Programmdokumentation

RECHNUNGSBEARBEITUNG

INHALTSVERZEICHNIS:

Bearbeiter:	Sascha Nemecek
Matrikelnummer:	9825815
Übung:	Einführung in das Programmieren – Syntaxpraktikum
Übungsleiter:	Seyfang
Tutor:	Fischmeister
Abgabedatum:	21.01.1999

1 Modularisierung und Datenstruktur

1.1 Grundidee

Das Prinzip ist bei diesem Programm das gleiche wie beim ersten Programm. Nur werden die Daten nicht am Systemoutput ausgegeben sondern in Listen gespeichert. Sobald alle Daten korrekt eingelesen sind, soll ein Frame geöffnet werden. In diesem Frame ist ein Scrollpane enthalten, der wiederum einen Canvas enthält. Die Größe des Canvas muss erst mit Hilfe von *FontMetrics* berechnet werden. In diesem Canvas soll dann das jeweils ausgewählte Diagramm (Balken/Kreis & Artikel/Kassa) dargestellt werden.
Zusätzlich zu den Datenbearbeitungsmöglichkeiten „*Rechnung löschen*“, „*Rechnung bearbeiten*“ und „*Rechnung erstellen*“ soll auch noch eine *Clone*-Funktion realisiert werden, die einen neuen Frame initialisiert.

1.2 Modularisierung

Das Programm kann in 3 Hauptteile zerlegt werden:

- die Daten einlesen,
- die Daten für die Ausgabe berechnen,
- die Daten in einem Diagramm ausgeben.

1.2.1 Das Einlesen der Daten

erfolgt in drei Schritten. Es werden zuerst die Artikeldaten, dann die Rechnungsdaten und zum Schluss die Gruppendaten eingelesen. Alle drei werden in jeweils einer doppelt verketteten Liste, die alle in der Klasse Daten enthalten sind abgelegt. Dadurch ist es möglich über eine Referenz die gesamten Daten anzusprechen.
Die dafür verwendete Klasse wird **Daten.java** genannt.

1.2.2 Die Ausgabedaten

Da für beide Diagrammarten (Kreis- & Balkendiagramm) jeweils nur zwei Daten, die Beschriftung und die Anzahl der Artikel, benötigt werden, wurde eine einheitliche Schnittstelle zwischen den Daten und der Ausgabe definiert.
Diese funktioniert so, dass eine abstrakte Klasse *Diagrammdaten* definiert wurde, die die Methoden *getnames()* und *getvalues()* garantiert. Von dieser werden 3 weiter Klassen - *Kassa*, *Gruppe* und *Artikel* – abgeleitet (Abbildung 1). Diese filtern die jeweils benötigten Daten für die Ausgabearten, „*verkaufte Artikel*“, „*verkaufte Artikelgruppen*“ und „*verkaufte Artikel pro Kassa*“ aus den Datenstrukturen.

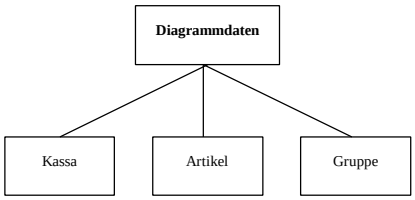

Abbildung 1: Schema der Diagrammdaten

1.2.3 Die Diagrammausgabe

Die Ausgabe erfolgt jeweils in einem Canvas über eine Paintfunktion. Dafür wurde wieder eine abstrakte Klasse *Diagramm* definiert. Von dieser wurden dann die zwei Klassen *Kreisdiagramm* und *Balkendiagramm* abgeleitet (Abbildung 2).

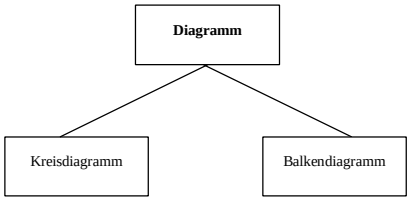

Abbildung 2: Schema der Diagrammausgabe

1.3 Wahl der Klassen

1.3.1 Klasse Rechnungsbearbeitung

Meine Hauptklasse. Zuerst werden alle Daten mit Hilfe der Klasse *Daten* eingelesen. Dann wird die graphische Ausgabe über die Klasse *MyGUI* gestartet. *MyGUI* wird dann noch als Observer der *Daten* bestimmt.

1.3.2 Klasse Daten

Ließt die Artikel-, Rechnungs- und Gruppendaten ein und legt sie in Listen ab.

1.3.3 Klasse Diagramm

Gibt das jeweils verlangte Diagramm (Kreis/Balken) in dem übergebenen graphischen Objekt aus. In unserem Fall ist dies immer der Canvas.

1.3.4 Klasse Diagrammdaten

Filtert die jeweils benötigten Daten für die Diagrammausgabe aus der Datenklasse. Dies geschieht über die Methoden *getnames()* und *getvalues()*, die eine Liste mit den verlangten Daten zurückliefert.

1.3.5 Klasse Dialoge

Abstrakte Klasse Dialoge. Um alle verwendeten Dialoge zu vereinheitlichen.

1.3.5.1 Klasse DialogBearbeiten

Ermöglicht das Bearbeiten der Rechnungsdaten in einem Dialogfenster. Folgende Veränderungen einer Rechnung können vorgenommen werden:

- Artikeleintrag in Rechnung löschen
- Artikeleintrag einer Rechnung bearbeiten
  Dies geschieht durch den Aufruf des Dialoges *DialogEditElement.*
- Neuen Artikeleintrag in einer Rechnung erstellen
  Dafür wird der Dialog DialogNeuesElement aufgerufen.

1.3.5.2 Klasse DialogEditElement

Ermöglicht das Bearbeiten eines Artikeleintrages einer Rechnung in einem Dialogfenster. Dabei kann aber nur die Menge des gewählten Artikels verändert werden. Der Dialog wird als modal definiert, um zu verhindern, dass zu viele Dialoge gleichzeitig geöffnet werden.

1.3.5.3 DialogLöschen

Zeigt alle Rechnungen in einer Liste an. Der Benutzer kann nun einen Eintrag auswählen, und durch das betätigen des Buttons „*Löschen*“ die ausgewählte Rechnung löschen.

1.3.5.4 DialogNeu

Erzeugt eine neue Rechnung, wobei darauf geachtet wird, dass keine alte Rechnung überschrieben wird. Es wird eine neue, leere Rechnung erstellt, die über die Dialog bearbeiten gefüllt werden kann.

1.3.5.5 DialogNeuesElement

Mit diesem Dialog kann ein neuer Artikel in die Rechnung eingefügt werden. Über ein *Choice* können die noch nicht in der Rechnung vorkommenden Artikel ausgewählt werden. Über ein Textfield muss dann nur noch die Menge bestimmt werden.

1.3.6 Liste

In dieser Klasse werden die Daten abgelegt. Es handelt sich dabei um eine doppelt verkettete Liste.

1.3.7 Menue

Erzeugt ein Auswahlmenü im Frame. Durch dieses Menü können Rechnungen erstellt, gelöscht und bearbeitet werden. Weiters kann der Frame geklont werden oder das Programm beendet werden.

1.3.8 MyColors

Da bei der Diagrammausgabe verschieden Farben verwendet werden sollen, kann mit dieser Klasse eine gewisse Farbpalette immer wieder durchlaufen werden.

1.3.9 MyException

Wird verwendet, um meine auftretenden Fehler weiterzuleiten

1.3.10 MyGUI

Die graphische Ausgabe und der Aufbau des Frames wird in dieser Klasse geregelt.

2 Pseudocode

Variablen definieren und initialisieren
File mit Artikeln öffnen
Solange nicht Fileende:
    Zeile einlesen
    Zeile in Substrings aufteilen (Tokenize) und Anzahl der Substrings überprüfen (muss 3 sein, sonst Fehlermeldung)
    Substrings in jeweiliges Format umwandeln (Artikelnummer: **Int**, Beschreibung: **String**, Preis: **double**)
    Wenn keine Fehlermeldung Artikeldaten in Artikelliste speichern

Files mit Rechnungen öffnen:
Solange nicht Fileende:
    Zeile einlesen
    Zeile in Substrings aufteilen (Tokenize) und Anzahl der Substrings überprüfen (muss 2 sein, sonst Fehlermeldung)
    Substrings in jeweiliges Format umwandeln (Artikelnummer: **Int**, Menge: **Int**)
    Wenn keine Fehlermeldung Rechnungsdaten in Rechnungsliste speichern

File mit Gruppen öffnen
Solange nicht Fileende:
    Zeile einlesen
    Zeile in Substrings aufteilen (Tokenize) und Anzahl der Substrings überprüfen (muss 2 sein, sonst Fehlermeldung)
    Substrings in jeweiliges Format umwandeln (Artikelgruppe: **Int**, Beschreibung: **String**)
    Wenn keine Fehlermeldung Gruppendaten in Gruppenliste speichern

Rechnungsdaten durchgehen
    Artikeldaten zur Rechnung suchen
    Rechnungsergebnis berechnen und Ausgabepuffer sicher

Frame initialisieren und die max. Canvasgröße berechnen
Ausgabepuffer durchgehen und jeden Datensatz am Graphikoutput ausgeben

3 Source Code

3.1 Rechnungsbearbeitung.java

```
import java.awt.*;
import java.awt.event.*;

/**
 * class Rechnungsbearbeitung.
 * Hier werden die Daten initialisiert und die graphische Ausgabe gestartet.
 * @author Sascha Nemecek/9825915.
 * @version 2.0 15.01.1999.
 */
public class Rechnungsbearbeitung {

    /**
     * Datenobjekt. Beinhaltet alle Daten.
     */
    private Daten daten;

    /**
     * Gibt den Pfad an, in dem die Datenfiles abgelegt sind.
     */
    final static String DATENPFAD = "f:\\daten\\uni\\1.semester\\eprog\\abgabe4\\Files\\";

    /**
     * Gibt den Names des Files mit den Artikeldaten an.
     * Dieses muss im DATENPFAD abgelegt sein.
     */
    final static String ARTIKELFILE = "artikel.txt";

    /**
     * Gibt den Names des Files mit den Gruppendefinitionen an.
     * Dieses muss im DATENPFAD abgelegt sein.
     */
    final static String GRUPPENFILE = "gruppen.txt";

    /**
     * Versucht die Daten einzulesen. Falls dies nicht mögliche ist, wird das Programm mit
     * einer Fehlermeldung abgebrochen.
     * Falls die Daten erfolgreich eingelesen wurden, wird die graphische Ausgabe gestartet.
     */
    Rechnungsbearbeitung () {
          try {
              // Einlesen der Daten
              daten = new Daten (DATENPFAD, ARTIKELFILE, GRUPPENFILE);
              }
          catch (MyException e) {
              // Ausgabe der Fehlermeldung und Programmende
              System.out.println (e.getMessage());
              System.exit (0);
              }

          // Startet graphische Ausgabe
          MyGUI m = new MyGUI (daten);

          // Initialisiert Observer
          daten.addObserver (m);
          }

    /**
     * Initialisiert Rechnungsdaten, Artikeldaten und Frame.
     */
    public static void main (String argv[]) {
          Rechnungsbearbeitung r = new Rechnungsbearbeitung ();
          }
    }
```

3.2 Daten.java

```
import java.util.*;
import java.io.*;


/**
 * class Daten.
 * Hier werden die Daten gespeichert und verwaltet.
 * @author Sascha Nemecek/9825815.
 * @version 1.0 15.01.1999
 */
class Daten extends Observable {
    Liste artikeldaten = new Liste ();
    Liste rechnungsdaten = new Liste ();
    Liste gruppendaten = new Liste ();

    /**
     * Konstruktor: Daten.
     * @param pfad Pfad der Daten.
     * @param artikelfile Name des Files, in dem die Artikeldaten gespeichert sind.
     * @param gruppenfile Name des Files, in dem die Gruppendaten gespeichert sind.
     * @exception MyException Wirft bei Fehler eine Meldung an das aufrufenden Programm.
     * @see MyException
     */
```

```
    public Daten (String pfad, String artikelfile, String gruppenfile) throws MyException {
          try {
              ReadArtikeldaten (pfad, artikelfile);
              ReadRechnungsdaten (pfad);
              ReadGruppendaten (pfad, gruppenfile);

              // Testausgabe der Daten
/*            for (artikeldaten.gotoStart (); !artikeldaten.eol(); artikeldaten.gotoNext ())
                   System.out.println (((Artikel)artikeldaten.getData()).artikel + " " +
((Artikel)artikeldaten.getData()).name + " " + ((Artikel)artikeldaten.getData()).preis);

              for (gruppendaten.gotoStart (); !gruppendaten.eol(); gruppendaten.gotoNext ())
                   System.out.println (((Gruppe)gruppendaten.getData()).artikel + " " +
((Gruppe)gruppendaten.getData()).name);

              for (rechnungsdaten.gotoStart (); !rechnungsdaten.eol(); rechnungsdaten.gotoNext ())
                    for (((Rechnung)rechnungsdaten.getData()).elements.gotoStart ();
!((Rechnung)rechnungsdaten.getData()).elements.eol (); ((Rechnung)rechnungsdaten.getData()).elements.gotoNext ())
                        System.out.println (((Rechnung)rechnungsdaten.getData()).rechnungsNr + " " +
((Rechnung)rechnungsdaten.getData()).kassa + " " + ((RechnungsElement)((Rechnung)rechnungsdaten.getData()).elements.getData
()).artikel + " " + ((RechnungsElement)((Rechnung)rechnungsdaten.getData()).elements.getData ()).menge);
                         */
              }
          catch (MyException e)
              {throw new MyException (e.getMessage ());}

          Observable o = new Observable ();
          }


    /**
     * methode Changed ().
     * Benachrichtigt Observer.
     */
    public void Changed () {
          setChanged ();
          notifyObservers ();
          }


/*********************************************************************************************
    Ab hier werden Artikeldaten behandelt
*********************************************************************************************/


    /**
     *  Methode: ReadArtikeldaten.
     *  Ließt das durch pfad und filename angegeben File ein und Wertet den Inhalt als Artikeldaten aus.
     *  Falls der Inhalt gültig ist, wird er in einer Liste abgelegt.
     *  @param String pfad: Gibt das Verzeichnis des einzulesenden Files an.
     *  @param String filename: Gibt den Filenamen des einzulesenden Files an.
     *  @exception MyException Wirft bei Fehler eine Meldung an aufrufende Methode.
     *  @see StringtoArtikel
     *    @see MyException
     */
   private void ReadArtikeldaten (String pfad, String filename) throws MyException {
          String s = null;
          ReadFile file;

          // Versucht File zu öffnen. Wenn nicht möglich, Abbruch
          file = new ReadFile (pfad, filename);

          // Ließt solange Artikel ein, bis keine mehr vorhanden sind
          while ((s = file.readLine()) != null) {
              // Wenn Leerzeile, nächste Zeile einlesen
              if (s.length () == 0) continue;

              try {
                    // Wandelt einen String ins Artikelformat um
                    StringtoArtikel (s);
                    }
              catch (MyException e) {
                    // Fehlermeldung (ErrorDialog)
                    System.out.println ("Fehler: " + e.getMessage());
                    }
              }

          file.close ();
          if (artikeldaten.isEmpty())
              throw new MyException ("Es wurden keine Daten eingelesen - Programmabbruch");
          }
```

```
    /**
     * Methode: String to Artikel.
     * Versucht den übergebenen String in der Liste "Artikeldaten" als "Artikel" abzulegen.
     * @param s - der umzuwandelnde String.
     * @exception MyException Wirft bei Fehler eine Meldung an aufrufende Methode.
     * @see Daten$Artikel
     */
    private void StringtoArtikel (String s) throws MyException
          {
          int nummer = 0;
          String name = null;
          double preis = 0;

          // Initialisiert Stringteiler und definiert die Trennzeichen
          StringTokenizer T = new StringTokenizer (s, ", ");

          try {
              // Überprüft, ob die Parameteranzahl korrekt ist
              if ((T.countTokens () != 3) || (s.charAt (s.length () - 1) == ','))
                    throw new MyException ("Falsche Syntax: " + s + " - Datensatz ignoriert.");

              // Ließt die Artikelnummer, wenn nicht dreistellig Abbruch
              name = T.nextToken ();
              if (name.length() !=3)
                    throw new MyException ("Die Artikelnummer " + nummer + " ist ungültig. Sie muss 3stellig sein. Datensatz
ignoriert.");
              nummer = Integer.parseInt (name);

              if ((nummer < 100) || (nummer > 399))
                    throw new MyException ("Die Artikelnummer " + nummer +  " ist ungültig. Sie muss zw. 100 und 399 liegen.
Datensatz ignoriert.");

              // Ließt Artikelnamen, Überprüft, ob der Artikelname zulässig ist, und bricht bei Fehler ab
              name = T.nextToken ();
              if (name.length() > 20)
                    throw new MyException ("Artikelbezeichnung zu lang (max. 20 Zeichen). Datensatz ignoriert.");

              // Ließt den Preis
              preis = Double.valueOf(T.nextToken()).doubleValue();
              }
          catch (NumberFormatException e) {
              throw new MyException (e.getMessage());
              }

          // Wenn kein Fehler bei der Umwandlung aufgetreten ist, Datensatz speichern
          artikeldaten.add (new Artikel (nummer, name,preis));
          }


    /**
     * Methode getArtikelName ().
     * Liefert den zur Artikelnummer gehörigen Namen.
     * @param nr die Artikelnummer.
     * @return string mit dem Artikelnamen.
     */
    public String getArtikelName (int nr) {
          for (artikeldaten.gotoStart (); !artikeldaten.eol (); artikeldaten.gotoNext ())
              if (((Artikel) artikeldaten.getData ()).artikel == nr)
                    return ((Artikel) artikeldaten.getData ()).name;
          return null;
          }


/*********************************************************************************************
    Ab hier werden die Rechnungsdaten behandelt
*********************************************************************************************/


    /**
     * Methode: ReadRechnungsdaten.
     * Ließt alle Rechnungsfiles aus dem angegebenen Verzeichnis ein.
     * @param String pfad: Verzeichnis aus dem die Rechnungsfiles gelesen werden sollen.
     * @exception MyException Wirft bei Fehler eine Meldung an aufrufende Methode.
     * @see MyException
     */
    public void ReadRechnungsdaten (String pfad) throws MyException {
          // Öffnet Directory
          File file = new File (pfad);

          // Ließt alles Filenamen im angegebenen Directory ein
          String f[] = file.list ();

          for (int i = 0; i < f.length; i++)
              if ((f[i].length() > 14) && (f[i].substring (0, 8).equalsIgnoreCase ("rechnung")) && (f[i].substring
(f[i].length() - 4, f[i].length()).equalsIgnoreCase (".txt")))
                    {
                    StringTokenizer st = new StringTokenizer (f[i], ".");
                    if ((st.countTokens() == 3) && (isValidNumber (st.nextToken())) && (isValidKassa (st.nextToken())))
                        ReadRechnungsFile (pfad, f[i]);
                    }
          }
```

```
    /**
     * Methode: ReadRechnungsFile.
     * Ließt die Rechnungen aus angegebenem File ein.
     * @param pfad - gibt den Pfad an, in dem sich das Rechnungsfile befindet.
     * @param filename - gibt den Filenamen an.
     * @exception MyException Wirft bei Fehler eine Meldung an aufrufende Methode.
     * @see MyException
     */
    void ReadRechnungsFile (String pfad, String filename) throws MyException
          {
          Liste rechnung = new Liste ();
          String s = null;
          int artind = 0;
          ReadFile file;
          boolean err;

          // Öffnet File
          file = new ReadFile (pfad, filename);

          // Ließt bis zum Fileende ein, oder bricht bei mehr als 10 Rechnungssätzen ab
          while ((s = file.readLine()) != null) {
              // Wenn Leerzeile, nächste Zeile einlesen
              if (s.length () == 0) continue;

              err = false;
              RechnungsElement h = null;
              // Wandelt Sting in Rechnungsdatensatz um
              try {h = StringtoRechnung (s);}
              catch (MyException e) {
                    System.out.println (e.getMessage ());
                    err = true;}

              // Überprüft, ob dieser Artikel in der Rechnung schon vorkommt
              //  ja: ergänzt den alten Rechnungssatz
              //  nein: legt neuen Rechnungssatz an
              if (!err)
                    if (RechnungFind (h.artikel, rechnung))
                        ((RechnungsElement) rechnung.getData ()).menge += h.menge;
                    else
                        rechnung.add (new RechnungsElement (h.artikel, h.menge));
              }


          if (!rechnung.isEmpty())
              rechnungsdaten.add (new Rechnung (Integer.parseInt(filename.substring (8, filename.indexOf (".".))),
Integer.parseInt (filename.substring (filename.indexOf (".")+1, filename.length () - 4)), rechnung));
          }


    /**
     * Methode StringtoRechnung.
     * Versucht den übergebenen String in das Datenformat umzuwandeln.
     * @param s - der umzuwandelnde String.
     * @exception MyException Wirft bei Fehler eine Meldung an aufrufende Methode.
     * @see MyException
     * @return Daten - Rechnungsdatensatz.
     */
    private RechnungsElement StringtoRechnung (String s) throws MyException
          {
          int artikel = 0, stück = 0, artnr = 0;
          String help = null;

          // Initialisiert Stringteiler und definiert die Trennzeichen
          StringTokenizer st = new StringTokenizer (s, ", ");

          try {
              if ((st.countTokens () != 2) || (s.charAt (s.length () - 1) == ','))
                    throw new MyException ("Falsche Syntax: " + s + " - Datensatz ignoriert.");
              artikel = Integer.parseInt (st.nextToken());
              stück = Integer.parseInt (st.nextToken());
              }
          catch (NumberFormatException e)
              {
              throw new MyException ("Fehler im Datensatz: " + s + " - Datensatz ignoriert.");
              }

          return new RechnungsElement (artikel, stück);
          }


    /**
     * Methode: isValidNumber (String s).
     * Überprüft, ob der Filename eine gültige Rechnungsnummer enthält.
     * @param s - Erster Teil des Filenamens.
     * @return true wenn gültig.
     */
    private boolean isValidNumber (String s) {
        try {if (Integer.parseInt (s.substring (8, s.length())) <= 0)
                    return false;}
          catch (NumberFormatException e) {return false;}
          return true;
          }


    /**
     * Methode: isValidKassa (String s).
     * Überprüft, ob der Filename eine gültige Kassennummer enthält.
     * @param s - Zweiter Teil des Filenamens.
     * @return true wenn gültig.
     */
    private boolean isValidKassa (String s) {
        try {if (Integer.parseInt (s) <= 0)
                    return false;}
          catch (NumberFormatException e) {return false;}
          return true;
          }
```

```
    /**
     * Durchsucht eine Rechnung nach einem bestimmten Artikel.
     * @param art - der gesuchte Artikel.
     * @param d - die zu untersuchende Rechnung.
     * @return boolean - true wenn der Artikel gefunden wurde.
     */
    private boolean RechnungFind (int art, Liste d) {
          if (d.isEmpty ()) return false;
          for (d.gotoStart (); !d.eol (); d.gotoNext())
              if (((RechnungsElement) d.getData ()).artikel == art)
                    return true;
          return false;
          }


    /**
     * Methode addRechnung ().
     * Fügt neuen Datensatz hinzu.
     * @param nr Die neue Rechnungsnummer.
     * @param kassa Die Kassa, bei der Bezahlt wurde.
     * @param l Die gekauften Artikel.
     */
    public void addRechnung (int nr, int kassa, Liste l) {
          rechnungsdaten.add (new Rechnung (nr, kassa, l));
          Changed ();
          }


    /**
     * Methode addrechnungsElement ().
     * Fügt neuen Datensatz hinzu.
     * @param d die Liste mit den Elementen.
     * @param artikel die neue Artikelnummer.
     * @param menge die konsumierte Menge.
     */
    public void addrechnungsElement (Liste d, int artikel, int menge) {
          d.add (new RechnungsElement (artikel, menge));
          Changed ();
          }


/*********************************************************************************************
    Ab hier werden Gruppendaten behandelt
*********************************************************************************************/


    /**
     *  Methode: ReadGruppe.
     *  Ließt das durch pfad und filename angegeben File ein und Wertet den Inhalt als Gruppen aus.
     *  Falls der Inhalt gültig ist, wird er in einer Liste abgelegt.
     *  @param String pfad: Gibt das Verzeichnis des einzulesenden Files an.
     *  @param String filename: Gibt den Filenamen des einzulesenden Files an.
     *  @see StringtoGruppe
     *  @exception MyException Wirft bei Fehler eine Meldung an aufrufende Methode.
     *    @see MyException
     */
   private void ReadGruppendaten (String pfad, String filename) throws MyException {
          String s = null;
          ReadFile file;

          // Versucht File zu öffnen. Wenn nicht möglich, Abbruch
          file = new ReadFile (pfad, filename);

          // Ließt solange Gruppen ein, bis keine mehr vorhanden sind
          while ((s = file.readLine()) != null) {
              // Wenn Leerzeile, nächste Zeile einlesen
              if (s.length () == 0) continue;

              try {
                    // Wandelt einen String ins Gruppenformat um
                    StringtoGruppe (s);
                    }
              catch (MyException e) {
                    // Fehlermeldung (ErrorDialog)
                    System.out.println ("Fehler: " + e.getMessage());
                    }
              }

          file.close ();
          if (gruppendaten.isEmpty())
              throw new MyException ("Es wurden keine Daten eingelesen - Programmabbruch");
          }
```

```
    /**
     * Methode: String to Gruppe.
     * Versucht den übergebenen String in der Liste "Gruppendaten" als "Gruppe" abzulegen.
     * @param s - der umzuwandelnde String.
     * @exception MyException Wirft bei Fehler eine Meldung an aufrufende Methode.
     * @see MyException
     * @see Gruppe
     */
    private void StringtoGruppe (String s) throws MyException
          {
          int nummer = 0;
          String name = null;
          double preis = 0;

          // Initialisiert Stringteiler und definiert die Trennzeichen
          StringTokenizer T = new StringTokenizer (s, ", ");

          try {
              // Überprüft, ob die Parameteranzahl korrekt ist
              if ((T.countTokens () != 2) || (s.charAt (s.length () - 1) == ','))
                    throw new MyException ("Falsche Syntax: " + s + " - Datensatz ignoriert.");

              // Ließt die Gruppennummer, wenn nicht einstellig Abbruch
              name = T.nextToken ();
              if (name.length() !=1)
                    throw new MyException ("Die Gruppennummer " + nummer + " ist ungültig. Sie muss 3stellig sein. Datensatz
ignoriert.");
              nummer = Integer.parseInt (name);

              if ((nummer < 1) || (nummer > 3))
                    throw new MyException ("Die Gruppennummer " + nummer +  " ist ungültig. Sie muss zw. 1 und 3 liegen.
Datensatz ignoriert.");

              // Ließt Gruppennamen, Überprüft, ob der Gruppenname zulässig ist, und bricht bei Fehler ab
              name = T.nextToken ();
              if (name.length() > 20)
                    throw new MyException ("Gruppenbezeichnung zu lang (max. 20 Zeichen). Datensatz ignoriert.");
              }
          catch (NumberFormatException e) {
              throw new MyException (e.getMessage());
              }

          // Wenn kein Fehler bei der Umwandlung aufgetreten ist, Datensatz speichern
          gruppendaten.add (new Gruppe (nummer, name));
          }
    }
```

3.2.1 Artikel

```
/**
 * class Artikel.
 * In diesem Format werden die Artikeldaten abgelegt.
 */
class Artikel
    {
    int artikel = 0;
    String name = null;
    double preis = 0;

    /**
     * Konstruktor für ein neues Element.
     * @param nr - die Artikelnummer.
     * @param na - der Artikelname.
     * @param p - der Preis des Artikels.
     */
    Artikel (int nr, String na, double p)
        {
        artikel = nr;
        name = na;
        preis = p;
          }
    }
```

3.2.2 RechnungsDaten

```
/**
 * Klasse RechnungsDaten.
 * In diesem Format wird eine gesamte Rechnung abgelegt.
 */
class Rechnung
    {
    int rechnungsNr, kassa;
    Liste elements;

    /**
     * Konstruktor für ein neues Element.
     * @param r - die Rechnungsnummer.
     * @param k - die Kassennummer.
     * @param e - die verrechneten Elemente (Artikel).
     */
    Rechnung (int r, int k, Liste e)
        {
        rechnungsNr = r;
        kassa = k;
        elements = e;
        }
    }
```

3.2.3 RechnungsElement

```
/**
 * Klasse RechnungsElement.
 * In diesem Format werden die einzelnen Elemente einer Rechnung gespeichert.
 */
class RechnungsElement {
    int artikel, menge;

    /**
     * Konstruktor für ein neues Element.
     * @param a - die Artikelnummer.
     * @param m - die Artikelmenge.
     */
    RechnungsElement (int a, int m)
        {
          artikel = a;
          menge = m;
        }
    }
```

3.2.4 Gruppe

```
/**
 * class Gruppe.
 * In diesem Format werden die einzelnen Elemente einer Rechnung gespeichert.
 */
class Gruppe {
    int artikel;
    String name;

    /**
     * Konstruktor für ein neues Element.
     * @param a - die Artikelgruppennummer (1-3).
     * @param n - der Artikelgruppenname.
     */
    Gruppe (int a, String n)
        {
        artikel = a;
        name = n;
          }
    }
```

3.3 Diagramm.java

```
import java.awt.*;


/**
 * class Diagramm.
 * Oberklasse für Kreis- und Balkendiagramm.
 * @see Kreisdiagramm
 * @see Balkendiagramm
 */
abstract class Diagramm {
    protected int hspace = 0, xbez = 20, breite = 20, maxX, maxY, ybez;
    protected MyColors m = new MyColors ();
    protected Liste werte;
    protected Liste texte;
    protected FontMetrics fm;
    protected Graphics g;
    protected Dimension groesse;
    protected Canvas output;
    protected ScrollPane scrollpane;


    /**
     * Konstruktor.
     * Übernimmt die übergebenen Referenzen und startet die Zeichnung des Diagrammes.
     * @param gr die Graphicsreferenz.
     * @param data die Diagarammdaten (Beschriftung und Werte).
     * @see Diagrammdaten
     * @param sp der ScrollPane, in dem der Canvas liegt.
     * @param op der Canvas, in dem die Ausgabe erfolgt.
     */
    public Diagramm (Graphics gr, Diagrammdaten data, ScrollPane sp, Canvas op) {
          // Scrollpane, Graphics, Output (Canvas) und Daten übernehmen
          scrollpane = sp;
          g = gr;
          output = op;
          texte = data.getNames ();
          werte = data.getValues ();

          // Diagramm ausgeben
          paintDiagramm ();
          }
```

```
    /**
     * methode fitFont ().
     * Berechnet die optimale Schriftgrösse.
     * @param max gibt die max. Grösse der Schrift in Pixel an.
     * @return Font die optimale Schriftart.
     */
    protected Font fitFont (int max) {
          FontMetrics fm;
          Font font = null;

          int groesse = 20;
          while (groesse > 6) {
              font = new Font ("Serif", Font.PLAIN, groesse);
              fm = g.getFontMetrics (font);
              if (fm.getHeight () + 5 < max) break;
              groesse--;
              }
          return font;
          }


    /**
     * abstract methode paintDiagramm ().
     * Zeichnet das jeweilige Diagramm.
     */
    public abstract void paintDiagramm ();
    }
```

3.3.1 Balkendiagramm

```
/**
 * class Balkendiagramm extends Diagramm.
 * Hier werden Balkendiagramme ausgegeben.
 * @see Diagramm
 */
class Balkendiagramm extends Diagramm {

    /**
     * Konstruktor.
     * Startet Superkonstruktor.
     * @param g die Graphicsreferenz.
     * @param data die Diagarammdaten (Beschriftung und Werte).
     * @see Diagrammdaten
     * @param scrollpane der ScrollPane, in dem der Canvas liegt.
     * @param ootput der Canvas, in dem die Ausgabe erfolgt.
     */
    public Balkendiagramm (Graphics g, Diagrammdaten data, ScrollPane scrollpane, Canvas output) {
        super (g, data, scrollpane, output);
        }

    /**
     * methode paintDiagramm ().
     * Zeichnet ein Balkendiagramm.
     */
    public void paintDiagramm () {
        int balken = 0;
        double fact = 0.0;

          // Berechnet Werte Grösse und Einteilung des Canvas
        groesse = scrollpane.getViewportSize ();
        maxX = groesse.width * 7 / 10;
        maxY = groesse.height * 7 / 10;
        ybez = groesse.height / 10;

          // Löscht altes Diagramm
        g.clearRect (0, 0, groesse.width, groesse.height);

          // Setzt Schriftgrösse ein und Berechnet die Fontmetrics
        g.setFont (fitFont (ybez));
        fm = g.getFontMetrics ();

          // Berechnet die max. Breite einer Bezeichnung + höchsten Balken
        for (texte.gotoStart (), werte.gotoStart (); !texte.eol (); texte.gotoNext (), werte.gotoNext ())
              if (Integer.parseInt ((String) werte.getData ()) != 0) {
                    balken ++;
                    if (fm.stringWidth ((String) texte.getData ()) + 10 > hspace)
                        hspace = fm.stringWidth((String) texte.getData ()) + 10;
                    if (fm.stringWidth((String) werte.getData ()) + 10 > hspace)
                        hspace = fm.stringWidth ((String) werte.getData ()) + 10;
                    if (fact < (1.0 * Integer.parseInt ((String) werte.getData ())/(maxY)))
                        fact = 1.0 * Integer.parseInt ((String) werte.getData ())/(maxY);
                    }

          if (fact == 0)
              fact = 1;
          if (hspace <= breite)
              hspace = breite * 2;

          if ((balken * hspace + 2 * xbez) < (groesse.width + 20))
              // Passt den Canvas an das auszugebende Diagramm an
              output.setSize (groesse.width, groesse.height);
          else
              // Passt den Canvas an das auszugebende Diagramm an
              output.setSize (balken * hspace + 2 * xbez, groesse.height);

          scrollpane.doLayout ();
```

```
         // Gibt das Diagramm aus
         int i = 0;
         for (texte.gotoStart (), werte.gotoStart (); !texte.eol (); texte.gotoNext (), werte.gotoNext ())
             if (Integer.parseInt ((String) werte.getData ()) != 0) {
                   g.setColor (m.getNext ());
                   g.fillRect(xbez + hspace*i + hspace/2 - breite/2, maxY + ybez*3/2 - (int)Math.round(Integer.parseInt
((String) werte.getData ()) / fact), breite, (int)Math.round(Integer.parseInt ((String) werte.getData ()) / fact));
                   g.setColor (Color.black);
                   drawStringZentriert ((String) texte.getData (), xbez + hspace*i, maxY + ybez*5/2, hspace);
                   drawStringZentriert ((String) werte.getData (), xbez + hspace*i, maxY + ybez -
(int)Math.round(Integer.parseInt ((String) werte.getData ()) / fact), hspace);
                   g.drawRect(xbez + hspace*i + hspace/2 - breite/2, maxY + ybez*3/2 - (int)Math.round(Integer.parseInt
((String) werte.getData ()) / fact), breite, (int)Math.round(Integer.parseInt ((String) werte.getData ()) / fact));
                   i++;
                   }
         }


    /**
     * methode drawStringZentriert ().
     * Gibt einen String zentriert aus.
     * @param s der auszugebende String.
     * @param x die x-Position des Strings.
     * @param y die y-Position des Strings.
     * @param m die Grösse des grössten auszugebenden Strings
     */
    private void drawStringZentriert (String s, int x, int y, int m) {
          FontMetrics fm = g.getFontMetrics ();
          int w = fm.stringWidth (s);

          g.drawString (s, x + (int)m/2 - (int)w/2, y);
          }
    }
```

3.3.2 Kreisdiagramm

```
/**
 * class Kreisdiagramm extends Diagramm.
 * Hier werden Kreisdiagramme ausgegeben.
 * @see Diagramm
 */
class Kreisdiagramm extends Diagramm {

    /**
     * Konstruktor.
     * Startet Superkonstruktor.
     * @param g die Graphicsreferenz.
     * @param data die Diagarammdaten (Beschriftung und Werte).
     * @see Diagrammdaten
     * @param scrollpane der ScrollPane, in dem der Canvas liegt.
     * @param ootput der Canvas, in dem die Ausgabe erfolgt.
     */
    public Kreisdiagramm (Graphics g, Diagrammdaten data, ScrollPane scrollpane, Canvas output) {
          super (g, data, scrollpane, output);
          }

    /**
     * methode paintDiagramm ().
     * Zeichnet ein Kreisdiagramm.
     */
    public void paintDiagramm () {
        int ges = 0, radius;
          double start = 0.0;

          // Berechnet Werte Grösse und Einteilung des Canvas
        groesse = scrollpane.getViewportSize ();
          maxX = groesse.width * 7 / 10;
        maxY = groesse.height * 7 / 10;
        ybez = groesse.height / 10;

          // Löscht altes Diagramm
        g.clearRect (0, 0, groesse.width, groesse.height);

          // Setzt Schriftgrösse ein und Berechnet die Fontmetrics
        g.setFont (fitFont (ybez));
        fm = g.getFontMetrics ();

          // Berechnet Werte Grösse und Einteilung des Canvas
        radius = Math.min (maxX, maxY)/2;

          // Berechnet die Gesamtanzahl der Artikel
        for (werte.gotoStart (); !werte.eol (); werte.gotoNext ())
             if (Integer.parseInt ((String) werte.getData ()) != 0)
                   ges += Integer.parseInt ((String) werte.getData ());

          if (hspace <= breite)
              hspace = breite * 2;

          // Passt den Canvas an das auszugebende Diagramm an
          output.setSize (groesse.width, groesse.height);
          scrollpane.doLayout ();
```

```
         // Gibt das Diagramm aus
         g.setColor (Color.black);
         g.fillOval (groesse.width/2 - radius - 2, groesse.height/2 - radius - 2, radius*2 + 4, radius*2 + 4);

       for (werte.gotoStart (), texte.gotoStart (); !werte.eol (); werte.gotoNext (), texte.gotoNext ())
            if (Integer.parseInt ((String) werte.getData ()) != 0) {
                  double bogen = 360.0 * Integer.parseInt ((String) werte.getData ()) / ges;
                  g.setColor (m.getNext ());
                  g.fillArc (groesse.width/2 - radius, groesse.height/2 - radius, radius*2, radius*2, (int) Math.round
(start), (int) Math.round (bogen));
                  g.setColor (Color.black);
                  drawKreisBeschriftung ((String) texte.getData (), groesse.width/2 - radius, groesse.height/2 - radius,
(int) Math.round (1.5 * radius), (int) Math.round (start), (int) Math.round (bogen));
                  start += bogen;
                }
         }


   /**
    * Methode drawKreisBeschriftung.
    * Berechnet die Koordinate für die Beschriftung der Kreissegmente und
    * ruft die Ausgabe des Textes auf. Ausserdem werden die Randlinien der Segmente gezeichnet.
    * @param s der auszugebende String.
    * @param x die x-Koordinate des Kreises (links).
    * @param y die y-Koordinate des Kreises (oberen).
    * @param r der Radius des Kreises.
    * @param wink der Bogenwinkel des Kreises.
    * @param endwink der Endwinkel des Kreises.
    */
   private void drawKreisBeschriftung (String s, int x, int y, int r, int wink, int endwink) {
       int centerX = x + r;
       int centerY = y + r;
       int xbe, ybe = xbe = 0;
       double store1 = 1.0*wink;
       double store2 = 1.0*endwink;
       double store3 = 2.0*r;
       double store4 = 2.0*r;

       double BWink = 1.0*(((store1+(store2/2))/360.0)*(Math.PI)*2.0);
       double LWink = 1.0*((store1/360.0)*(Math.PI)*2.0);
       double sinLWink = Math.sin (LWink);
       double cosLWink = Math.cos (LWink);
       double sinBWink = Math.sin (BWink);
       double cosBWink = Math.cos (BWink);

       // Sektor von 0 bis 90 Grad
       if (BWink <= Math.PI/2) {
             xbe = (int) (centerX + cosBWink*store3/2);
             ybe = (int) (centerY - sinBWink*store4/2);
             drawText (s, xbe, ybe, false, true);
             g.drawLine (centerX, centerY, (int) (centerX + cosLWink*store3/2), (int) (centerY - sinLWink*store4/2));
             }

       // Sektor von 90 bis 180 Grad
         else if (BWink <= Math.PI && BWink > Math.PI/2) {
             xbe = (int) (centerX + cosBWink*store3/2);
             ybe = (int) (centerY - sinBWink*store4/2);
             drawText (s, xbe, ybe, true, true);
             g.drawLine (centerX, centerY, (int) (centerX + cosLWink*store3/2), (int) (centerY - sinLWink*store4/2));
             }

         // Sektor von 180 bis 270 Grad
         else if (BWink <= 3*Math.PI/2 && BWink > Math.PI) {
             xbe = (int) (centerX + cosBWink*store3/2);
             ybe = (int) (centerY - sinBWink*store4/2);
             g.drawLine (centerX, centerY, (int) (centerX + cosLWink*store3/2), (int) (centerY - sinLWink*store4/2));
             drawText (s, xbe, ybe, true, false);
             }

         // Sektor von 270 bis 360 Grad
         else if (BWink > 3*Math.PI/2 && BWink <= 2*Math.PI ) {
             xbe = (int) (centerX + cosBWink*store3/2);
             ybe = (int) (centerY - sinBWink*store4/2);
             g.drawLine (centerX, centerY, (int) (centerX + cosLWink*store3/2), (int) (centerY - sinLWink*store4/2));
             drawText (s, xbe, ybe, false, false);
             }
         }
```

```
    /**
     * Methode drawText ().
     * Gibt übergebenen String mit der gewünschten Ausrichtung aus.
     *    - Links-/Rechtsbündig.
     *  - An der oberen/unteren Kante ausgerichtet.
     * @param s der auszugebende String.
     * @param x die x-Position des Strings.
     * @param y die y-Position des Strings.
     * @param left true, wenn der String linksbündig ausgegeben werden soll.
     * @param top true, wenn der String an der oberen Kante ausgerichtet sein soll.
     */
    private void drawText (String s, int x, int y, boolean left, boolean top) {
          FontMetrics fm = g.getFontMetrics ();
          int w = fm.stringWidth (s);
          int h = fm.getMaxAdvance();
          int c = fm.charWidth (' ');

          if (left)
              if (top)
                    g.drawString (s, x - w - c, y);
              else
                    g.drawString (s, x - w - c, y + h);
          else
              if (top)
                    g.drawString (s, x + c, y);
              else
                    g.drawString (s, x + c, y + h);
          }
    }
```

3.4 Diagrammdaten.java

```
/**
 * Abstract Class Diagrammdaten.
 * Sie garantiert die Daten, die zur Anzeige des Diagramms benötigt werden.
 * Aus ihr werden 3 Klassen abgeleitet: KassaDaten, GruppenDaten, ArtikelDaten.
 * Diese berechnen jeweils die benötigten Daten (Beschriftung, Anzahl).
 * @version        1.0 15 Jan 1999
 * @author         Sascha Nemecek
 */
abstract class Diagrammdaten {
    Daten daten = null;


    /**
     * Konstruktor.
     * @param daten Die gesamten Daten.
     */
    public Diagrammdaten (Daten daten) {
          this.daten = daten;
          }


    /**
     * Abstract Method getNames.
     * Liefert die Beschriftung (Artikel-, Gruppenname oder Kassa) für Diagramm.
     * @return Liste mit den Beschriftungen
     */
    abstract public Liste getNames ();


    /**
     * Abstract Method getValues.
     * Liefert die Anzahl (Artikel-, Gruppenmenge oder Stk/Kassa) für Diagramm.
     * @return Liste mit den Werten
     */
    abstract public Liste getValues ();


    /**
     * Abstract Method CountOccurence.
     * Berechnet die Gesamtmenge einer Artikelgruppe/eines Artikels in einer Rechnung.
     * @param art die/der gesuchte Artikelgruppe/Artikel.
     * @param d die zu untersuchende Rechnung.
     * @return die Menge der gesuchten Gruppe/Artikel in dieser Rechnung.
     */
    abstract protected int CountOccurence (int art, Liste d);


    /**
     * getArtikelMenge.
     * Berechnet die Gesamtmenge eines Artikels/einer Artikelgruppe in allen Rechnungen.
     * @param art der gesuchte Artikel.
     * @param d Liste mit den Rechnungen.
     * @return die Menge des/der gesuchten Artikels/Artikelgruppe in allen Rechnungen.
     */
    protected int getArtikelMenge (int art, Liste d) {
          int occ = 0;
          for (d.gotoStart (); !d.eol (); d.gotoNext ())
              occ += CountOccurence (art, ((Rechnung) d.getData ()).elements);
          return occ;
          }
    }
```

3.4.1 KassaDaten

```
/**
 * Class KassaDaten.
 * Liefert die benötigten Daten zur Anzeige der verkauften Artikel pro Kassa.
 * @version        1.0 15 Jan 1999.
 * @author         Sascha Nemecek.
 */
class KassaDaten extends Diagrammdaten {
```

```
    public KassaDaten (Daten daten) {
          super (daten);
          }


    /**
     * Method getNames.
     * Liefert die Beschriftung (Kassennummer) für Diagramm.
     * @return Liste mit den Kassennummern.
     */
    public Liste getNames () {
          Liste grp = new Liste ();
          Liste d = daten.rechnungsdaten;

          for (d.gotoStart (); !d.eol (); d.gotoNext ())

              // Überprüft, ob diese Kassa in der Liste schon vorkommt
              //  ja: lässt die List
              //  nein: legt neues Element an
              if (!KassaExists (grp, ((Rechnung) d.getData ()).kassa))
                    grp.add ("" + ((Rechnung) d.getData ()).kassa);
          return grp;
          }


    private boolean KassaExists (Liste grp, int kassa) {
          for (grp.gotoStart (); !grp.eol (); grp.gotoNext ())
              if (((String) grp.getData ()).equals ("" + kassa))
                    return true;
          return false;
          }


    /**
     * Method getValues.
     * Liefert die Anzahl der verkauften Artikel pro Kassa für Diagramm.
     * @return Liste mit den verkauften Stück pro Kassa.
     */
    public Liste getValues () {
          Liste names = getNames ();
          Liste grp = new Liste ();

          for (names.gotoStart (); !names.eol (); names.gotoNext ())
              grp.add ("" + getArtikelMenge (Integer.parseInt ((String) names.getData ()), daten.rechnungsdaten));
          return grp;
          }


    /**
     * getArtikelMenge.
     * Overwrites Diagrammdaten.getArtikelMenge.
     * Berechnet die Gesamtmenge eines Artikels in einer Kassa.
     * @param kas die gesuchte Kassa.
     * @param d Liste mit den Rechnungen.
     * @return die Menge der gekaufen Artikel pro Kassa.
     */
    protected int getArtikelMenge (int kas, Liste d) {
          int occ = 0;
          for (d.gotoStart (); !d.eol (); d.gotoNext ())
              if (((Rechnung) d.getData ()).kassa == kas)
                    occ += CountOccurence (0, ((Rechnung) d.getData ()).elements);
          return occ;
          }


    /**
     * CountOccurence.
     * Berechnet die Gesamtmenge der verkauften Artikel in einer Rechnung.
     * @param art ohne Funktion.
     * @param d die zu untersuchende Rechnung.
     * @return die Menge der Artikel in dieser Rechnung.
     */
    protected int CountOccurence (int art, Liste d) {
          int i = 0;
          for (d.gotoStart (); !d.eol (); d.gotoNext ())
              i += ((RechnungsElement)d.getData ()).menge;
          return i;
          }
    }
```

3.4.2 GruppenDaten

```
/**
 * Class GruppenDaten.
 * Liefert die benötigten Daten zur Anzeige der verkauften Artikel pro Kassa.
 * @version        1.0 15 Jan 1999.
 * @author         Sascha Nemecek.
 */
class GruppenDaten extends Diagrammdaten {

    public GruppenDaten (Daten daten) {
          super (daten);
          }


    /**
     * Method getNames.
     * Liefert die Beschriftung (Gruppennamen) für das Diagramm.
     * @return Liste mit den Gruppennamen.
     */
    public Liste getNames () {
          Liste grp = new Liste ();
          Liste d = daten.gruppendaten;

          for (d.gotoStart (); !d.eol (); d.gotoNext ())
              grp.add (((Gruppe) d.getData ()).name);
```

```
          return grp;
          }


    /**
     * Method getValues.
     * Liefert die Anzahl der verkauften Artikel pro Artikelgruppe für das Diagramm.
     * @return Liste mit den verkauften Artikel pro Artikelgruppe.
     */
    public Liste getValues () {
          Liste grp = new Liste ();
          Liste g = daten.gruppendaten;

          for (g.gotoStart (); !g.eol (); g.gotoNext ())
              grp.add ("" + getArtikelMenge (((Gruppe) g.getData ()).artikel, daten.rechnungsdaten));
          return grp;
          }


    /**
     * CountOccurence.
     * Berechnet die Gesamtmenge einer Artikelgruppe in einer Rechnung.
     * @param art die gesuchte Artikelgruppe.
     * @param d die zu untersuchende Rechnung.
     * @return die Menge der gesuchten Gruppe in dieser Rechnung.
     */
    protected int CountOccurence (int art, Liste d) {
          int i = 0;
          for (d.gotoStart (); !d.eol (); d.gotoNext ())
              if (art == ((RechnungsElement)d.getData ()).artikel/100)
                    i += ((RechnungsElement)d.getData ()).menge;
          return i;
          }
    }
```

3.4.3 ArtikelDaten

```
/**
 * Class ArtikelDaten.
 * Liefert die benötigten Daten zur Anzeige der verkauften Artikel.
 * @version        1.0 15 Jan 1999.
 * @author         Sascha Nemecek.
 */
class ArtikelDaten extends Diagrammdaten {
    public ArtikelDaten (Daten daten) {
          super (daten);
          }


    /**
     * Method getNames.
     * Liefert die Beschriftung (Artikelnamen) für das Diagramm.
     * @return Liste mit den Artikelnamen.
     */
    public Liste getNames () {
          Liste grp = new Liste ();
          Liste d = daten.artikeldaten;

          for (d.gotoStart (); !d.eol (); d.gotoNext ())
              grp.add (((Artikel) d.getData ()).name);
          return grp;
          }


    /**
     * Method getValues.
     * Liefert die Anzahl der verkauften Artikel für das Diagramm.
     * @return Liste mit der Anzahl der verkauften Artikel.
     */
    public Liste getValues () {
          Liste grp = new Liste ();
          Liste g = daten.artikeldaten;

          for (g.gotoStart (); !g.eol (); g.gotoNext ())
              grp.add ("" + getArtikelMenge (((Artikel) g.getData ()).artikel, daten.rechnungsdaten));
          return grp;
          }


    /**
     * CountOccurence.
     * Berechnet die Gesamtmenge eines Artikels in einer Rechnung.
     * @param art der gesuchte Artikel.
     * @param d die zu untersuchende Rechnung.
     * @return die Menge des gesuchten Artikels in dieser Rechnung.
     */
    protected int CountOccurence (int art, Liste d) {
          int i = 0;
          for (d.gotoStart (); !d.eol (); d.gotoNext ())
              if (art == ((RechnungsElement)d.getData ()).artikel)
                    i += ((RechnungsElement)d.getData ()).menge;
          return i;
          }
    }
```

3.5 Dialoge.java

```
import java.awt.*;
import java.awt.event.*;
import java.util.*;


/**
 * Dialog MyDialog.
```

```
 * Abstrakte Klasse für Dialoge
 */
abstract class MyDialog extends Dialog {

    /**
     * Konstruktor MyDialog ().
     * Öffnet einen neuen Dialog.
     * @param p der Mutterframe.
     * @param title der Dialogname.
     */
    MyDialog (Frame p, String title) {
          super (p, title);
          }

    /**
     * Konstruktor MyDialog ().
     * Öffnet einen neuen Dialog.
     * @param p der Mutterframe.
     * @param title der Dialogname.
     * @param mod true, wenn der Dialog Modal ist.
     */
    MyDialog (Frame p, String title, boolean mod) {
          super (p, title, mod);
          }


    /**
     * abstrakte Methode cancel ().
     * Beendet den Dialog und kehrt zurück zum Hauptframe.
     */
    abstract public void cancel ();
    }


/**
 * class Dialogschliesser.
 * Schliesst den Dialog.
 */
class Dialogschliesser extends WindowAdapter {

    /**
     * Referenz auf den Dialog, dem der Dialogschliesser gehört.
     */
    MyDialog dialog;

    /**
     * Konstruktor.
     */
    Dialogschliesser (MyDialog d) {
          super ();
          this.dialog = d;
          }

    /**
     * Methode windowClosing ().
     * startet die Schliessung des Fensters.
     */
    public void windowClosing (WindowEvent e) {
          dialog.cancel ();}
    }
```

3.5.1 DialogBearbeiten

```
/**
 * DialogBearbeiten.
 * Ermöglicht das Bearbeiten einer oder mehrerer Rechnungen.
 * Zur Zeit ist nur das Löschen von Artikeln in Rechnungen und das bearbeiten von Artikeln möglich.
 */
class DialogBearbeiten extends MyDialog implements ActionListener, ItemListener, Observer {
    private Choice choiceRechnungen = new Choice();
    private Panel pEingabe = new Panel ();
    private List lElements = new List (5, false);
    private Button bNeu = new Button ("Neu");
    private Button bEd = new Button ("Bearbeiten");
    private Button bDel = new Button ("Löschen");
    private Button bOk = new Button ("OK");
    private TextField tfArt = new TextField ();
    private Daten daten;
    private Liste rechnungsDaten;
    private Frame parent;


    /**
     * Konstrukor.
     * Hier wird der Dialog initialisiert.
     */
    DialogBearbeiten (Frame p, String titel, boolean modal, Daten d) {
          super(p,titel);
          parent = p;
          daten = d;
          rechnungsDaten = daten.rechnungsdaten;

          setBackground(Color.lightGray);
          setResizable(false);
          setLayout(new BorderLayout());

          // Liste füllen
        for (rechnungsDaten.gotoStart (); !rechnungsDaten.eol (); rechnungsDaten.gotoNext ())
              choiceRechnungen.add ("" + ((Rechnung) rechnungsDaten.getData()).rechnungsNr);

          // Füllt die Liste
          fillList ();

          pEingabe.setLayout(new GridLayout(3,1));
```

```
          pEingabe.add(bNeu);
          pEingabe.add(bEd);
          pEingabe.add(bDel);

          choiceRechnungen.addItemListener(this);
          bOk.addActionListener(this);
          bNeu.addActionListener(this);
          bDel.addActionListener(this);
          bEd.addActionListener(this);

          lElements.addActionListener (this);
          lElements.addItemListener (this);

          add ("North", choiceRechnungen);
          add ("Center", lElements);
          add ("East", pEingabe);
          add ("South", bOk);

          // WindowListener zum Schließen des Dialoges
          addWindowListener(new Dialogschliesser (this));
          pack();
          }


    /**
     * Methode: Update.
     * Führt bei einer Veränderung des zu überwachenden Objektes die fillListmethode des Dialoges aus.
     * @param o - das Überwachte Objekt.
     * @param arg - die Veränderung.
     */
    public void update(Observable o, Object arg) {
          fillList ();
          }


    /**
     * methode fillList.
     * Füllt die Liste mit dem Inhalt einer Rechnung.
     */
    private void fillList () {
          String sRechnung;
          Liste rechnungsElemente = null;

          // selektierte Rechnungsnummer holen
          if ((sRechnung = choiceRechnungen.getSelectedItem()) == null)
              cancel ();

          lElements.removeAll ();

          // Sucht Rechnung
          for (rechnungsDaten.gotoStart (); !rechnungsDaten.eol (); rechnungsDaten.gotoNext ())
              if (((Rechnung) rechnungsDaten.getData()).rechnungsNr == Integer.parseInt (sRechnung)) {
                    rechnungsElemente = ((Rechnung) rechnungsDaten.getData()).elements;

                    for (rechnungsElemente.gotoStart (); !rechnungsElemente.eol (); rechnungsElemente.gotoNext ())
                        lElements.add (daten.getArtikelName (((RechnungsElement) rechnungsElemente.getData ()).artikel) + "
" + ((RechnungsElement) rechnungsElemente.getData ()).menge);
                    }
          }


    /**
     * Methode cancel ().
     * Beendet den Dialog und kehrt zurück zum Hauptframe.
     */
    public void cancel() {
          setVisible(false);
          dispose();
          ((Window)getParent()).toFront();
          getParent().requestFocus();
          }


    /**
     * Methode actionPerformed ().
     * Wertet die ActionEvent aus und reagiert je nach Event.
     * @param e das ausgelößte ActionEvent.
     */
    public void actionPerformed (ActionEvent e) {
          // Beendet Bearbeitung
          if (e.getActionCommand().equals("OK")) cancel();

          // noch nicht ausprogrammiert
          else if (e.getActionCommand().equals("Neu")) {
              String r;
              DialogNeuesElement dNeu;

              // selektierte Rechnungsnummer holen
              if ((r = choiceRechnungen.getSelectedItem()) == null)
                    cancel ();
              else {
                    dNeu = new DialogNeuesElement (parent, r, daten);
                    dNeu.setVisible (true);
                    }
              }

          // Startet einen neuen Dialog zum verändern des ausgewählten Artikels
          else if (e.getActionCommand().equals("Bearbeiten")) {
              String r, s = lElements.getSelectedItem ();
              DialogEditElement dEd;

              // selektierte Rechnungsnummer holen
              if ((r = choiceRechnungen.getSelectedItem()) == null)
                    cancel ();

              if (s != null) {
                    dEd = new DialogEditElement (parent, s, r, daten);
```

```
                    dEd.setVisible (true);
                    }
                }

            // Löscht den ausgewählten Artikeleintrag einer Rechnung
            else if (e.getActionCommand().equals("Löschen")) löschen();
            }


        /**
         * methode itemStateChanged.
         * Verarbeitet die Item Events des Choice- und Listobjekts..
         */
        public void itemStateChanged (ItemEvent e) {
            // Wenn eine andere Rechnung ausgewählt wurde - aktualisieren
            if (e.getSource () != lElements)
                fillList ();
            }


        /**
         * Löscht das selektierte Listenelement.
         */
        public void löschen() {
            String sElement, sRechnung;
            Liste rechnungsElemente = null;

            // selektierte Rechnungsnummer holen
            if ((sRechnung = choiceRechnungen.getSelectedItem()) == null)
                cancel ();

            // Sucht Rechnung
            for (rechnungsDaten.gotoStart (); !rechnungsDaten.eol (); rechnungsDaten.gotoNext ())
                if (((Rechnung) rechnungsDaten.getData()).rechnungsNr == Integer.parseInt (sRechnung))
                    rechnungsElemente = ((Rechnung) rechnungsDaten.getData()).elements;

            // selektierten Artikel holen
            if ((sElement = lElements.getSelectedItem()) != null)
                {
                // Rechnungsnummer aus Choice entfernen
                lElements.remove (sElement);

                // Rechnungsnummer aus Daten entfernen & Observer benachrichtigen
                for (rechnungsElemente.gotoStart (); !rechnungsElemente.eol (); rechnungsElemente.gotoNext ())
                    if (sElement.equals (daten.getArtikelName (((RechnungsElement) rechnungsElemente.getData ()).artikel) + "
" + ((RechnungsElement) rechnungsElemente.getData ()).menge)) {
                        rechnungsElemente.remove ();
                        daten.Changed ();
                        }
                }
            }
        }
```

3.5.2 DialogEditElement

```
/**
 * Klasse DialogEditElement.
 * Ermöglicht das Editieren eines Rechnungselements.
 * Dies geschieht über einen modalen Dialog.
 */
class DialogEditElement extends MyDialog implements ActionListener {
    Button bOk = new Button ("OK");
    Button bCancel = new Button ("Cancel");
    Panel p = new Panel ();
    TextField tf = new TextField ("0", 1);
    Label lb = new Label ();
    Liste artikelDaten;
    Liste rechnungsDaten;
    Daten daten;
    String art;
    String nr;


    /**
     * Kostruktor.
     * Öffnet den modalen Dialog.
     * @param parent Referenz auf das Mutterfenster.
     * @param s der zu verändernde Artikel.
     * @param n die Rechnung die den Artikel beinhaltet.
     * @param d die Daten.
     */
    DialogEditElement (Frame parent, String s, String n, Daten d) {
        super (parent, "Edit", true);
        artikelDaten = d.artikeldaten;
        rechnungsDaten = d.rechnungsdaten;
        daten = d;
        art = s;
        nr = n;

        // Setzt Parameter für Dialog
       setBackground(Color.lightGray);
       setResizable(false);
       setLayout(new BorderLayout());

        // Initialisiert die ActionListener
        bOk.addActionListener (this);
        bCancel.addActionListener (this);
        lb.setText ("Stück: " + s.substring (0, s.indexOf (" ")));
        tf.setText (s.substring (s.indexOf (" ") + 4, s.length ()));

        // Fügt buttons in Panel ein
        p.setLayout (new GridLayout(1, 2));
          p.add (bOk);
```

```
          p.add (bCancel);

        // Positioniert die Objekte im Dialog
        add ("Center", tf);
        add ("North", lb);
        add ("South", p);

       // WindowListener zum Schließen des Dialoges
       addWindowListener(new Dialogschliesser (this));

        // Richtet Objekte ein
       pack();
       }


    /**
     * Methode cancel ().
     * Beendet den Dialog und kehrt zurück zum Mutterfenster..
     */
    public void cancel() {
          setVisible(false);
          dispose();
          ((Window)getParent()).toFront();
          getParent().requestFocus();
          }


    /**
     * Verarbeitet ActionEvents.
     * @param e das ausgelößte Event.
     */
    public void actionPerformed (ActionEvent e) {
          // Beendet Bearbeitung, wenn Eingabe korrekt
          if (e.getActionCommand().equals ("OK"))
              {
              boolean err = false;
              int menge = 0;
              try {menge = Integer.parseInt (tf.getText ());}
              catch (NumberFormatException ex) {err = true;}

              // Wenn kein Fehler aufgetreten ist - sichern
              if ((menge > 0) && !err) {
                    Liste rechnungsElemente = null;

                    // Sucht Rechnung
                    for (rechnungsDaten.gotoStart (); !rechnungsDaten.eol (); rechnungsDaten.gotoNext ())
                        if (((Rechnung) rechnungsDaten.getData()).rechnungsNr == Integer.parseInt (nr))
                              rechnungsElemente = ((Rechnung) rechnungsDaten.getData()).elements;

                    // Sucht Artikel in Rechnung und verändert seine Menge
                    for (rechnungsElemente.gotoStart (); !rechnungsElemente.eol (); rechnungsElemente.gotoNext ())
                        if (art.equals (daten.getArtikelName (((RechnungsElement) rechnungsElemente.getData ()).artikel) + "
" + ((RechnungsElement) rechnungsElemente.getData ()).menge)) {
                              ((RechnungsElement) rechnungsElemente.getData ()).menge = menge;
                              daten.Changed ();
                              }

                    cancel ();
                    }
              }

          // Beendet Bearbeitung sofort - ohne Änderung
          if (e.getActionCommand().equals ("Cancel"))
              cancel ();
          }
    }
```

3.5.3 DialogLöschen

```
/**
 * DialogLöschen.
 * Löschen einer oder mehrerer Rechnungen.
 */
class DialogLoeschen extends MyDialog implements ActionListener, ItemListener {
    List choiceRechnungen = new List (5, false);
    Panel pEingabe = new Panel();
    Button bOk = new Button("Löschen");
    Button bCancel = new Button ("Cancel");
    Daten d;
    Liste daten;


    /**
     * Konstrukor.
     * Hier wird der Dialog initialisiert.
     */
    DialogLoeschen (Frame parent, String titel, boolean modal, Daten v) {
          super(parent,titel);
          d = v;
          daten = d.rechnungsdaten;
          setBackground(Color.lightGray);
          setResizable(false);

          // Liste füllen
        for (daten.gotoStart (); !daten.eol (); daten.gotoNext ())
              choiceRechnungen.add ("" + ((Rechnung) daten.getData()).rechnungsNr);

          pEingabe.setLayout(new GridLayout(2,1));
            pEingabe.add(bOk);
            pEingabe.add(bCancel);
            bOk.addActionListener(this);
            bCancel.addActionListener(this);

          choiceRechnungen.addItemListener(this);
          add("Center", choiceRechnungen);
          add("South", pEingabe);
```

```
        // WindowListener zum Schließen des Dialoges
        addWindowListener(new Dialogschliesser (this));
        pack();
        }


    /**
     * Beendet den Dialog und kehrt zurück zum Hauptframe.
     */
    public void cancel() {
        setVisible(false);
        dispose();
        ((Window)getParent()).toFront();
        getParent().requestFocus();
        }


    /**
     * Wertet die ActionEvent aus und reagiert je nach Event.
     */
    public void actionPerformed (ActionEvent e) {
        // löschen aufrufen
        if (e.getActionCommand().equals("Löschen")) löschen();

        // Zurück zum Hauptframe
        if (e.getActionCommand().equals("Cancel")) cancel();
        }


    /**
     * methode itemStateChanged.
     * Verarbeitet die Item Events der Listbox.
     */
    public void itemStateChanged (ItemEvent e) {}


    /**
     * Methode löschen.
     * Löscht die ausgewählte Rechnung.
     */
    void löschen() {
        // selektierte Rechnungsnummer holen
        String sRechnung = choiceRechnungen.getSelectedItem();

        if (sRechnung != null)
            {
            // Rechnungsnummer aus Choice entfernen
            choiceRechnungen.remove (sRechnung);

            // Rechnungsnummer aus Daten entfernen
            for (daten.gotoStart (); !daten.eol (); daten.gotoNext ())
                if (((Rechnung) daten.getData()).rechnungsNr == Integer.parseInt (sRechnung))
                    {
                    daten.remove ();
                    d.Changed ();
                    }


            if (daten.isEmpty ())
                cancel ();
            }
        }
    }
```

3.5.4 DialogNeu

```
/**
 * DialogNeu.
 * Fügt eine Rechnung hinzu.
 * Noch nicht funktionstüchtig, da noch die Ausprogrammierung der Artikelauswahl fehlt.
 */
class DialogNeu extends MyDialog implements ActionListener, ItemListener {
    private Panel pEingabe = new Panel ();
    private Label errMsg = new Label ("", Label.CENTER);
    private TextField tfNr = new TextField("0", 1);
    private TextField tfKassa = new TextField("0", 1);
    private Button bCancel = new Button ("Cancel");
    private Button bOk = new Button ("OK");
    private Daten daten;
    private Liste rechnungsdaten;


    /**
     * Konstrukor.
     * Hier wird der Dialog initialisiert.
     */
    DialogNeu (Frame parent, String titel, boolean modal, Daten d) {
        super(parent,titel);
        daten = d;
        rechnungsdaten = d.rechnungsdaten;
        setBackground(Color.lightGray);
        setResizable(false);
        setLayout(new BorderLayout());

        pEingabe.setLayout (new GridLayout(3,2));
        pEingabe.add (new Label ("Rechnungnummer:"));
        pEingabe.add (new Label ("Kassa:"));
        pEingabe.add (tfNr);
        pEingabe.add (tfKassa);
        pEingabe.add (bOk);
        pEingabe.add (bCancel);

        bOk.addActionListener (this);
        bCancel.addActionListener (this);
```

```
          add ("Center", pEingabe);
          add ("South", errMsg);

          // WindowListener zum Schließen des Dialoges
          addWindowListener(new Dialogschliesser (this));
          pack();
          }


    /**
     * Beendet den Dialog und kehrt zurück zum Hauptframe.
     */
    public void cancel() {
          setVisible(false);
          dispose();
          ((Window)getParent()).toFront();
          getParent().requestFocus();
          }


    /**
     * Wenn Button gedrückt wurde
     */
    public void actionPerformed (ActionEvent e) {
          // Beendet Bearbeitung
          if (e.getActionCommand().equals("Cancel")) cancel();

          // Fügt neue Rechnung hinzu
          else if (e.getActionCommand().equals("OK")) add();
          }


    /**
     * methode itemStateChanged.
     * Verarbeitet die Item Events des Choice- und Listobjekts..
     */
    public void itemStateChanged (ItemEvent e) {
          }


    /**
     * Fügt die neue Rechnung zu den Daten hinzu.
     */
    public void add() {
          String sElement;
          int nr = 0, kassa = 0;
          boolean err = false;

          errMsg.setText ("");

          try {nr = Integer.parseInt (tfNr.getText ());}
          catch (
              NumberFormatException e) {errMsg.setText ("Ungültige Rechnungsnummer");
              err = true;
              }

          try {kassa = Integer.parseInt (tfKassa.getText ());}
          catch (
              NumberFormatException e) {errMsg.setText ("Ungültige Kassennummer");
              err = true;
              }

          // Wenn die Rechnungsnummer ungültig ist
          if ((nr <= 0) && !err) {
              err = true;
              errMsg.setText ("Rechnungsnummer zu klein");
              }

          // Wenn die Kassennummer ungültig ist
          else if ((kassa <= 0) && !err) {
              err = true;
              errMsg.setText ("Kassennummer zu klein");
              }

          // Wenn die Rechnung schon existiert
          for (rechnungsdaten.gotoStart (); !rechnungsdaten.eol (); rechnungsdaten.gotoNext ())
              if (((Rechnung) rechnungsdaten.getData()).rechnungsNr == nr) {
                    err = true;
                    errMsg.setText ("Rechnung existiert bereits");
                    }

          // Wenn kein Fehler aufgetreten ist, wird die neue Rechnung angelegt
          if (!err) {
              Liste l = new Liste ();

              daten.addRechnung (nr, kassa, l);
              errMsg.setText ("Daten angelegt");
              }
          }
    }
```

3.5.5 DialogNeuesElement

```
/**
 * Klasse DialogNeuesElement.
 * Ermöglicht das Einfügen eines neuen Rechnungselements.
 * Dies geschieht über einen modalen Dialog.
 */
class DialogNeuesElement extends MyDialog implements ActionListener {
    Button bOk = new Button ("OK");
    Button bCancel = new Button ("Cancel");
    Panel p = new Panel ();
    TextField tf = new TextField ("1", 1);
    Choice choiceArtikel = new Choice ();
    Liste artikelDaten;
    Liste rechnungsDaten;
```

```
    Liste rechnungsElemente;
    Daten daten;
    String nr;


    /**
     * Kostruktor.
     * Öffnet den modalen Dialog.
     * @param parent Referenz auf das Mutterfenster.
     * @param n die Rechnung die um einen Artikel erweitert werden soll.
     * @param d die Daten.
     */
    DialogNeuesElement (Frame parent, String n, Daten d) {
          super (parent, "Artikel hinzufügen", true);
          artikelDaten = d.artikeldaten;
          rechnungsDaten = d.rechnungsdaten;
          daten = d;
          nr = n;

          // Setzt Parameter für Dialog
        setBackground(Color.lightGray);
        setResizable(false);
        setLayout(new BorderLayout());

          // Sucht Rechnung
          for (rechnungsDaten.gotoStart (); !rechnungsDaten.eol (); rechnungsDaten.gotoNext ())
              if (((Rechnung) rechnungsDaten.getData()).rechnungsNr == Integer.parseInt (nr))
                        rechnungsElemente = ((Rechnung) rechnungsDaten.getData()).elements;

          // Liste füllen, wobei schon in der Rechnung enthaltene Artikel nicht aufgelistet werden
        for (artikelDaten.gotoStart (); !artikelDaten.eol (); artikelDaten.gotoNext ()) {
              boolean fund = false;
              for (rechnungsElemente.gotoStart (); !rechnungsElemente.eol (); rechnungsElemente.gotoNext ())
                  if (((RechnungsElement) rechnungsElemente.getData ()).artikel == ((Artikel)
artikelDaten.getData()).artikel)
                        fund = true;
              if (!fund) choiceArtikel.add ("" + ((Artikel) artikelDaten.getData()).name);
              }

          bOk.addActionListener (this);
          bCancel.addActionListener (this);
          tf.setText ("1");

          // Fügt buttons in Panel ein
          p.setLayout (new GridLayout(2, 2));
            p.add (new Label ("Menge:"));
            p.add (tf);
            p.add (bOk);
            p.add (bCancel);

          // Positioniert die Objekte im Dialog
          add ("North", choiceArtikel);
          add ("Center", p);

        // WindowListener zum Schließen des Dialoges
        addWindowListener(new Dialogschliesser (this));

          // Richtet Objekte ein
        pack();
        }


    /**
     * Methode cancel ().
     * Beendet den Dialog und kehrt zurück zum Mutterfenster..
     */
    public void cancel() {
          setVisible (false);
          dispose();
          ((Window) getParent()).toFront();
          getParent().requestFocus();
          }


    /**
     * Verarbeitet ActionEvents.
     * @param e das ausgelößte Event.
     */
    public void actionPerformed (ActionEvent e) {
          // Beendet Bearbeitung, wenn Eingabe korrekt
          if (e.getActionCommand().equals ("OK"))
              {
              boolean err = false;
              int menge = 0;
              try {menge = Integer.parseInt (tf.getText ());}
              catch (NumberFormatException ex) {err = true;}

              // Wenn kein Fehler aufgetreten ist - sichern
              if ((menge > 0) && !err) {
                    // selektierten Artikel auslesen und Artikelnummer bestimmen
                    String artikel = choiceArtikel.getSelectedItem();
                    for (artikelDaten.gotoStart (); !artikelDaten.eol (); artikelDaten.gotoNext ())
                        if (artikel.equals (((Artikel) artikelDaten.getData ()).name))
                              daten.addrechnungsElement (rechnungsElemente, ((Artikel) artikelDaten.getData ()).artikel,
menge);

                    // den Dialog schliessen
                    cancel ();
                    }
              }

          // Beendet Bearbeitung sofort - ohne Änderung
          if (e.getActionCommand().equals ("Cancel"))
              cancel ();
          }
    }
```

3.6 Liste.java

```
/**
 * Klasse: Liste.
 * Speichert und Verwaltet dynamische Datenstrukturen in einer doppelt verketteten Liste.
 */
public class Liste {
    // Hilfsnodes, zur Verwaltung und Ansteuerung der Liste
    public Node start, now, end;

    /**
     * Konstruktor.
     * Hier werden die Hilfsnodes zum Verwalten der Daten initialisiert.
     */
    public Liste () {
          now = null;
          start = null;
          end = null;
          }

    /**
     * Methode: add.
     * Fügt ein neues Element in die Liste mit der größe des übergebenen Objektes ein.
     * Falls kein aktuelles Element ausgewählt (now = null), wird das Element am Ende der
     * Liste angefügt.
     * @param x - das Objekt, das in die Liste eingefügt werden soll.
     */
    public void add (Object x) {

          // Wenn kein Element initialisiert ist: neues Element erzeugen und Hilfsnodes setzen
          if (start == null)
              {
              start = new Node (x, null, null);
              end = now = start;
              }
          // Wenn schon mind. ein Element initialisiert ist: neues Element erzeugen
          else
              {
              if (now == null) now = end;
              Node T = new Node (x, now.next, now);
              now.next = T;
              now = T;
              if (now.next == null)
                    end = T;
              }
          }

    /**
     * Methode: eol.
     * Überprüft, ob das Ende der Liste erreicht wurde.
     * @return true, wenn das Ende der Liste erreicht, sonst false.
     */
    public boolean eol () {
          return now == null;
          }

    /**
     * Methode: gotoNext.
     * Setzt den Hilfspointer now auf das nächste Element in der Liste.
     */
    public void gotoNext () {
          if (now != null) now = now.next;
          }

    /**
     * Methode: gotoLast.
     * Setzt den Hilfspointer now auf das vorherige Element in der Liste.
     */
    public void gotoLast () {
          if (now != null) now = now.last;
          }

    /**
     * Methode: gotoStart.
     * Setzt den Hilfspointer now zurück auf den Start der Liste.
     */
    public void gotoStart () {
          now = start;
          }

    /**
     * Methode: getData.
     * @return Liefert das gespeicherte Objekt, der aktuellen Listenposition zurück.
     */
    public Object getData () {
          return now.data;
          }


    /**
     * Methode: destroy.
     * Setzt alle Hilfnodes auf null zurück -> Löschen der Liste.
     */
    public void destroy () {
          now = end = start = null;
          }
```

```
    /**
     * Methode: remove ().
     * Entfernt das aktuelle Element aus der Liste.
     */
    public void remove () {
          // Wenn das erste Listenelement: löschen
          if ((now == start) && (now != end)) {
              now = start = now.next;
              now.last = null;
              }

          // Wenn das letzte Listenelement: löschen
          else if ((now == end) && (now != start)) {
              end = now = now.last;
              now.next = null;
              }

          // Wenn Element in Mitte: löschen
          else if ((now != end) && (now != start)) {
              Node T = now.last;
              Node S = now.next;
              T.next = now.next;
              S.last = now.last;
              now = T;
              }

          // Wenn nur ein Element in der Liste enthalten: Liste löschen
          else if (start == end)
              destroy ();
          }


    /**
     * Überprüft, ob die Liste leer ist.
     * @return True, falls die Liste leer ist.
     */
    public boolean isEmpty () {
          if (start == null)
              return true;
          return false;
          }


    /**
     * Bestimmt die Anzahl der Einträge.
     * @return Anzahl der Einträge.
     */
    public int length () {
          int anz = 0;
          if (isEmpty ()) return anz;
          for (gotoStart (); !eol (); gotoNext ())
              anz ++;
          return anz;
          }
    }



/**
 * Klasse: Node.
 * In diesem Format werden die einzelnen Elemente der Liste abgelegt.
 */
class Node {
    /**
     * Beinhaltet die Abgelegten Daten.
     */
    Object data;

    /**
     * Zeigt auf den nächsten Listeneintrag.
     */
    Node next;

    /**
     * Zeigt auf den vorherigen Listeneintrag.
     */
    Node last;


    /**
     * Konstruktor Node.
     * Erzeugt ein neues Listenelement.
     * @param d das Objekt (Datenelement) das eingefügt werden soll.
     * @param n der Verweis auf das nächste Element.
     * @param l der Verweis auf das vorhergehende Element.
     */
    Node (Object d, Node n, Node l)
        {
        data = d;
        next = n;
        last = l;
        }
    }
```

3.7 Menue.java

```
import java.awt.*;
import java.awt.event.*;


/**
 * Erzeugt des Menü.
 * Das Menü enthält folgende Punkte:
 *  -) eine neue Rechnung eingeben.
 *  -) eine Rechnung verändern.
 *  -) eine Rechnung löschen.
 *
 *  -) das Programm beenden.
 */
class Menue extends MenuBar {
    private Menu menuRechnung;

    /**
     * Konstruktor.
     * Hier wird das Menü initialisiert.
     */
    public Menue(ActionListener listener){
          menuRechnung = new Menu ("Bearbeiten");
          addMenuItem(menuRechnung,"Rechnung Eingeben",listener);
          addMenuItem(menuRechnung,"Rechnung Ändern",listener);
          addMenuItem(menuRechnung,"Rechnung Löschen",listener);
          menuRechnung.addSeparator();
          addMenuItem(menuRechnung,"Diagramm klonen",listener);
          menuRechnung.addSeparator();
          addMenuItem(menuRechnung,"Beenden",listener);
          add(menuRechnung);
          }


    /**
     * Methode addMenuItem.
     * Reagiert auf Menüauswahlen.
     */
    void addMenuItem(Menu m, String name, ActionListener l) {
          MenuItem mi;
          mi = new MenuItem(name);
          mi.setActionCommand(name);
          mi.addActionListener(l);
          m.add(mi);
          }
   }
```

3.8 MyColors.java

```
import java.awt.*;

/**
 * Klassa MyColors.
 * Verwaltet meine Farbpalette, die für die Balken-/Kreisdiagramme notwendig ist.
 * Wenn man einmal eine Instanz erzeugt hat, erhält man mit der Methode getNext() die nächste Farbe.
 * Wenn alle Farbe verwendet wurden, wird von Neuem begonnen.
 */
class MyColors {
    private final static Color C[] = {Color.blue, Color.cyan, Color.darkGray, Color.green, Color.magenta, Color.lightGray,
                                      Color.orange, Color.pink, Color.gray, Color.red, Color.yellow};
    private int aktualColor = 0;

    /**
     * Konstruktor.
     * ohne Anweisung.
     */
    MyColors () {}


    /**
     * Methode: reset ().
     * Setzt die aktuelle Farbe auf die Erste zurück.
     */
    public void reset () {
          aktualColor = 0;
          }


    /**
     * Methode: getNext ().
     * Liefert die aktuelle Farbe und geht zur nächsten.
     * @return die aktuelle Farbe.
     */
    public Color getNext () {
          if (aktualColor >= C.length)
              reset ();
          return C [aktualColor++];
          }
    }
```

3.9 MyException.java

```
/**
 * Klasse: MyExceptions.
 * Leitet meine Exceptions weiter.
 */
class MyException extends Exception {

    /**
     * Konstruktor: MyExceptions.
     * @param String s: zu werfende Fehlermeldung.
     */
    MyException (String s)
          {
          super (s);
          }
     }
```

3.10 MyGUI.java

```
import java.awt.*;
import java.awt.event.*;
import java.util.*;

/**
 * class MyGUI.
 * Hier wird die graphische Ausgabe erzeugt.
 * Mit Hilfe des Observer Interface werden die Rechnungsfiles überwacht.
 */
public class MyGUI extends Frame implements ItemListener, Observer, ActionListener {

    private Daten daten;
    private CheckboxGroup cbg1, cbg2;
    private Checkbox artikel, kassa, balken, kreis;
    private ScrollPane scrollpane;
    private OutputCanvas output;
    private Label header;

    private final static int BL_VGAP = 10;
    private final static int BL_HGAP = 10;

    /**
     * Konstruktor.
     * Öffnet Frame und initialisiert die Anzeige.
     * @param artikelDaten - die Artikeldaten.
     * @param rechnungsDaten - die Rechnungsdaten.
     * @see Artikel
     * @see Rechnung
     * @see OutputCanvas
     * @see Fensterschliesser
     */
    MyGUI (Daten daten) {
          // Initialisiert Frame
          super ("Rechnungsbearbeitung");
          setSize (600, 400);
          setBackground (Color.lightGray);
          addWindowListener(new Fensterschliesser ());
          Menue mb = new Menue (this);
          setMenuBar (mb);

          // kopiert die Referenzen
          this.daten = daten;

          // Setzt Leyout auf BorderLayout
          setLayout (new BorderLayout(BL_HGAP, BL_VGAP));

            // Erzeugt Label:
            header = new Label ("", Label.CENTER);
            add ("North", header);

            // Erzeugt den Canvas + Scrollpane
            scrollpane = new ScrollPane ();
            output = new OutputCanvas (scrollpane, header, daten);
            scrollpane.add (output);
            add ("Center", scrollpane);
            scrollpane.doLayout ();

            // Setzt Scrollbareinheiten
            scrollpane.getHAdjustable().setUnitIncrement (40);
            scrollpane.getVAdjustable().setUnitIncrement (30);

            // Erzeugt die Radiobuttons und deren Listener
            Panel p = new Panel ();
              p.setLayout (new GridLayout(3, 2));
              cbg1 = new CheckboxGroup();
              cbg2 = new CheckboxGroup();
              p.add (new Label ("Was anzeigen?", Label.LEFT));
              p.add (new Label ("Wie anzeigen?", Label.LEFT));
              artikel = new Checkbox("Artikel", cbg1, true);
              artikel.addItemListener (this);
              p.add (artikel);
              balken = new Checkbox("Balkendiagramm", cbg2, true);
              balken.addItemListener (this);
              p.add (balken);
              kassa = new Checkbox("Kassen", cbg1, false);
              kassa.addItemListener (this);
              p.add (kassa);
              kreis = new Checkbox("Kreisdiagramm", cbg2, false);
              kreis.addItemListener (this);
              p.add (kreis);
            add ("South", p);

          // Zeigt fertigen Frame an
          setVisible (true);
          }
```

```
/**
 * Methode: itemStateChanged.
 * Wertet Veränderungen in der CheckBoxGroup aus & veranlasst ein Neuzeichnen des Canvas.
 */
public void itemStateChanged (ItemEvent e) {
    if ((e.getSource() == kassa) && output.artikel)
                output.artikel = false;
      else if ((e.getSource () == artikel) && !output.artikel)
                output.artikel = true;
    else if ((e.getSource () == kreis) && output.balken)
                output.balken = false;
      else if ((e.getSource () == balken) && !output.balken)
                output.balken = true;

      output.repaint ();
      }


/**
 * Wertet die ActionEvents des Menüs aus.
 */
public void actionPerformed (ActionEvent e) {
      String s = e.getActionCommand();

      // Dialog zum Rechnung eingeben aufrufen
      if (s.equals("Rechnung Eingeben")) {
          DialogNeu dNeu = new DialogNeu (this, "Neue Rechnung erstellen", false, daten);
          dNeu.setVisible(true);
          }

      // Dialog zum Rechnung ändern aufrufen
      else if (s.equals("Rechnung Ändern")) {
          DialogBearbeiten dEdit = new DialogBearbeiten (this, "Rechnung Bearbeiten", false, daten);
          daten.addObserver (dEdit);
          dEdit.setVisible(true);
          }

      // Dialog zum Rechnung löschen aufrufen
      else if (s.equals("Rechnung Löschen")) {
          DialogLoeschen dLöschen = new DialogLoeschen (this, "Rechnung Löschen", false, daten);
          dLöschen.setVisible(true);
          }

      // Dialog zum klonen des Fensters
      else if (s.equals("Diagramm klonen")) {
          // Startet graphische Ausgabe
          MyGUI m = new MyGUI (daten);

          // Initialisiert Observer
          daten.addObserver (m);
          }

      // Programm beenden
      else if (s.equals("Beenden")) System.exit(0);
      }


/**
 * Methode: Update.
 * Führt bei einer Veränderung des zu überwachenden Objektes die Repaintmethode des Canvas aus.
 * @param o - das Überwachte Objekt.
 * @param arg - die Veränderung.
 */
public void update (Observable o, Object arg) {
      output.repaint ();
      }
}
```

3.10.1 OutputCanvas

```
/**
 * Klasse: OutputCanvas.
 * Die Zeichenfläche. Die Abmessungen des Canvas werden durch die auszugebenden Strings definiert.
 * Der Canvas ist aber mindestens so groß, wie der geöffnete Frame.
 */
class OutputCanvas extends Canvas {
    public boolean artikel = true, balken = true;

    private ScrollPane scrollpane;
    private Label header;
    private Diagramm diagramm;
    private Daten daten;

    // Die Ausgabetexte für die Überschrift
    private final static String LBL_1 = "Verkaufte Artikel als Balkendiagramm";
    private final static String LBL_2 = "Verkaufte Artikelgruppen als Kreisdiagramm";
    private final static String LBL_3 = "Verkaufte Artikel pro Kassa als Balkendiagramm";
    private final static String LBL_4 = "Verkaufte Artikel pro Kassa als Kreisdiagramm";

    /**
     * Konstruktor.
     * Hier wird die Größe des Canvas berechnet & gesetzt.
     * @param scrollpane der ScrollPane, in dem der Canvas beinhaltet ist.
     * @param header das Label in dem die Überschrift angezeigt werden soll.
     * @param daten das Datenobjekt.
     */
    OutputCanvas (ScrollPane scrollpane, Label header, Daten daten) {
        this.scrollpane = scrollpane;
          this.header = header;
        this.daten = daten;
        setBackground (Color.white);
        Dimension groesse = scrollpane.getViewportSize ();
        setSize (groesse.width, groesse.height);
        }


    /**
     * Methode: paint.
     * Hier wird das gewünschte Diagramm ausgegeben.
     **/
    public void paint (Graphics g) {

          // Auswertung, welches Diagramm gezeichnet werden soll
          // Balkendiagramm der gekauften Artikel
          if (artikel && balken) {
              header.setText (LBL_1);
              diagramm = new Balkendiagramm (g, new ArtikelDaten (daten), scrollpane, this);
              }

          // Kreisdiagramm der gekauften Artikelgruppen
          else if (artikel && !balken) {
              header.setText (LBL_2);
              diagramm = new Kreisdiagramm (g, new GruppenDaten (daten), scrollpane, this);
              }

          // Balkendiagramm der gekauften Artikel pro Kassa
          else if (!artikel && balken) {
              header.setText (LBL_3);
              diagramm = new Balkendiagramm (g, new KassaDaten (daten), scrollpane, this);
              }

          // Balkendiagramm der gekauften Artikel pro Kassa
          else if (!artikel && !balken) {
              header.setText (LBL_4);
              diagramm = new Kreisdiagramm (g, new KassaDaten (daten), scrollpane, this);
              }
        }
    }
```

3.10.2 Fensterschliesser

```
/**
 * Klasse: Fensterschliesser.
 * Implementiert das windowClosing Event zum Schließen des Frames.
 */
class Fensterschliesser extends WindowAdapter {
    public void windowClosing (WindowEvent e) {
          System.exit (0);}
    }
```