Kunde kunde=mapper.getKundeMitKonten(kundennummer);
 4     return kunde.getKontostand(kontonummer);
 5 }
 6 public double auszahlenBetrag(int kontonummer,
       double betrag) throws Exception {
 7     int kundennummer=kontonummer/100000;
 8     Kunde kunde=mapper.getKunde(kundennummer);
 9     double kontostand= kunde.auszahlenBetrag(kontonummer, betrag);
10     mapper.updateKontostand(kontonummer, kontostand);
11     return kontostand;
12 }
13 public double einzahlenBetrag(int kontonummer,
       double betrag) throws Exception {
14     int kundennummer=kontonummer/100000;
15     Kunde kunde=mapper.getKunde(kundennummer);
16     double kontostand= kunde.einzahlenBetrag(kontonummer, betrag);
17     mapper.updateKontostand(kontonummer, kontostand);
18     return kontostand;
19 }
```

Damit auch die Konsequenzen von Ein- und Auszahlungsvorgängen in der Datenbank richtig widergespiegelt werden, sind einerseits Anpassungen in den Methoden *getKontostand()*, *auszahlenBetrag()* und *einzahlenBetrag()* in der Klasse *BankHandler* notwendig (vgl. Listing 18), andererseits müssen in der Klasse *Mapper* die zwei Methoden *getKundeMitKonten()* und *updateKontostand()* neu entwickelt werden. In Zeile 3 (vgl. Listing 18) wird statt der alten Methode *getKunde()*, die neue Methode *getKundeMitKonten()* aufgerufen. Die Methoden *auszahlenBetrag()* und *einzahlenBetrag()* werden in den Zeilen 10 und 17 durch den Aufruf der Methode *updateKontostand()* ergänzt.

In der Klasse *Mapper* wird neue Funktionalität ergänzt, die es erlaubt, ein *Kunde*-Objekt einschließlich der zugehörigen Konten zurückzuliefern (vgl. *getKundeMitKonten()* in den Listings 19a und 19b). Die Methode *loadKunde()* übernimmt den eigentlichen Datenbankzugriff und Instanziierung der *Konto*-Objekte. In den Zeilen 15-20 (vgl. Listings 19a und b) wird das gespeicherte Kategorie-Kennzeichen verwendet, um den richtigen Konstruktor-Aufruf zu veranlassen. Die Methode *updateKontostand()* stellt die Funktionalität bereit, so dass die ein- bzw. auszahlungsbedingte Kontostandsänderung auch auf der Datenbank durchgeführt werden kann. Veranlasst wird diese Änderung durch das *BankHandler*-Objekt (vgl. Zeilen 10 und 17 in Listing 18).

Listing 19a: Neue Methoden in Klasse *Mapper* zur Abwicklung von Ein-/Auszahlungen

```
 1 public Kunde getKundeMitKonten(int kundennummer) throws Exception{
 2     Kunde kunde=getKunde(kundennummer);
 3     if(kunde.getKontenliste().size()==0){
 4       loadKonten(kunde);
 5     }
 6     return kunde;
 7 }
 8 private void loadKonten(Kunde kunde) throws Exception{
 9     String sql="Select * from Konto where
         kundennummer ="+kunde.getKundennummer()+";";
10     Konto konto=null;
11     java.sql.ResultSet rs=reader.query(sql);
12     while(rs.next()){
13         int kontonummer=rs.getInt("Kontonummer");
14         double kontostand=rs.getDouble("Kontostand");
15         String kategorie=rs.getString("Kategorie");
```

Listing 19b: Neue Methoden in Klasse *Mapper* zur Abwicklung von Ein-/Auszahlungen

```
16          if(kategorie.equals("S")){
17              konto=new Sparkonto(kontonummer);
18          }
19          else{
20              konto=new Girokonto(kontonummer);
21          }
22          konto.setKontostand(kontostand);
23          kunde.hinzufuegenKonto(konto);
24      }
25  }
26  public void updateKontostand(int kontonummer,
      double kontostand) throws Exception{
27      String sql="Update Konto set Kontostand=? where kontonummer=?;";
28      java.sql.PreparedStatement pstmt=writer.update(sql);
29      pstmt.setDouble(1, kontostand);
30      pstmt.setInt(2, kontonummer);
31      pstmt.executeUpdate();
32  }
```

Die Speicherung des Überweisungsvorgangs macht es notwendig, dass wir eine neue Tabelle *Ueberweisung* in der Datenbank definieren. Technisch können wir das wieder über den Datenbank-Browser von *NetBeans* ausführen. Als Spalten übernehmen wir die Kontonummern des Quell- und Zielkontos (*vonKonto* bzw. *nachKonto*), den Betrag und das Überweisungsdatum. Als Primärschlüssel führen wir entsprechend dem **Identity-Field**-Muster (vgl. 7.5.3) eine Id ein, deren Inhalt wir wie bei der Klasse *Kunde* über eine Schlüsseltabelle in der Datenbank vergeben. Hierzu fügen wir in der Tabelle *sequence* eine Zeile für die Tabelle *Ueberweisung* hinzu und ergänzen das Attribut *id* in der Klasse *Ueberweisung* und dem entsprechenden Konstruktor. In der Methode *ueberweisen()* der Klasse *BankHandler* mussten kleinere Ergänzungen durchgeführt werden (vgl. Listing 20). In Zeile 8 wird die Überweisungsnummer generiert, in Zeile 9 wird das neue Objekt vom Typ *Ueberweisung* erzeugt und in Zeile 11 wird die Speicherung veranlasst. Die Speicherung wird in der Klasse *Mapper* mit der Methode *speichernUeberweisung()* implementiert. Eine kleine Besonderheit ergibt sich in der Zeile 9 in Listing 21. Das Datum ist vom Typ *java.util.Date*, zur Speicherung in einer relationalen Datenbank erfordert die JDBC-Methode den Datentyp *java.sql.Date*. Daher ist eine Konvertierung notwendig. Neben der Speicherung des Überweisungsobjektes werden auch

die veränderten Kontostände mittels der bereits implementierten Methode *update-Kontostand()* in die Datenbank übernommen (vgl. Zeilen 11 und 12).

Listing 20: Methode *ueberweisen()* in der Klasse *BankHandler*

```
 1 public void ueberweisen(int vonKontoNr, int nachKontoNr,
     double betrag, java.util.Date datum) throws Exception {
 2     int vonKundennummer=vonKontoNr/100000;
 3     int nachKundennummer=nachKontoNr/100000;
 4     Kunde vonKunde=mapper.getKundeMitKonten(vonKundennummer);
 5     Kunde nachKunde=mapper.getKundeMitKonten(nachKundennummer);
 6     Konto vonKonto=vonKunde.getKonto(vonKontoNr);
 7     Konto nachKonto=nachKunde.getKonto(nachKontoNr);
 8     int id=Sequence.getSequence().getNextIdFor("Ueberweisung");
 9     Ueberweisung ueberweisung=new Ueberweisung(
         id, vonKonto,nachKonto,betrag, datum);
10     ueberweisung.durchfuehrenUeberweisung();
11     mapper.speichernUeberweisung(ueberweisung);
12 }
```

Für die Umsetzung der letzten Funktion *Kontostandsübersicht anzeigen* ist nur eine kleine Änderung in der Klasse *BankHandler* vorzunehmen. Die Nachricht *mapper.getKunde(kundennummer)* ist durch die Nachricht *mapper.getKundeMitKonten-(kundennummer)* in der Methode *getKunde()* zu ersetzen. Damit haben wir unsere kleine Bankanwendung mit der Datenbank so verbunden, dass alle Daten unserer Objekte über die relationale Datenbank persistent gehalten sind.

Listing 21: Methode *speichernUeberweisung()* in der Klasse *Mapper*

```
 1 public void speichernUeberweisung (Ueberweisung ueberweisung){
 2     try{
 3         String sql="Insert into Ueberweisung (id, vonKontoNummer,
             nachKontoNummer, betrag, datum) values (?,?,?,?,?); ";
 4         java.sql.PreparedStatement pstmt=writer.insert(sql);
 5         pstmt.setInt(1, ueberweisung.getId());
 6         pstmt.setInt(2, ueberweisung.getVonKonto().getKontonummer());
 7         pstmt.setInt(3, ueberweisung.getNachKonto().getKontonummer());
 8         pstmt.setDouble(4, ueberweisung.getBetrag());
 9         pstmt.setDate(5, new java.sql.Date(
             ueberweisung.getDatum().getTime()));
10         pstmt.executeUpdate();
11         updateKontostand(ueberweisung.getVonKonto()
       .getKontonummer(),ueberweisung.getVonKonto().getKontostand());
12         updateKontostand(ueberweisung.getNachKonto()
       .getKontonummer(),ueberweisung.getNachKonto().getKontostand());
13     }
14     catch(Exception e){
15         new javax.swing.JOptionPane().showMessageDialog(
             null, "Datenbankfehler: "+e.getMessage()+ ".
             Datenbankadministrator benachrichtigen.");
16         e.printStackTrace();
17         System.exit(0);
18     }
19 }
```

6 Web-Anwendung mit JavaServer Pages und Servlet

In diesem Unterkapitel werden wir uns anhand unseres Bank-Beispiels mit Java-Technologien beschäftigen, welche für die Implementierung einer einfachen Web-Anwendung verwendet werden. Nach einem kurzen Überblick über die technischen Zusammenhänge, werden wir die beiden Anwendungsfälle *Überweisung durchführen* und *Kontostandsübersicht anzeigen* als Web-Anwendungen implementieren.

6.1 Grundlagen zur verwendeten Technologie

Eine **Web-Anwendung** ist durch den Browser als Web-Client und einen Web-Server gekennzeichnet. Der Web-Client kommuniziert mit dem Web-Server über HTTP-Anfragen (http-request) und HTTP-Antworten (http-response). Durch die Eingabe und das Abschicken einer URL (uniform resource locator) im Browser wird grundsätzlich eine Datei auf dem Web-Server über das Protokoll HTTP angefordert. Dies kann eine einfache HTML-Seite sein, die grundsätzlich bei jedem Abruf den gleichen Inhalt aufweist und daher auch als statische Seite bezeichnet wird. Wir wollen jedoch eine dynamische, interaktive Web-Anwendung entwickeln, die **Servlets** und **JavaServer Pages** (JSP) nutzt. Dynamisch bedeutet, dass die Ergebnisse dynamisch entsprechend der Anforderung des Clients erzeugt werden, z.B. entsprechend der eingegebenen Kundennummer wird die dazugehörige Kontoübersicht im Browser angezeigt. Damit ist auch schon die Interaktivität angesprochen, da durch die Kundennummer-Eingabe die Verarbeitung auf dem Server beeinflusst wird. Die Web-Orientierung ist darin begründet, dass bewährte Internet-Standards verwendet werden, so z.B. http-request/response als Netzprotokoll und HTML als Darstellungsformat. Der Browser als Web-Client wird im Gegensatz zu einem Swing-GUI vielfach als Thin-Client bezeichnet. Als Web-Server verwenden wir **Tomcat** in der Version 5. Tomcat ist wird auch als Servlet-Container bezeichnet und stellt eine Laufzeitumgebung dar, in der Servlets verwaltet und aufgerufen werden. Ein Servlet ist eine Java-Klasse, welche von der Klasse HttpServlet abgeleitet wird. Die serverseitigen Kommunikationspartner einer Web-Anwendung sind Objekte der Servlet-Klassen, die man unpräzise auch als Servlets bezeichnet. Ein Servlet wird entweder bei der ersten Anforderung oder unmittelbar beim Starten des Servlet-Containers geladen. Über den parameterlosen Standard-Konstruktor wird ein Servlet-Objekt erzeugt und vom Container wird die Methode *init()* aufgerufen. Nach diesen Schritten verarbeitet das Servlet mittels der überschriebenen Methoden *doGet()* bzw. *doPost()* entsprechend den Methoden *GET* und *POST* des HTTP-Protokolls die Anforderungen an den Server. In dem so genannten Deployment-Deskriptor (*web.xml*-Datei) wird das Servlet dem Container bekannt gemacht. Im Normalfall existiert genau ein Objekt von jeder Servlet-Klasse. Jede Anforderung wird in einem eigenen Thread ausgeführt, so dass ein Servlet-Objekt alle Anforderungen parallel bedient. Dabei können sich Anforderungen gegenseitig unterbrechen. Falls sich daraus Probleme ergeben können, können kritische Code-Teile bzw. Methoden über **synchronized** abgesichert werden.

Listing 22: Einfaches Servlet-Beispiel

```
 1 package servlets;
 2 import java.io.*;
 3 import java.net.*;
 4 import javax.servlet.*;
 5 import javax.servlet.http.*;
 6 public class TestServlet extends HttpServlet {
 7     public void init(ServletConfig config) throws ServletException {
 8         super.init(config); }
 9     public void destroy() {}
10     protected void processRequest(
         HttpServletRequest request,
         HttpServletResponse response)
11     throws ServletException, IOException {
12         response.setContentType("text/html");
13         PrintWriter out = response.getWriter();
14         // TODO output your page here
15         out.println("<html>");
16         out.println("<head>");
17         out.println("<title>Servlet</title>");
18         out.println("</head>");
19         out.println("<body>");
20          out.println("Das ist mein erstes Servlet");
21         out.println("</body>");
22         out.println("</html>");
23         out.close(); }
24     protected void doGet(HttpServletRequest
         request, HttpServletResponse response)
25     throws ServletException, IOException {
26         processRequest(request, response); }
27     protected void doPost(HttpServletRequest
         request, HttpServletResponse response)
28     throws ServletException, IOException {
29         processRequest(request, response);}
30     public String getServletInfo() {
31         return "Short description";}}
```

Wenn wir *NetBeans* verwenden, müssen wir uns um die Installation des Servlet-Containers *Tomcat* nicht kümmern, da dieser schon Bestandteil unserer Installation ist. Wir können in *NetBeans* einfach ein neues Projekt *WebTest* anlegen. Im entsprechenden Wizard wählen wir unter *Web* die Option *Web Application*. Im Projekt-Browser wird eine Struktur erzeugt, welche insbesondere den Ordner *WEB-INF* enthält, in diesem ist auch bereits die oben erwähnte *web.xml*-Datei. Unter *Source Packages* legen wir ein neues Paket *servlets* anlegen. In diesem Ordner erstellen wir die Klasse *TestServlet* mit Hilfe des Klassen-Wizards (vgl. Listing 22).

An dem generierten Rahmen wurden nur die Kommentarzeichen in den Zeilen 18 ff. entfernt und die Zeile 23 hinzugefügt. Im Ordner *Web Pages* legen wir noch die *index.html*-Datei an (vgl. Listing 23). Diese HTML-Datei richtet eine Anfrage (einen *request*) an den Server und zwar an das Servlet *TestServlet*.

Listing 23: Einfach *index.html*-Datei zumAufruf des Servlets

```
1 <!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
2 <html>
3   <head>
4     <title></title>
5   </head>
6   <body>
7    <a href="TestServlet">TestServlet starten</a>
8   </body>
9 </html>
```

Listing 24: Ausschnitt aus *web.xml*

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <web-app version="2.4" xmlns="http://java.sun.com/xml/ns/j2ee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
    http://java.sun.com/xml/ns/j2ee/web-app_2_4.xsd">
3   <servlet>
4     <servlet-name>TestServlet</servlet-name>
5     <servlet-class>servlets.TestServlet</servlet-class>
6   </servlet>
7   <servlet-mapping>
8     <servlet-name>TestServlet</servlet-name>
9     <url-pattern>/TestServlet</url-pattern>
10  </servlet-mapping>
```

Im Listing 24 ist ein Ausschnitt aus der *web.xml* wiedergegeben. Der <*servlet*>-Tag und der <*servlet-mapping*>-Tag wurde von *Netbeans* generiert und ermöglicht erst die Funktionsfähigkeit der Zeile 7 in Listing 23.

Im Listing 25 ist die HTML-Datei wiedergegeben, die als Ergebnis der Ausführung der *doGet()*-Methode in Zeile 25 (vgl. Listing 22) bzw. der *processRequest()*-Methode in den Zeilen 10 ff. im Servlet als Antwort an den Web-Browser zurückgeliefert wird. Da das Servlet eine normale Java-Klasse ist, kann hier natürlich auch wirklich etwas dynamisch zurückgeliefert werden, z.B. das Ergebnis einer Datenbankabfrage.

Listing 25: Response vom Servlet

```
1 <html>
2 <head>
3 <title>Servlet</title>
4 </head>
5 <body>
6 Das ist mein erstes Servlet
7 </body>
8 </html>
```

Die Verwendung der Servlet-Technik in dieser Form ist dadurch eingeschränkt, dass der gesamte Seitenaufbau im Servlet festgelegt wird.

Eine andere Technik ist die JavaServer Page (**JSP**), welche im Kern eine **HTML**-Seite mit eingebetteten Java-Anweisungen darstellt. Die Java-Anweisungen werden durch besondere Markierungen (**tags**) innerhalb von HTML gekennzeichnet. Im Endeffekt wird eine JSP von einem JSP-Server (bei uns dem JSP-Compiler *Jasper* der Servlet-Umgebung *Tomcat*) in ein Java-Servlet übersetzt und anschließend ausgeführt. Von daher sind JSPs im Endeffekt nichts anderes als Servlets. Allerdings können sie entwickelt werden wie HTML-Seiten ergänzt um Java-Anweisungen. Zur Verdeutlichung der Aussagen machen wir ein einfaches Beispiel. Wir implementieren eine Klasse *Kunde* in dem anzulegenden Paket *beans*, wobei der Standardkonstruktor die Attribute *kundennummer* und *name* mit den Standardwerten *99* und *TestKunde* initialisiert. Weiterhin schreiben wir eine JSP-Seite (*index.jsp*), die ja bereits beim Anlegen des Web-Projektes generiert wurde (vgl. Listing 26).

Listing 26: Einfaches JSP-Beispiel

```
 1 <%@page contentType="text/html"%>
 2 <%@page pageEncoding="UTF-8"%>
 3 <html>
 4     <head><title>JSP Page</title></head>
 5     <body>
 6         <jsp:useBean id="kunde" scope="session" class="beans.Kunde" />
 7         Kunde <jsp:getProperty name="kunde"  property="name" />
 8         <br>Kunde <%=kunde.getName()%>
 9     </body>
10 </html>
```

Geändert wurden die Zeilen 6 sowie 7 und Zeile 8 wurde hinzugefügt. In Zeile 6 wird eine so genannte **JavaBean** verwendet. Die Minimalanforderungen an eine JavaBean sind (vgl. Balzert 2003, S. 78):

- Sie ist eine Java-Klasse.
- Sie besitzt einen Konstruktor ohne Parameter.
- Sie besitzt Standard-*get*- und –*set*-Methoden für die Attribute.

Diese Anforderungen sind durch die Klasse *Kunde* erfüllt (vgl. Listing 27). Durch die Standardaktion *useBean* wird ein Objekt mit dem Namen *kunde* der Klasse *Kunde* erzeugt. Auf die private-Attribute kann auf zwei Weisen zugegriffen werden. In Zeile 7 erfolgt der Zugriff über die Standardaktion *getProperty*. Hinter Standardaktionen stehen nichts anderes als Java-Klassen, deren Funktionalität über diese tag-Notation mit Hilfe von Parametern (z.B. name bzw. property) verwendet werden kann. Die hier verwendeten Standardaktionen sind in vordefinierten **Standard-Tag-Libraries** hinterlegt, in gleicher Weise könnten jedoch auch eigene Tag-Libraries angelegt werden (vgl. Wille 2001, S. 197 ff.). Daneben kann auch über die definierten get- und set-Methoden zugegriffen werden. Somit kann auf die Attributwerte in Ausdrücken (siehe Zeile 8), Anweisungen und Aufrufen zugegriffen werden. Die JSP wird im *Tomcat* in ein Servlet konvertiert und ausgeführt. Im *NetBeans*-Explorer kann über das Umgebungsmenü des Knotens *index.jsp* mit der Option *View Servlet* der generierte Quellcode angeschaut werden.

Listing 27: JavaBean *Kunde*

```
 1 package beans;
 2 public class Kunde {
 3     private int kundennummer;
 4     private String name;
 5     public Kunde() {
 6         setKundennummer(99);
 7         setName("TestKunde"); }
 8     public int getKundennummer() {
 9         return kundennummer; }
10     public void setKundennummer(int kundennummer) {
11         this.kundennummer = kundennummer; }
12     public String getName() {
13         return name; }
14     public void setName(String name) {
15         this.name = name; }
16 }
```

Beim Browser kommt reiner HTML-Code an (vgl. Listing 28).

Listing 28: Response des einfachen JSP-Beispiels

```
1 <html>
2     <head><title>JSP Page</title></head>
3     <body>
4         Kunde TestKunde
5         <br>Kunde TestKunde
6     </body>
7 </html>
```

6.2 Anwendung von Servlets und JSPs auf ein einfaches Beispiel

Wie bereits angedeutet, soll aufbauend auf den Fachklassen und den Klassen zur Datenbankanbindung unseres Bank-Beispiels eine äußerst vereinfachte Online-Banking-Anwendung entwickelt werden. Von den bisher realisierten Transaktionen bieten sich aufgrund sachlicher Überlegungen die Überweisung und die Kontostandsübersicht an. Wie im obigen Beispiel müssen wir in *NetBeans* als erstes ein

Web-Projekt einrichten. Als Projektnamen vergeben wir *JavaTutorialWeb*. Unter Source Packages legen wir einen Ordner *bankbeispiel* an, in den wir die Klassen des Pakets *bankMitGUIUndDB* im *NetBeans* Explorer kopieren. Ebenso kopieren wir die Pakete *database* und *helper* unter *Source Packages*. Anschließend kann das Projekt *JavaTutorial* geschlossen werden.

Da sich der Web-Benutzer authentifizieren sollte, ergibt sich eine notwendige inhaltliche Anpassung. Bei der Klasse *Kunde* fügen wir das Attribut Passwort ein. Dies erfordert auch entsprechende Anpassungen an der Datenbanktabelle *Kunde* und den Zugriffsmethoden sowie dem Dialog zum Anlegen eines neuen Kunden. Ansonsten müssen wir uns nur mit der Benutzersicht beschäftigen. Hierzu setzen wir JSPs, Servlets und JavaBeans ein. Abb. 13 verdeutlicht das Zusammenspiel der einzelnen Komponenten bei der Abarbeitung eines Requests. Dabei übernimmt das Controller-Servlet die Rolle des **Input-Controllers** in Form des **Front-Controllers** (vgl. 7.4.1, 7.4.2 und Abb. 7.6) und ist damit auch vergleichbar mit der *CtrlDlgBank*-Klasse in Abschnitt 4. Grundsätzlich für jede View in der Gestalt einer JSP verwenden wir eine JavaBean-Klasse in der Art einer **Form-Bean**, welche ein inhaltliches Abbild des in der JSP verwendeten HTML-Formulars darstellt. Diese Form-Bean ist ein Teil der Dialogschicht und übernimmt insbesondere die Validierung der Eingabedaten. Sofern es sich um rein formale Prüfungen handelt (z.B. Prüfung auf numerischen Inhalt) ist die Logik in der Form-Bean implementiert. Handelt es sich um fachinhaltliche Prüfungen (z.B. Kundennummer existiert, Passwort zulässig) delegiert die Form-Bean diese Aufgaben an die Fachkonzeptschicht. In unserem einfachen Fall geschieht dies über das schon bekannte *Bank-Handler*-Objekt. Weiterhin wird von der Form-Bean auch die fachinhaltliche Verarbeitung im Sinne der notwendigen Transaktion an die Fachkonzeptschicht über das Handler-Objekt delegiert. In Abhängigkeit vom Ergebnis der Validierung bzw. Transaktionsausführung steuert der Front-Controller die Auswahl der relevanten JSP als Antwortseite, welche den darzustellenden Inhalt an den Web-Browser weiterleitet.

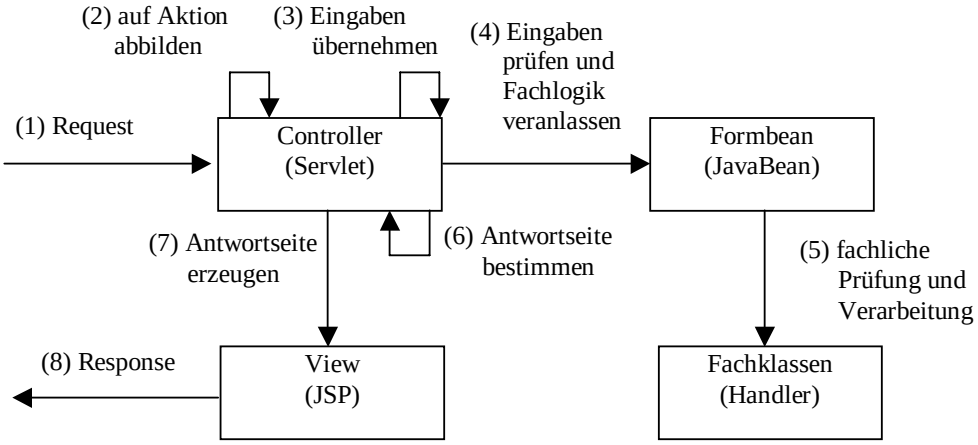

Abb. 13: Grundsätzliche Architektur des Request-Ablaufs der Web-Anwendung

Listing 29a: JSP *startLogin*

```
 1 <%@page contentType="text/html"%>
 2 <%@page pageEncoding="UTF-8"%>
 3 <jsp:useBean id="loginForm" class="formbean
     .LoginForm" scope="session" />
 4 <jsp:useBean id="kunde" class="bankbeispiel.Kunde" scope="session" />
 5 <HTML>
 6 <BODY bgcolor="#FFFFFF" text="#000000">
 7 <H2><FONT face="Arial, Helvetica, sans-serif">Kunden-Login</FONT></H2>
 8 <FORM name="login" method="get" action="Controller">
 9   <input type="hidden" name="dispatchto" value="login">
10   <%
11      if (loginForm.getErrors().size() > 0)
12      {
13   %>
14   <FONT color="#FF0000">Ihre Eingaben sind unvollständig.</FONT><br>
15   <% } %>
16     <TABLE border="0">
17       <TR>
18         <TD>Kundennummer: </TD>
```

Listing 29b: JSP *startLogin*

```
19          <TD>
20            <INPUT type="text" name="kundennummer" size="20"
21                value="<jsp:getProperty name="loginForm"
                    property="ekundennummer" />">
22          </TD>
23          <TD><FONT color="#FF0000"><%=loginForm
              .getError("kundennummer")%> </FONT>
24          </TD>
25        </TR>
26        <TR>
27          <TD>Passwort: </TD>
28          <TD>
29            <INPUT type="password" name="passwort" size="20"
30             value="<jsp:getProperty name="loginForm"
                 property="passwort" />">
31          </TD>
32          <TD><FONT color="#FF0000"><%=loginForm
              .getError("passwort")%> </FONT></TD>
33        </TR>
34      </TABLE>
35      <P> <INPUT type="submit" name="submit" value="Login"></P>
36    <%
37       if (loginForm.isAngemeldet())
38        {
39    %>
40    <br> <br> <b>Kunde <%= kunde.getName() %> ist angemeldet</b>.<br><br>
41    <table border="1" cellpadding="10">
42          <tr>
43    <td> <a href="ueberweisung.jsp?ok=nein"><b>Überweisung</b></a> </td>
44    <td> <a href="kontoUebersicht.jsp"><b>Kontoübersicht</b></a></td>
45         </tr>
46    </table>
47    <% } %>
48 </FORM>
49 </BODY>
50 </HTML>
```

Nach dieser generellen Beschreibung wollen wir ein wenig auf die Details der Transaktionen *Login*, *Überweisung* und *Kontostandsübersicht* näher eingehen. In den Listings 29a und 29b ist der Quellcode der JSP für das Login wiedergegeben. In den Zeilen 3 und 4 werden mit JSP-Aktionen je ein Objekt der Klasse *LoginForm* und *Kunde* erzeugt, sofern nicht schon durch vorausgehende Aktionen ein Objekt im Kontext erzeugt wurde. In Zeile 8 wird im *Form*-Tag als Aktion festgelegt, dass die HTTP-Anfrage an den Controller gerichtet wird. Die *Controller*-Klasse wurde im Paket *servlet* als Servlet angelegt und durch *NetBeans* wurde das Servlet in der *web.xml*-Datei bekannt gegeben weiterhin wurde ein Mapping auf *Controller* durchgeführt. In Zeile 9 wird über das verdeckte *input*-Feld die Aktion festgelegt, welche vom Controller-Servlet ausgewertet wird, um die entsprechende Aktion auszulösen. Die Zeilen 10-15 sind notwendig, da die JSP auch verwendet wird, wenn nach der Validierung der Eingaben die bisher gültigen Eingaben und die entsprechenden Fehlermeldungen angezeigt werden. Die HTML-Ausgabe in Zeile 14 wird nur dann ausgegeben, wenn das *HashMap*-Objekt *errors* des *Login*-Objektes mindestens eine Fehlermeldung enthält. Die Steuerung erfolgt über Java-Code der als Scriptlet in die JSP-Seite eingefügt ist. Die linke Hälfte der Abb. 14 zeigt das Ergebnis, wenn z.B. ein ungültiges Passwort eingegeben wurde. Die Führungstexte und Eingabefelder werden in einer HTML-Tabelle angeordnet. In den Zeilen 21 bzw. 30 werden in den Textfeldern *kundennummer* bzw. *passwort* die Werte angezeigt, welche in der JavaBean *loginForm* in den entsprechenden Feldern (im JavaBean Sinne nennt man diese Felder properties) hinterlegt sind. Wird die JSP in einer HTTP-Session das erste Mal durchlaufen, sind die Werte entsprechend der Vorbelegung durch den Standard-Konstruktor der Klasse *LoginForm*. Bei weiteren Aufrufen werden die vormals eingegebenen Werte angezeigt. In den Zeilen 23 und 32 werden Fehlermeldungen aus dem *HashTable*-Objekt *errors* der *loginFrom*-JavaBean angezeigt. Als Schlüssel-Objekt wird ein *String*-Objekt verwendet, dessen Inhalt dem Namen des Eingabefeldes entspricht. Diese Konvention erlaubt eine leichtere Lesbarkeit und erhöht die Verständlichkeit. In den Zeilen 37-47 wird eine Ausgabe erzeugt, falls sich der Kunde erfolgreich angemeldet hat. In Zeile 40 wird der Name des angemeldeten Kunden ausgegeben und anschließend zwei Links auf die JSPs *ueberweisen.jsp* bzw. *kontoUebersicht.jsp*.

Abb. 14: Anzeigen der JSP loginStart

Listing 30a: JavaBean *LoginForm* (um Platz zu sparen wurden die Standard-*get*- und –*set*-Methoden nicht ausgegeben)

```
 1 package formbean;
 2 import bankbeispiel.*;
 3 public class LoginForm{
 4    private String ekundennummer;
 5    private int kundennummer;
 6    private String passwort;
 7    private java.util.Hashtable  errors=new java.util.Hashtable () ;
 8    private BankHandler bankHandler;
 9    private boolean angemeldet = false;
10    public LoginForm()  {
11        ekundennummer="";
12        passwort="";}
13    public Kunde validate (BankHandler bankHandler){
14       boolean ok=true;
15       errors=new java.util.Hashtable();
16       Kunde kunde=null;
17       try{
18           kundennummer=Integer.parseInt(getEkundennummer());
```

Listing 30b: JavaBean *LoginForm* (um Platz zu sparen wurden die Standard-*get*- und –*set*-Methoden nicht ausgegeben)

```
19          if (getKundennummer()==0 ){
20              errors.put ("kundennummer", "Bitte
                  geben Sie die Kundennummer an");
21              ok=false; }
22          else{
23              try{kunde=bankHandler.getKunde(getKundennummer()); }
24              catch(Exception e){
25                  errors.put("kundennummer",e.getMessage());
26                  ok=false; }
27          }
28          if (getPasswort().equals ("")){
29            errors.put ("passwort","Bitte geben Sie Ihr Passwort ein");
30            ok=false;}
31          else{
32              if(ok){
33                  try{bankHandler.validatePW(getPasswort(), kunde);}
34                  catch(Exception e){
35                      ok=false;
36                      errors.put("passwort",e.getMessage());}
37              }
38          }
39          if(ok){setAngemeldet(true);}
40      }
41      catch(NumberFormatException e){
42          ok=false;
43          errors.put("kundennummer", "Bitte Zahlenwert eingeben");}
44      return kunde;    }
45   public String getError (String key) {
46       String msg = (String)getErrors().get(key);
47       return msg != null ? "- " + msg : ""; }
48    public int getKundennummer() {
49        return kundennummer; }
50    public void setKundennummer(int kundennummer) {
51        this.kundennummer = kundennummer;} }
```

In den Listings 30a und 30b ist der wesentliche Quellcode der JavaBean *LoginForm* wiedergegeben. Neben den Standard-*get*- und –*set*-Methoden, die nur für das Attribut Kundennummer abgedruckt sind und der Methode *getError()* ist die Methode *validate()* der Hauptbestandteil dieser Klasse. Auffällig ist vielleicht, dass es neben dem Attribut *kundennummer* das Attribut *ekundennummer* gibt. Dies hat seinen Grund darin, dass im HTML-Formular der JSP nur Zeichen und keine Zahlen eingegeben bzw. angezeigt werden können. Somit wird der eingegebene Wert erst mal in dem *Controller*-Objekt in das Feld *ekundennummer* eingelesen und in der *validate()*-Methode in Zeile 18 auf numerischen Inhalt geprüft. Werden bei den Prüfungen Fehler festgestellt, werden in die *HashTable errors* entsprechende Einträge vorgenommen, die wie oben dargestellt, von der JSP ausgegeben werden. Handelt es sich um fachliche Prüfungen, z.B. ob überhaupt eine Kunde mit der angegebenen Kundennummer existiert, so wird diese Prüfung durch die Fachklassen vorgenommen, welche über das *BankHandler*-Objekt angesprochen werden (vgl. Zeilen 23 und 33 in Listing 30b). In Zeile 39 wird das Attribut *angemeldet* auf *true* gesetzt, damit wird die Anzeige in der *startLogin*-JSP (vgl. Zeile 37 im Listing 29b) gesteuert. Die Methode *validate()* wird von der Methode *login* der Servlet-Klasse *Controller* verwendet (vgl. Listings 31a und b). Im Servlet *Controller* wird entweder die *doGet()*- oder *doPost()*-Methode ausgeführt. Beide verwenden die Methode *processRequest()*, die bei jeder Anfrage ausgeführt wird. Beim erstmaligen Aufruf im Rahmen einer Session wird ein *BankHandler*-Objekt erstellt und im *Session*-Objekt als Attribut abgelegt (vgl. Zeilen 25-33). Anschließend wird der Request-Parameter *dispatchto* ausgewertet. Ist der Inhalt gleich *login*, so wird die Methode *login()* ausgeführt (vgl. Zeilen 36-37). Die Methode *login()* bildet eine Referenz auf das *LoginForm*-Objekt, das als JavaBean in der JSP *startLogin* über die *useBean*-Aktion (Zeile 3 in Listing 29a) erzeugt wurde (vgl. Zeile 3 in Listing 32a). Anschließend werden die Eingabewerte des HTML-Formular, welche im Request als Parameter übertragen wurden in die korrespondierenden Attribute der Form-Bean übertragen. Die Validierung der Eingabewerte wird in Zeile 6 ausgelöst. Hat die Validierung keine Fehler erbracht, wird das Objekt *kunde* im aktuellen Session-Objekt abgelegt. Sowohl in diesem Fall, als auch im Fehlerfall wird die *startLogin*-JSP über die *dispatchPage()*-Methode (vgl. Zeilen 41-43 in Listing 31b) angezeigt. War die Kunden-Anmeldung erfolgreich, hat der Benutzer die Möglichkeiten Überweisungen auszuführen bzw. die Kontoübersicht anzeigen zu lassen (vgl. Zeilen 43 und 44 in Listing 29b).

Listing 31a: Controller-Servlet (ausschnittsweise)

```
 1 package servlet;
 2 import java.io.*;
 3 import javax.servlet.*;
 4 import javax.servlet.http.*;
 5 import bankbeispiel.*;
 6 import database.*;
 7 import formbean.*;
 8 public class Controller extends HttpServlet {
 9     private ServletContext application;
10     private HttpSession session;
11     private HttpServletRequest req;
12     private HttpServletResponse res;
13     private RequestDispatcher dispatcher;
14     private BankHandler bankHandler;
15     private String page;
16     public void init(ServletConfig config) throws ServletException {
17         super.init(config); }
18     public void destroy() {}
19     protected void processRequest(
         HttpServletRequest request,
         HttpServletResponse response)
20     throws ServletException, IOException {
21         this.session=request.getSession();
22         this.application=getServletContext();
23         this.req=request;
24         this.res=response;
25         this.bankHandler=(BankHandler)
             session.getAttribute(„bankHandler“);
26         if(bankHandler==null){
27            try{
28                bankHandler=new BankHandler(new DbConnection());
29                session.setAttribute(„bankHandler“,bankHandler);}
30            catch(Exception e){
```

Listing 31b: Controller-Servlet (ausschnittsweise)

```
31                System.out.println(
                    „Fehler beim bankHandler ini-
                     tialisieren: „+e.getMessage());
32                e.printStackTrace();}
33          }
34          String dispatchto=req.getParameter("dispatchto");
35          String page="";
36          if(dispatchto.equals("login")){
37              login();  }
38          if(dispatchto.equals("ueberweisen")){
39              ueberweisen(); }
40       }
41      private void dispatchPage(String page) throws Exception{
42          dispatcher=application.getRequestDispatcher(page);
43          dispatcher.forward(req,res); }
44      protected void doGet(HttpServletRequest
          request, HttpServletResponse response)
45          throws ServletException, IOException {
46          processRequest(request, response); }
47      protected void doPost(HttpServletRequest
          request, HttpServletResponse response)
48            throws ServletException, IOException {
49            processRequest(request, response); }
```

Listing 32a: *login()*-Methode der Servlet-Klasse *Controller*

```
1 private void login(){
2      try{
3          LoginForm loginForm=(LoginForm)session
             .getAttribute(„loginForm“);
4          loginForm.setEkundennummer(req.getParameter(„kundennummer“));
5          loginForm.setPasswort(req.getParameter(„passwort“));
6          Kunde kunde=loginForm.validate(bankHandler);
7          if (loginForm.getErrors().size()!=0){
8              page=“/startLogin.jsp“; }
9           else {
```

Listing 32b: *login()*-Methode der Servlet-Klasse *Controller*

```
10              session.setAttribute(„kunde“, kunde);
11              page=“/startLogin.jsp“;}
12              dispatchPage(page);
13              return;
14      }
15      catch(Exception e) {
16          System.out.println(
              "Fehler beim Dispatch login
              /optionenAuswahl,
               code: "+e.getMessage());
17             e.printStackTrace();}
18      }
```

Listing 33a: JSP *ueberweisung*

```
 1 <%@page contentType="text/html"%>
 2 <%@page pageEncoding="UTF-8"%>
 3 <jsp:useBean id="ueberweisungsForm" class
     ="formbean.UeberweisungsForm" scope="session" />
 4 <HTML>
 5 <script language="Javascript" src="JavaScript\calendar.js"></script>
 6 <BODY bgcolor="#FFFFFF" text="#000000">
 7 <H2><FONT face="Arial, Helvetica, sans-serif">
     Überweisung durchführen</FONT></H2>
 8 <FORM name="ueberweisung" method="get" action="Controller">
 9 <input type="hidden" name="dispatchto" value="ueberweisen">
10 <%  if (ueberweisungsForm.getErrors().size() > 0)  { %>
11 <p><FONT color="#FF0000">Ihre Eingaben sind
     unvollständig/fehlerhaft.</FONT></p>
12 <p><FONT color="#FF0000"><%=ueberweisungsForm
     .getError("fehlerBeiUeberweisung")%> </FONT></P>
13 <p>
14 <%  } %>
15   <TABLE border="0">
16     <TR>  <TD>von Konto </TD>
17       <TD> <INPUT type="text" name="vonKontoNr" size="20"
```

Listing 33b: JSP *ueberweisung*

```
18              value="<jsp:getProperty name=
                  "ueberweisungsForm" property=
                  "evonKonto" />">
19        </TD>
20        <TD><FONT color="#FF0000"><%=ueberweisungsForm
            .getError("vonKontoNr")%> </FONT></TD>
21      </TR>
22      <TR>  <TD>nach Konto </TD>
23        <TD>  <INPUT type="text" name="nachKontoNr" size="20"
24              value="<jsp:getProperty name=
                  "ueberweisungsForm" property=
                  "enachKonto" />">
25        </TD>
26        <TD><FONT color="#FF0000"><%=ueberweisungsForm
            .getError("nachKontoNr")%> </FONT></TD>
27      </TR>
28      <TR> <TD>Datum </TD>
29        <TD>  <INPUT type="text" name="datum" size="20"
30              value="<jsp:getProperty name=
           "ueberweisungsForm" property="edatum" />">
31              <a href="#" onclick="return getCalendar
                 (document.ueberweisung.datum);">
32              <img src="JavaScript\calendar.png" border="0" /></a>
33        </TD>
34        <TD><FONT color="#FF0000"><%=ueberweisungsForm
            .getError("datum")%> </FONT></TD>
35      </TR>
36      <TR>  <TD>Betrag </TD>
37        <TD>  <INPUT type="text" name="betrag" size="20"
38            value="<jsp:getProperty name=
       "ueberweisungsForm" property="ebetrag" />">
39        </TD>
40        <TD><FONT color="#FF0000"><%=ueberweisungsForm
            .getError("betrag")%> </FONT> </TD> </TR>
41    </TABLE>
42 <P><INPUT type="submit" name="ueberweisen" value="Überweisen"> </P>
43 </FORM>
```

Listing 33c: JSP *ueberweisung*

```
44 <%  if (request.getParameter("ok").equals("ja"))  { %>
45 <p>Überweisung von <%=ueberweisungsForm.getBetrag()%>
     von <%=ueberweisungsForm.getVonKontoNr()%>
46 nach <%=ueberweisungsForm.getNachKontoNr()%> ausgeführt. </p>
47 <table border="1" cellpadding="10">
48   <tr><td> <a href="kontoUebersicht.jsp"><b>Kontoübersicht</b></a></td>
49     <td> <a href="loggedOut.jsp"><b>Logout</b></a> </td></tr>
50 </table>
51 <%   } %>
52 </BODY> </HTML>
```

Die JSP *ueberweisung* ist in kompakter Darstellung in den Listings 33a, 33b und 33c wiedergegeben. Im Prinzip ist der Aufbau dem der JSP *startLogin* sehr ähnlich. Eine kleine Besonderheit ist bei der Eingabe des Datums vgl. Zeilen 30-32 in Listing 33b). Hier wird eine fertige JavaScript-Komponente zur Datumsauswahl verwendet (vgl. http://www.mojavelinux.com/projects/popupcalendar/). Die Dateien sind im Ordner *JavaScript* zusammengefasst. Durch die Verwendung dieses Kalenders reduziert sich der Aufwand für Plausiblitätskontrollen enorm und die Benutzbarkeit wird positiv beeinflusst (vgl. Abb. 15). Zu dieser JSP wurde im Ordner *FormBean* die JavaBean *UeberweisungsForm* entwickelt und im *Controller*-Servlet wurde eine Methode *ueberweisen()* hinzugefügt (vgl. Listing 34), die in Abhängigkeit vom Inhalt des *dispatchto*-Parameters (vgl. Zeile 9 in Listing 33a) von der Methode *processRequest()* ausgelöst wird (vgl. Zeilen 38-39 in Listing 31b). Die Details des Quellcodes sind weitgehend selbsterklärend. Fachlich musste eine Ergänzung durchgeführt werden. Es muss ja sichergestellt sein, dass der angemeldete Bankkunde nur Überweisungsaufträge von seinen Konten durchführen kann. Diese Einschränkung war bei der bisherigen Überweisungs-Transaktion nicht notwendig, da ja der Bankmitarbeiter Überweisungsaufträge aller Kunden im System durchführen darf. Dies stellt die Anweisung in Zeile 32 (vgl. Listing 35) sicher. Dazu war es notwendig in der *BankHandler*-Klasse eine neue Methode *pruefenKontoKunde()* einzufügen, die allerdings auf eine bereits vorhandene Methode *getKonto()* in der Klasse *Kunde* zurückgreifen konnte. Dieses Beispiel macht eindrucksvoll deutlich, dass aufgrund der strikten Trennung der Dialogschicht von den anderen Schichten außer notwendigen Ergänzungen keine Änderungen an den Fachklassen bzw. Datenbankklassen durchzuführen waren.

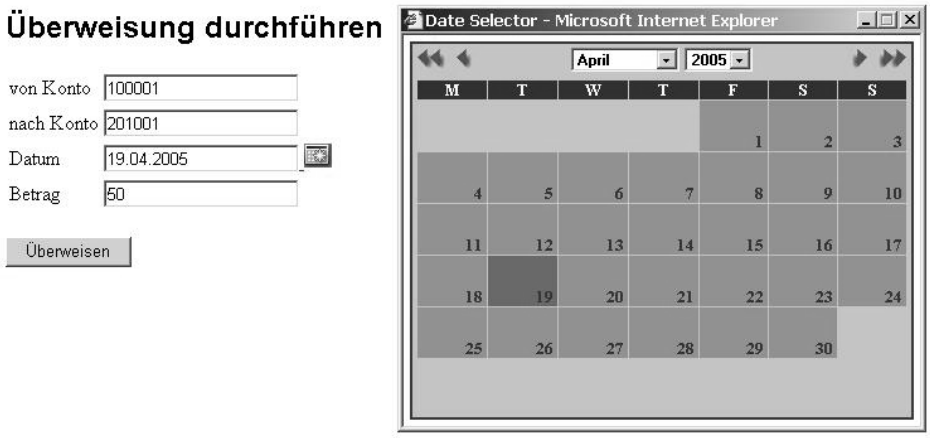

Abb. 15: Dialog der JSP *ueberweisung*

Listing 34: Methode *ueberweisen()* im *Controller*-Servlet

```
1  private void ueberweisen(){
2          UeberweisungsForm uebForm=(UeberweisungsForm)
             session.getAttribute("ueberweisungsForm");
3          uebForm.setEvonKonto(req.getParameter("vonKontoNr"));
4          uebForm.setEnachKonto(req.getParameter("nachKontoNr"));
5          uebForm.setEdatum(req.getParameter("datum"));
6          uebForm.setEbetrag(req.getParameter("betrag"));
7          uebForm.validate(bankHandler,
8          if(uebForm.getErrors().size()!=0){
9              page="/ueberweisung.jsp?ok=nein";
10         }
11         else{
12             page="/ueberweisung.jsp?ok=ja";
13         }
14         try{
15             dispatchPage(page); }
16         catch(Exception e){
17             System.out.println("Fehler beim Dispatch login
                 /optionenAuswahl, code: "+e.getMessage());
18             e.printStackTrace(); } }
```

Die Transaktion Kontoübersicht stößt ja keine Verarbeitung an, sondern stellt nur Informationen des angemeldeten Kunden dar (vgl. Abb. 16). Da insbesondere alle drei *Konto*-Klassen verwendet werden, wird das Paket *bankbeispiel* importiert (vgl. Zeile 3 in Listing 36). Dies erfolgt über die so genannte JSP-Direktive *page*. Mit dieser Direktive können Attribute definiert werden, die für eine JSP-Seite gelten und vom JSP-Compiler verwendet werden. In unserem Fall bewirkt die Anweisung in Zeile 3 das Gleiche wie eine *import*-Anweisung in einem Java-Programm. In den Zeilen 11-19 werden die *Konto*-Daten des Kunden in einer Schleife in Form einer HTML-Tabelle ausgegeben. In Zeile 23 ist ein Link auf eine *Logout*-Seite. In dieser JSP wird die laufende Session beendet und die Möglichkeit für eine Neuanmeldung gegeben.

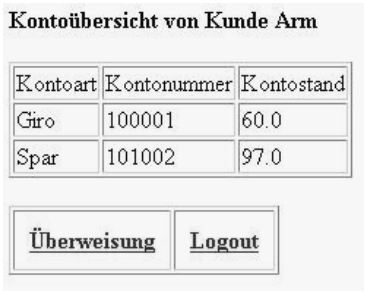

Abb. 16: Dialog der JSP *kontoUebersicht*

Listing 35a: Auszug aus Klasse *UeberweisungsForm*

```
 1 package formbean;
 2 import bankbeispiel.*;
 3 public class UeberweisungsForm{
 4    private String evonKonto;
 5    private String enachKonto;
 6    private String edatum;
 7    private String  ebetrag;
 8    private int vonKontoNr;
 9    private int nachKontoNr;
10    private java.util.Date datum;
11    private double betrag;
12    private java.util.Hashtable  errors=new java.util.Hashtable () ;
13    public UeberweisungsForm()  {
```

```
14        evonKonto="";
15        enachKonto="";
16        edatum="";
17        ebetrag="";}
18    public void validate (BankHandler bankHandler, Kunde kunde){
19       int error=0;
20       setErrors(new java.util.Hashtable());
21       try{
22           error++;
23           setVonKontoNr(Integer.parseInt(getEvonKonto()));
24           error++;
25           setNachKontoNr(Integer.parseInt(getEnachKonto()));
26           error++;
27           betrag=Double.parseDouble(getEbetrag());
28           error++;
29           java.text.DateFormat df= new java.text
               .SimpleDateFormat("dd.MM.yyyy");
30           datum=df.parse(getEdatum());
31           error++;
```

Listing 35b: Auszug aus Klasse *UeberweisungsForm*

```
32           bankHandler.pruefenKontoKunde(getVonKontoNr(), kunde);
33           bankHandler.ueberweisen(getVonKontoNr(),
               getNachKontoNr(), betrag, datum);}
34        catch(Exception e){
35            switch(error){
36                case 1:
37                    System.out.println("von "+getEvonKonto());
38                    setVonKontoNr(0);
39                    getErrors().put("vonKontoNr",
                        "Bitte Eingabe einer Zahl.");
40                    break;
41                case 2:
42                    setNachKontoNr(0);
43                    getErrors().put("nachKontoNr",
                        "Bitte Eingabe einer Zahl.");
```

```
44                  break;
45              case 3:
46                  setBetrag(0);
47                  getErrors().put("betrag",
                      "Bitte einen Betrag eingeben.");
48                  break;
49              case 4:
50                  getErrors().put("datum", "Ungültiges Datum");
51                  break;
52              case 5:
53                  getErrors().put("fehlerBeiUeberweisung",
                      e.getMessage());
54                  break;}
```

Listing 36: JSP *kontoUebersicht*

```
1  <%@page contentType="text/html"%>
2  <%@page pageEncoding="UTF-8"%>
3  <%@page import="bankbeispiel.*"%>
4  <jsp:useBean id="kunde" class="bankbeispiel.Kunde" scope="session" />
5  <html>
6      <head><title>Konto-Übersicht</title></head>
7      <body>
8      <p> <b>Kontoübersicht von Kunde <%=kunde.getName()%></b> </p>
9       <TABLE border="1">
10         <TR> <td>Kontoart</td> <td>Kontonummer</td>
             <td>Kontostand</td> </tr>
11         <% java.util.Iterator it=kunde.getKontenliste().iterator();
12            while(it.hasNext()){
13              Konto konto=(Konto)it.next();
14              String kontoart=(konto instanceof
                  Girokonto) ? "Giro" : "Spar";
15              String kontonummer=String.valueOf(konto.getKontonummer());
16              String kontostand=String.valueOf(konto.getKontostand());
17         %>
18         <tr> <td><%=kontoart%></td>
             <td><%=kontonummer%></td> <td><%=kontostand%></td> </tr>
19         <%  }  %>
```

```
20          </table> <br>
21              <table border="1" cellpadding="10">
22                  <tr> <td> <a href="ueberweisung.jsp?ok=nein">
                      <b>Überweisung</b></a> </td>
23                              <td> <a href="loggedOut.jsp">
                                  <b>Logout</b></a></td>  </tr>
24              </table>
25      </body>
26 </html>
```

7 Zusammenfassung

In diesem Kapitel haben wir anhand eines einfachen Fallbeispiels **Java-Technologien** zur Programmierung kennen gelernt und angewandt. Abb. 17 gibt die einzelnen Schritte wieder. Ausgehend von einem einfachen Fachklassen-Modell für eine einfache Bank haben wir mit der Programmiersprache Java in der Version 5.0 und der **integrierten Entwicklungsumgebung** *NetBeans* Grundkonzepte der Programmiersprache zur Umsetzung dieser Fachklassen angewandt. Bevor wir eine grafische Dialogschnittstelle mit dem **Swing-Framework** entwickelten, haben wir die Funktionsweise der Fachklassen getestet. Mit Swing wurden einige Anwendungsfälle umgesetzt. Dabei haben wir zusätzliche Sprachelemente kennengelernt und verwendet. Insbesondere wurden wir auch mit der Anwendung des klassischen Model View Controller-Musters anhand der JTable-Klasse vertraut. Als nächsten Schritt haben wir das **relationale Datenbankmanagementsystem** *MySQL* dafür verwendet, um unsere Objekte persistent zu machen. Dabei haben wir das Zusammenspiel von Java, JDBC und Datenbanktreibern kennengelernt sowie die Entwurfskonzepte aus dem siebten Kapitel implizit angewandt, ohne, dass wir im Detail den Entwurf modelliert hätten. Entsprechende Verweise machen jedoch die Verwendung deutlich. Im letzten Abschnitt haben wir für unser Beispiel noch zwei Anwendungsfälle als eine **Web-Anwendung** umgesetzt. Dabei haben wir die Java-Technologien **Java Server Page** (JSP) und **Servlet** verwendet und die Servlet-Umgebung ***Tomcat*** eingesetzt. Insgesamt haben wir damit ein breites Spektrum an Programmiergrundlagen repetiert und praktisch angewandt.

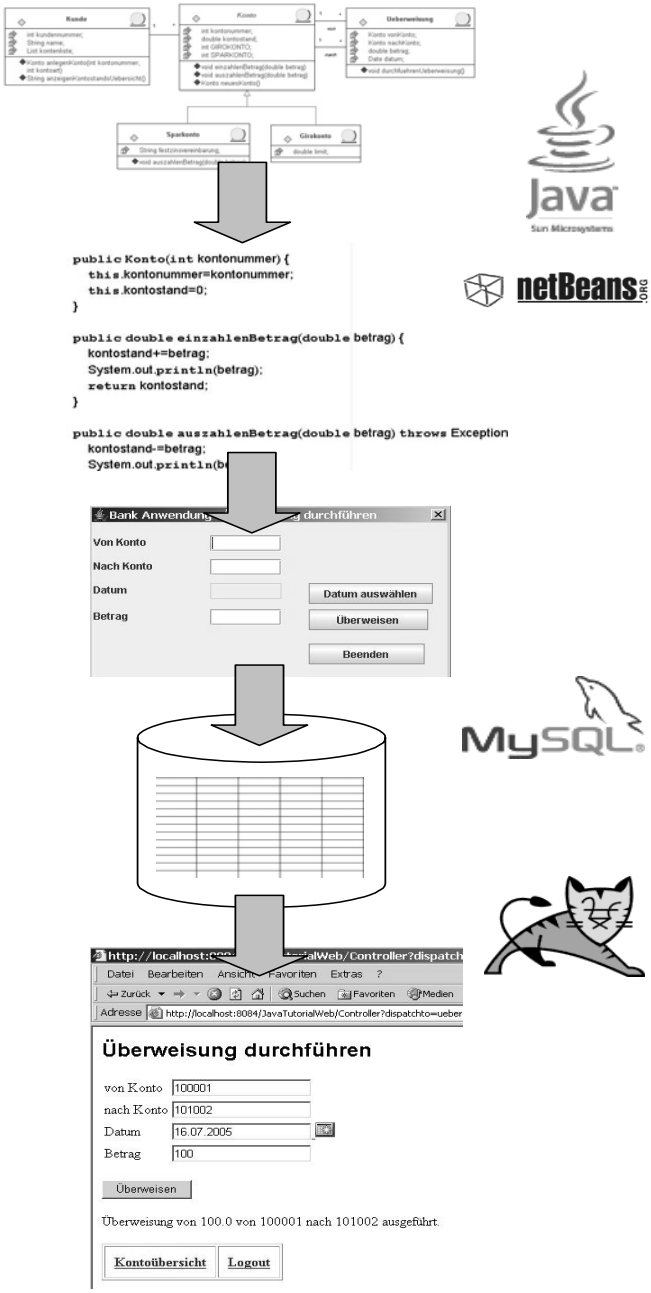

Abb. 17: Inhalt und Struktur des Java Tutoriums