TECHNISCHE
UNIVERSITÄT
MÜNCHEN

Astronomie und Java

Objekte der Astronomie

D. Egger

IAPG / FESG No. 13

Institut für Astronomische und Physikalische Geodäsie
Forschungseinrichtung Satellitengeodäsie

München 2001

Dieter Egger

Astronomie und Java

Objekte der Astronomie

Vorwort

Um die Astronomie oder wenigstens einen kleinen Teil davon programmiertechnisch gut erfassen zu können, ist es ausgesprochen hilfreich, die beteiligten „realen“ Objekte zu erkennen und als vereinfachte „abstrakte“ Objekte zu formulieren. Jedem Astronomie-Objekt entspricht dann idealerweise ein Programm-Objekt, das mittels Daten und Methoden das „reale“ Objekt charakterisiert.

Im vorliegenden Band werden einige wichtige Objekte aus dem Bereich der Astronomie (Geodäsie und Himmelsmechanik sind oft mit gemeint) theoretisch und programmtechnisch (in Java) aufbereitet.

Bemerkungen bitte an: dieter@astro-toolbox.com
Java-Implementierung: www.astro-toolbox.com

Dieter Egger, München, den 18.07.01

Inhalt

Objekte der Astronomie

Allgemeine Bemerkung

Viele Objekte sind von der Zeit abhängig und beinhalten diese daher implizit oder explizit. Eine *Epoche* kennzeichnet einen Zeitpunkt auf einer ausgewählten Zeitskala (UTC, UT1, TAI, TT). Fehlt die Angabe der Zeitskala, so wird meist UTC unterstellt, wenn nicht aus dem Kontext eine andere als offensichtlich hervorgeht. Objekte sind meist nur dann sinnvoll zu kombinieren, wenn sie zu äquivalenten Epochen gegeben sind.

Diagramme zur Darstellung der Objekte orientieren sich an den Richtlinien der *Unified Modelling Language* [Fowler/Scott]. Der Pfeil deutet jeweils die Generaliserungsrichtung an, d.h. er zeigt auf das allgemeinere Objekt. Man sagt: das speziellere Objekt „ist abgeleitet vom“ oder „erbt vom“ allgemeineren Objekt.

Neben der allgemeinen Beschreibung eines Objektes finden Sie meist ein UML-Diagramm und den Java-Source-Code vor. Letzterer ist ab und an gekürzt worden, um nicht stets alle Daten und Methoden anzuführen, sondern nur die für das Verständnis des Objekts wesentlichen.

Astronomische Konstante

Ein System von (aufeinander abgestimmten) astronomischen Konstanten ist notwendig, um konsistente Berechnungen durchführen zu können. Als Basis dient das System der *IAU* (International Astronomical Union) von 1976, das 1979 und 1982 ergänzt wurde (siehe Supplement to the Astronomical Almanac for 1984, Seidelmann). 1996 wurden einige Änderungen eingeführt, die verbesserte Beobachtungs- und Modellgenauigkeiten berücksichtigen (*IERS Conventions*, http://hpiers.obspm.fr/webiers/general/syframes/SY.html).

Physikalische Grundlage des *Konstantensystems* ist das *SI-System*.

```
/**
 *Aconst.java
 *
 *Often used constants in astronomy
 *
 * @author Dieter Egger
 * @version  1.3.1    2001/01/10
 */

package mie;

public class Aconst
{
  // astronomische Konstante

  public final static double SpeedOfLight       = 299792458.0;
```

```
  public final static double EarthEquatorialRadius= 6378137.0;
  public final static double EarthOblateness    = 1.0/298.2572;
  public final static double AstronomicalUnit   = 149.5978707E9;
  // new value for 1 au = 1.49597870691(30)e11

  public final static double Epsilon2000             = 84381.448;
  public final static double Planck                  = 6.62607e-34;

  public final static double JD19                    = 2415020.0;
  public final static double J1900                   = 2415020.0;
  public final static double JD20                    = 2451545.0;
  public final static double J2000                   = 2451545.0;
  public final static double JulianCentury           = 36525.0;
  public final static double JulianYear              = 365.25;
  public final static double BD19                    = 2415020.31352;
  public final static double B1900                   = 2415020.31352;
  public final static double TropicalYear       = 365.242198781;
  public final static double SiderealDay        = 86164.0989038;
  public final static double SolarDay                = 86400.0;
  public final static double TDT_TAI                 = 32.184;
  public final static double JulianEphemZero    = 2440000.0;
  public final static double JulianMJDZero      = 2400000.5;
  public final static double JulianComputerZero = 2440587.5;

  public final static double GM_Sun        =132712438000.0E9;
  public final static double GM_Earth      =      398600.5E9;
  public final static double GM_Moon       =      4902.799703E9;
  public final static double GM_EarthMoon  =      403503.2997E9;
  public final static double GM_Mercury    =       22032.08015E9;
  public final static double GM_Venus      =      324858.7609E9;
  public final static double GM_Mars       =       42828.28596E9;
  public final static double GM_Jupiter    =   126712596.6E9;
  public final static double GM_Saturn     =    37939519.15E9;
  public final static double GM_Uranus     =     5780158.449E9;
  public final static double GM_Neptune    =     6871307.756E9;
  public final static double GM_SolarSystem     = 132890534800.0E9;

  public final static double Pi = 3.14159265358979323846;
  public final static double HalfPi             = 0.5*Pi;
  public final static double TwoPi                   = 2.0*Pi;
  public final static double RadianToDegree          = 180.0/Pi;
  public final static double DegreeToRadian          = Pi/180.0;
  public final static double Radian2Degree      = 180.0/Pi;
  public final static double Degree2Radian      = Pi/180.0;
  public final static double EpsilonDouble=2.2204460492503131E-16;
}
```

Zeit und Kalender

Der *Kalender* stellt eine für praktische Zwecke geeignete Darstellung und Zählweise des *Zeitablaufs* dar. Wiederkehrende Ereignisse, wie Tag und Nacht oder Neumond und Vollmond oder Jahreszeiten definieren Zeiteinheiten, wie Tage, Wochen, Monate und Jahre und dienen dadurch der *Strukturierung der Zeit*. Verschiedene Völker haben zu verschiedenen Zeiten verschiedene *Kalender* verwendet (siehe http://www.treasure-troves.com/astro/Calendar.html).

Für astronomische Zwecke am besten geeignet ist eine kontinuierliche und gleichmässig verstreichende Zeit. Diese ist auch die Zeit eines *Inertialsystems* im NEWTONschen Sinne, die seinen *Bewegungsgleichungen* zugrundeliegt (*Ephemeridenzeit*, *dynamische Zeit*).

Zeit

Gemeint ist eine linear und stetig fortlaufende Zeit in einer Form, die für in Frage kommende Berechnungen genügend genau ist. Sie basiert auf dem sogenannten *Julianischen Datum*, das am 1.1.4713 v. Chr. mittags 12 Uhr (bezogen auf Greenwich) beginnt. Das *Julianische Jahr* dauert genau 365.25 Tage à 86400 SI-Sekunden und entspricht somit nicht der tatsächlichen Länge des *tropischen Jahres* von ca. 365.2422 Tagen, erleichtert aber die rechnerische Handhabung. Diese Zeit bildet die Grundlage für alle anderen zeitabhängigen Objekte.

```
/**
 *JulianTime.java               ---Excerpt---
 *
 * high resolution representation of continuously flowing time
 * based upon Julian day numbers
 *
 * @author Dieter Egger
 * @version  1.3.1    2001/01/10
 *
 */

package mie;

import java.util.*;

public class JulianTime extends Object     // julian daynumber separated
to int and double part
{
  protected int julianDay;
  protected double julianSecond;

  public JulianTime () { julianDay = 0; julianSecond = 0.0; }
  public JulianTime (int t, double s) { setDaySecond (t, s); }
  public JulianTime (double z) { setJulianDate (z); }

  public JulianTime (String calendardate, String calendartime)
  { julianBesselianDate (calendardate, calendartime); }
  public JulianTime (JulianTime t)
  { julianDay = t.julianDay; julianSecond = t.julianSecond; }

  public void julianBesselianDate (String cdate, String ctime)
  /*-----------------------------------------------
    Civil Date (e.g. cdate = "1990 12 24", ctime = "14 12 13.1")
    or Besselian (e.g. cdate = "B1950.0")
    or Julian Epoch (e.g. cdate = "J2000")
    ---> JulianTime
     (continously increasing daynumber,
    #0 means 1st of Jan. -4712 at noon)
    -----------------------------------------------*/
  {
    double jd;
```

```
    int year = 0;
    int day, month, hour, minute;
    double second = 0.0;

    day = month = hour = minute = 0;
    String cd = new String (cdate.toUpperCase().trim ());
    boolean bessel = cd.indexOf ('B') >= 0;
    boolean julian = cd.indexOf ('J') >= 0;

    if (bessel  || julian)
    {
      Double djd = new Double (cd.substring (1));
      jd = djd.doubleValue () - 1900.0;
      if (bessel) { jd *= Aconst.TropicalYear; jd += Aconst.BD19; }
      else        { jd *= 365.25;      jd += Aconst.JD19; }
      julianDay = (int) Math.floor (jd);
      julianSecond = (jd - julianDay)*Aconst.SolarDay;
      normalize ();
    }
    else
    {
      StringTokenizer sd = new StringTokenizer (cd, ". ymdYMD");
      if (sd.hasMoreTokens()) year  =
        (new Integer(sd.nextToken())).intValue ();
      if (sd.hasMoreTokens()) month =
        (new Integer(sd.nextToken())).intValue ();
      if (sd.hasMoreTokens()) day   =
        (new Integer(sd.nextToken())).intValue ();
      StringTokenizer st = new StringTokenizer (ctime, ": hmsHMS");
      if (st.hasMoreTokens()) hour  =
        (new Integer(st.nextToken())).intValue ();
      if (st.hasMoreTokens()) minute  =
        (new Integer(st.nextToken())).intValue ();
      if (st.hasMoreTokens()) second =
        (new Double(st.nextToken())).doubleValue ();
    julianSecond = hour*3600.0 + minute*60.0 + second;
      julianDay = julianDay (year, month, day);
      julianSecond += Aconst.SolarDay/2;
      normalize ();
    }
  }

  public int julianDay (int year, int month, int day)
  /*-----------------------------------------------
    Calendar Date ---> Julian Date
    -----------------------------------------------*/
  {
    boolean reform;
    int a, b=0, c, d;
    double x1;
    int xj;

    if (month<3) { year--; month += 12; }
    reform = (year==1582) &&
    ((month==10) && (day>=15) || (month>10)) || (year>1582);
    if (reform) { a = year/100; c = a/4; b = 2 - a + c; }
    x1 = 365.25 * year;
    if (year<0) x1 -= 0.75;
    c = (int) (x1);
    d = (int) (30.6001 * (month + 1));
```

```
    xj = c + d + day + 1720994;
    if (reform) xj += b;
    return (xj);
  }
}
```

Kalender

Zur bürgerlichen und damit traditionell vertrauten Darstellung der Zeit verhelfen die Zählweisen „Tag, Monat, Jahr“ und „Stunde, Minute, Sekunde“. Bei der letzten *Kalenderreform* im Jahre 1582 durch *Papst Gregor*, wurde der 21.März als *Frühjahrsanfang* festgelegt. Ausserdem folgte einmalig in jenem Jahr auf den 4.Oktober gleich der 15.Oktober. Das war notwendig geworden, da sich der Frühlingsanfang bereits um 10 Tage verschoben hatte. Schuld war die alte *Schaltjahr*-Regel, die *jedes* 4. Jahr einen zusätzlichen Tag einfügte. Das gregorianische Jahr umfasst im Mittel 365.2425 Tage, was durch eine modifizierte *Schaltjahrregel* erreicht wird: in jedem durch 4 teilbaren Jahr hat der Monat Februar 29 Tage (wie vorher), ausser das Jahr ist durch 100, aber nicht durch 400 teilbar. Alle anderen Jahre gestehen dem Februar nur 28 Tage zu. Der gregorianische Kalender weicht in 10000 Jahren nur um etwa 3 Tage vom tatsächlichen Sonnenlauf ab.

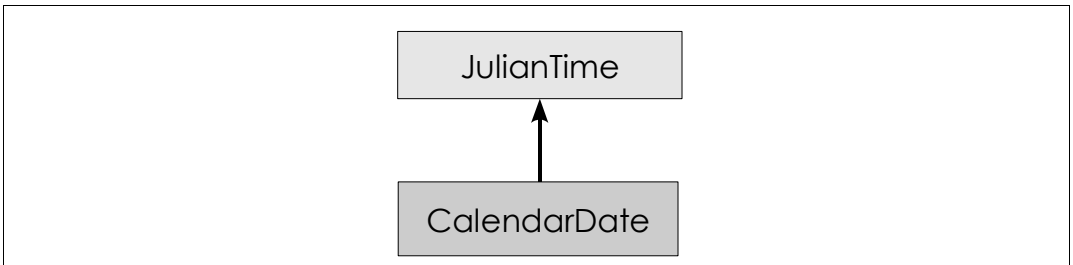

Abbildung 1: Kalender erbt Eigenschaften und Methoden von Zeit und stellt darüber hinaus die vertraute bürgerliche Darstellung bereit.

Die Umrechnung zwischen Julianischem Datum und bürgerlichem Datum [siehe Meeus] entspricht ab 15.10.1582 dem Gregorianischen Kalender. Für frühere Zeitpunkte wird der Julianische Kalender mit Jahresanfang 1. Januar unterstellt, auch wenn für sehr frühe Zeitpunkte der Julianische Kalender noch gar nicht bekannt war bzw. der *Jahresanfang* am 1. März war. Man spricht dann vom proleptischen Julianischen Kalender.

```
/**
 *CalendarDate.java              ---Excerpt---
 *
 * JulianTime plus usual civilian representation of time and date
 *
 * @see JulianTime
 *
 * @author Dieter Egger
 * @version  1.3.1    2001/01/10
 *
 */
```

```
package mie;

import java.util.*;

public class CalendarDate extends JulianTime
{
  private int day;
  private int month;
  private int year;
  private int hour;
  private int minute;
  private double second;

  public CalendarDate ()
      { super (); now (); calculateElements (); }
  public CalendarDate (double epoch)
      { super (epoch); calculateElements (); }
  public CalendarDate (JulianTime t)
      { super (t); calculateElements (); }
  public CalendarDate (String date, String time)
      { super (date, time); calculateElements (); }

  void calculateElements ()
  {
    double sgenau = 0.5e-3;
    civildate ();
    if (second + sgenau >= 60.0)
    {
      second = 0.0; minute++;
      if (minute >= 60) { minute = 0; hour++; }
    }
  }

  private void civildate ()     // Julian Date ---> Calendar Date
  {
    double f, x, jdt;
    int alfa, a, b, c, d, e, z;

    jdt = julianDate () + 0.5;
    z = (int) Math.floor (jdt);

    f = jdt - z;
    f *= 24.0;

    if (z < 2299161)
    a = z;
    else
    {
      alfa = (int)Math.floor(((z - 1867216.25)/36524.25));
      a = z + 1 + alfa - alfa/4;
    }
    b = a + 1524;
    c = (int)Math.floor((b - 122.1)/365.25);
    d = (int)Math.floor((365.25 * c));
    e = (int)Math.floor(((b - d)/30.6001));
    day = (int)(b - d - (int)Math.floor((30.6001*e)));
    if (e < 14) month = (int)(e - 1); else month = (int)(e - 13);
    if (month < 3) year = c - 4715; else year = c - 4716;
    x = f;
```

```
    hour = (int)Math.floor(x); x -= hour; x *= 60.0;
    minute = (int)Math.floor(x); x -= minute; second = x*60.0;
  }
}
```

Zeitskala

Gebräuchliche Zeitskalen sind

UTC	(Universal Time Coordinated),
UT1	(UTC mit Berücksichtigung der Erdrotation),
TAI	(Temps Atomique International) und
TT	(Terrestrial Time)

TT wurde früher als *TDT*, Terrestrisch Dynamische Zeit bzw. *ET*, Ephemeridenzeit bezeichnet.

TAI wird durch ein Ensemble von hochgenauen Atomuhren auf der Erdoberfläche realisiert und dient zur Definition der anderen Zeitskalen. Sie stellt eine Koordinatenzeitskala dar, mit Ursprung im Geozentrum und der SI-Sekundenlänge, ermittelt auf dem rotierenden Geoid, als Einheit.

Dynamische Zeitskalen (TDT, terrestrisch und TDB, baryzentrisch), die den Bewegungsgleichungen von Himmelskörpern zu Grunde liegen, weichen von den korrespondierenden *Koordinatenzeitskalen* (TCG, geozentrisch und TCB, baryzentrisch) säkular ab. Dafür bleibt die Variation von TDT (jetzt TT) und TDB periodisch mit einer Maximalamplitude von etwa 1.7 Millisekunden.

Die Koordinatenzeitskalen wurden so vereinbart, dass sie alle am 1.Januar, 1977, 0^h TAI mit der terrestrischen Zeit TT übereinstimmen.

Die Differenz *„TT - TAI“* wurde für den 1.Januar 1977, 0^h TAI zu exakt 32.184 Sekunden festgelegt und bleibt nahezu konstant, während die Differenz *„UT1 - UTC“* durch Schaltsekunden (angebracht an UTC) kleiner als 0.9 Sekunden gehalten wird. Dadurch wird auch die Differenz „UTC - TAI“ bzw. „UTC - TT“ in Sekundensprüngen verändert. „UTC - TAI“ ist dabei ganzzahlig. Die aktuellen Werte können online vom IERS (International Earth Rotation Service) via FTP (ftp://hpiers.obspm.fr/iers/bul/bulc/) oder www (www.obspm.fr) abgerufen werden.

Die Zeit (das Objekt JulianTime) wird entsprechend einer Zeitskala interpretiert. *UTC* repräsentiert beispielsweise den bürgerlichen Zeitablauf, während *TT (Terrestrial Time)* den Zeitablauf von Planeten widerspiegelt. Zeitskala ist somit auch ein Objekt Zeit.

Verzichtet man auf die Spezifikation einer Zeitskala, so wird UTC angenommen. Möchte man keine Zeitskala festlegen, so wird einfach nur das Objekt Zeit allein verwendet.

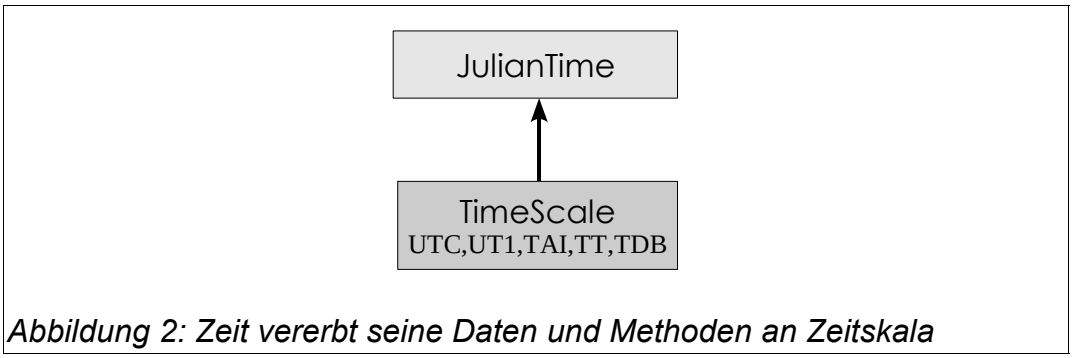

Abbildung 2: Zeit vererbt seine Daten und Methoden an Zeitskala

Ausgehend von UTC (Coordinated Universal Time), ist die Zeitdarstellung in anderen Zeitskalen möglich:

Übergang	Verfahren	Quelle
UTC –> UT1	Addition von UT1-UTC	aus Bulletin B
UT1 –> TAI	Subtraktion von UT1-TAI	aus Bulletin B
TAI –> TT	Addition von 32.184 s	IAU, 1991 [Seidelmann]
TT –> TDB	|TDB - TT| < 2 ms	[Seidelmann]

Tabelle 1: Übergänge zwischen verschiedenen Zeitskalen

Für *TDB - TT* dürfte in den meisten Fällen die Näherung

$TDB - TT = 0^s.001658 \sin g + 0^s.000014 \sin 2g$

mit

$g = 357°.53 + 0.9856003 (JD - J2000)$

ausreichend sein.

TAI - UTC ist ganzzahlig. Der aktuelle Wert ist jeweils dem *Bulletin C* zu entnehmen. Eine Übersicht über die Entwicklung der *Schaltsekunden* seit 1972 gibt Tabelle 2.

Zeitraum					TAI - UTC
1972	Jan.1	-		Jul.1	10s
	Jul.1	-	1973	Jan.1	11s
1973	Jan.1	-	1974	Jan.1	12s
1974	Jan.1	-	1975	Jan.1	13s
1975	Jan.1	-	1976	Jan.1	14s
1976	Jan.1	-	1977	Jan.1	15s
1977	Jan.1	-	1978	Jan.1	16s
1978	Jan.1	-	1979	Jan.1	17s
1979	Jan.1	-	1980	Jan.1	18s
1980	Jan.1	-	1981	Jul.1	19s
1981	Jul.1	-	1982	Jul.1	20s
1982	Jul.1	-	1983	Jul.1	21s
1983	Jul.1	-	1985	Jul.1	22s
1985	Jul.1	-	1988	Jan.1	23s
1988	Jan.1	-	1990	Jan.1	24s
1990	Jan.1	-	1991	Jan.1	25s
1991	Jan.1	-	1992	Jul.1	26s
1992	Jul.1	-	1993	Jul.1	27s
1993	Jul.1	-	1994	Jul.1	28s
1994	Jul.1	-	1996	Jan.1	29s
1996	Jan.1	-	1997	Jul.1	30s
1997	Jul.1	-	1999	Jan.1	31s
1999	Jan.1	-			32s

Tabelle 2: Schaltsekunden seit 1972, die jeweils am Ende eines Halbjahres eingefügt worden sind

Vor 1972 ist der Zusammenhang durch lineare Funktionen (seit 1961) angenähert worden. Für historische Epochen gibt es ebenfalls funktionale Näherungen [Meeus, Seidelmann]. Der Zeitraum 1620-2050 wird zusätzlich, falls keine aktuellen Informationen verfügbar sind, durch eine Approximationsfunktion abgedeckt, die aus dem Lernprozess eines neuronalen Netzes gewonnen wurde.

```
/**
 *TimeScale.java
 *
 * specify a time scale for JulianTime
 * convert time scales between (UTC, UT1, TAI, TT)
 * PolarData have to be supplied
 *
 * @see JulianTime
 * @see PolarData
 *
 * @author Dieter Egger
 * @version  1.3.1    2001/01/10
 *
```

```
 */

import java.lang.*;

import mie.*;

public class TimeScale extends JulianTime
// distinguish UTC, UT1, TAI, TT
{
  public final static int numTimeScales = 4;
  public final static int UTC = 0;
  public final static int UT1 = 1;
  public final static int TAI = 2;
  public final static int TT  = 3;
  public final static String [] sTimeScale =
                                    { "UTC", "UT1", "TAI", "TT" };

  private int timeScale = 0;

  public TimeScale () { super (); timeScale = UTC; }
  public TimeScale (int sc) { super (); setTimeScale (sc); }
  public TimeScale (String sts) { super (); setTimeScale (sts); }

  public TimeScale (JulianTime t) { super (t); timeScale = UTC; }
  public TimeScale (JulianTime t, int sc)
  { super (t); setTimeScale (sc); }
  public TimeScale (TimeScale ts)
  { super (ts); setTimeScale (ts.getTimeScale ()); }

  public void setTimeScale (int sc)
  { if (sc>=UTC && sc<=TT) timeScale = sc; else timeScale = UTC; }

  public void setTimeScale (String sts)
  {
    int i;
    for (i=0; i<numTimeScales; i++)
      if (sts.equals (sTimeScale[i])) break;
    if (i<numTimeScales) timeScale = i;
  }

  public void setTimeScale (TimeScale ts)
  {
    setJulianTime (ts);
    setTimeScale (ts.getTimeScale());
  }

  public int getTimeScale () { return timeScale; }

  public String getNameOfTimeScale ()
  { return sTimeScale [timeScale]; }

  public void setTimeScale (int sc, PolarData pd)
  {
    if (sc < UTC || sc > TT) return;
    convertTo (sc, pd);
  }

  public void setTimeScale (String sts, PolarData pd)
  { convertTo (sts, pd); }
```

```
  public boolean isUTC () { return timeScale == UTC; }
  public boolean isUT1 () { return timeScale == UT1; }
  public boolean isTAI () { return timeScale == TAI; }
  public boolean isTT  () { return timeScale == TT;  }

  public String toString ()
  { return super.toString () + ", TimeScale "
           + sTimeScale [timeScale]; }

  public void convertTo (String sts, PolarData pd)
  {
    int i;
    for (i=0; i<numTimeScales; i++)
      if (sts.equals (sTimeScale[i])) break;
    if (i<numTimeScales) convertTo (i, pd);
  }

  public JulianTime ETminusUT () {
    double w = 0.0;
    double jbd =
      (julianDate ()-Aconst.J1900)/Aconst.JulianYear+1900.0;
    System.out.println ("jbd = " + jbd);
    if (jbd > 1615.0 && jbd < 2051.0) { w = neuroETmUT (jbd); }
    else {
      double td =
        (julianDate () - Aconst.J2000)/Aconst.JulianCentury;
        // centuries since 2000.0
      w = (32.5*td + 123.5)*td + 102.3;    // Meeus S.85
    }
    System.out.println ("w = " + w);
    return new JulianTime (0, w);
  }

  /**
   * Calculation of ephemeris time
   * using result of neural network training
   * with data from table 9.1, page 86, Meeus Astronomische Algorithmen
   * accuracy is mostly better than 1 second
   * valid for 1615 till 2020 (from 2000 on extrapolated)
   * @author Dieter Egger
   * @copyright 2000 by Dieter Egger, FESG, TUM
   * @param x0 represents the year (fraction allowed)
   * @return (ET minus UT) in seconds
   */

  private double neuroETmUT (double x0) {
    x0 = (x0-(1620))/(378)*(2)+(-1);
    double y0 = ( 0.918113645472459 * ( x0) +
                 (10.9200463338378)) * (-1.03751307134623);
    double y1 = Math.sin ( -0.8737323897455 * ( y0*(11.9324746360878)) +
(-1.27478349009397)) * (-5.9046844656477);
    double y2 = Math.sin ( 78.6833362073588 * ( y0*(-
0.0880618012969841)) + (1626911737.9243)) * (-1.56701809674082);
    double y3 = Math.sin ( 1.12480718780214 * ( y0*(-0.0930436778214848)
+ y1*(0.137238006562846) + y2*(0.0471992343671378)) + (6.46970182041585))
* (6.83057210450741);
    double y4 = Math.sin ( 173.034491362782 * ( y0*(-
0.00284670895068486)) + (0.0386976775182412)) * (-1.03798517756555);
    double y5 = Math.sin ( 1039.35768060073 * ( y0*(0.0208316550416681)
+ y4*(-0.0519297046294189)) + (0.0739990033312818)) * (96.2171646078275);
```

```
    double y6 = Math.sin ( -875.328689534054 *
( y0*(0.0268429433230669)) + (-277828.099280039)) * (1.09839753496883);
    double y7 = Math.sin ( 49.7410604700863 * ( y0*(-
0.0416056243890313)) + (0.10952036977271)) * (-88.2996014394608);
    double y8 = Math.sin ( 49.5624779098763 * ( y0*(0.303086193782102) +
y2*(-0.0176252313883792) + y6*(0.00230739701450744) +
y4*(0.0111278809117251)) + (9263464873.48988)) * (-0.576943822896963);
    double y9 = ( 1.74480790584278 * ( y0*(-0.0111003163026465) +
y6*(0.0281774593100627) + y8*(0.0877129353891322) + y5*(-
0.00309276380584962) + y2*(0.046134210535281) + y7*(0.005031055055985 14)
+ y3*(-0.0284262625879017) + y4*(-0.0291537297839362)) +
(0.0386507804619302)) * (-1.04347352056399);
    double z0 = (y9-(-1))/(2)*(130.4)+(-6.4);
    return z0;
  }

  public void convertTo (int newScale, PolarData pd)
  {
    if (newScale != timeScale)
        // all the necessary time transformations
    {
      if (isUTC ())
      {
        plus (new JulianTime (0, pd.ut1_utc)); // now UT1
        if (newScale == TAI || newScale == TT)
        {
          minus (new JulianTime (0, pd.ut1_tai));    // now TAI
          if (newScale == TT)
              plus (new JulianTime (0, Aconst.TDT_TAI));
        }
      }
      else
      if (isUT1 ())
      {
        if (newScale == TAI || newScale == TT)
        {
          minus (new JulianTime (0, pd.ut1_tai));    // now TAI
          if (newScale == TT)
              plus (new JulianTime (0, Aconst.TDT_TAI));
        }
        else minus (new JulianTime (0, pd.ut1_utc));
      }
      else
      if (isTAI ())
      {
        if (newScale == TT)
            plus (new JulianTime (0, Aconst.TDT_TAI));
        else
        {
          plus (new JulianTime (0, pd.ut1_tai)); // now UT1
          if (newScale == UTC)
              minus (new JulianTime (0, pd.ut1_utc));
        }
      }
      else
      if (isTT ())
      {
        minus (new JulianTime (0, Aconst.TDT_TAI));  // now TAI
        if (newScale == UT1 || newScale == UTC)
        {
```

```
            plus (new JulianTime (0, pd.ut1_tai)); // now UT1
            if (newScale == UTC)
                minus (new JulianTime (0, pd.ut1_utc));
          }
        }

        timeScale = newScale;
      }
    }
}
```

Sternzeit

Die Lage des *Frühlingspunktes (Schnittpunkt Ekliptik-Äquator, in dem die Sonne am Frühlingsanfang steht)* auf dem Äquator, ausgehend vom *Meridian von Greenwich* nach Westen als Winkel gezählt *(15 Grad entsprechen einer Stunde)*, ist die Sternzeit von Greenwich. Lokale Sternzeiten starten beim lokalen Meridian (Längenkreis, der durch den Südpunkt verläuft).

Man unterscheidet wahre (auch scheinbare genannt) und mittlere Sternzeit. Die tatsächliche Lage des Frühlingspunktes wird mit wahrer Sternzeit, die mittlere Lage (gleichmässig rotierende Erde angenommen) als mittlere Sternzeit bezeichnet.

Der wahre *Sternzeit-Winkel* spiegelt exakt die *Erddrehung* wider. Somit entsprechen 360 Grad Erddrehung genau 24 Stunden wahrer Sternzeit. Nach 24 Stunden wahrer Sternzeit ist ein Stern wieder in derselben Himmelsrichtung zu sehen (falls sich der Stern nicht merklich bewegt, was für die meisten Sterne der Fall ist). Die wahre Sternzeit verläuft nicht gleichmässig. Wegen komplexer Abhängigkeiten von aktuellen Masseverteilungen rotiert die Erde verschieden schnell und verursacht dadurch Schwankungen in der Grössenordnung von Millisekunden.

Der Zusammenhang zwischen dem Objekt zur Darstellung der Zeit in der Zeitskala *UT1* (Daten: t_{UT1}) und dem Objekt zur Darstellung der *(mittleren) Sternzeit von Greenwich* (Daten: θ_{mean} , bzw. *GMST*) ist gegeben durch:

θ_{mean} (0^h UT1) =

$24110^s.54841 + 8640184^s.812866\ T_u + 0^s.093104\ T_u^2 - 6.2x10^{-6}\ T_u^3$

in Sekunden mit

T_u = (t_{UT1} – *J2000*)/36525, julianische Jahrhunderte seit 1.1.2000, 12^h UT1

Für andere Zeitpunkte als 0^h UT1 ist die Anzahl der Sekunden des Tages, multipliziert mit dem Sternzeitfaktor (ca. 366.25/365.25), zu addieren.

Wird t_{UT1} von vorne herein auch für andere Zeitpunkte als 0^h UT1 zugelassen, so kann θ_{mean} (in Stunden) auch wie folgt berechnet werden:

θ_{mean} =

$[67310^s.54841 + T_u\ (8640184^s.812866 + T_u\ (\ 0^s.093104\ - 0^s.0000062\ T_u\ ))] / 3600^{s/h}$

$+ 876600^h.0\ T_u$

Das Objekt *Sternzeit* ist sehr eng mit dem Objekt *Zeit* verknüpft, wobei das Objekt *Zeit* das elementare ist. Folglich signalisieren *Zeit* und *Sternzeit* eine *Objekthierarchie*. *Sternzeit* erbt von *Zeit*, ergänzt diese aber durch den obigen Zusammenhang. Nicht zu vergessen ist, dass *Sternzeit* seiner geerbten Eigenschaft *Zeit* stillschweigend die Zeitskala UT1 unterlegt.

Obwohl das rechte Diagramm den korrekten Sachverhalt wiedergibt, könnte

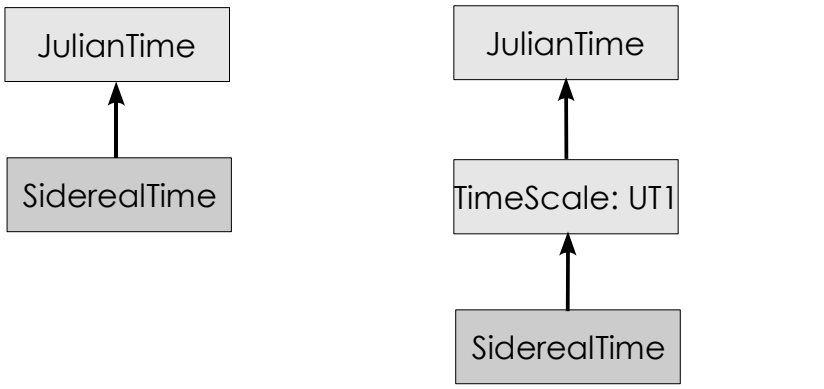

Abbildung 3: Sternzeit erbt direkt von Zeit (links), oder indirekt über Zeitskala:UT1 von Zeit (rechts). Obwohl der Zusammenhang auf der rechten Seite den „Tatsachen" entspricht, wird der einfachere auf der linken Seite gewählt (siehe Text)

die Realisierung Probleme bereiten, da zur Bestimmung von UT1 Erdrotationsparameter benötigt werden, die unter Umständen gar nicht oder nicht für den gewünschten Zeitpunkt vorhanden sind. Korrektheit könnte mithin nur suggeriert werden.

Aus diesem Grund erscheint es günstiger, die links angedeutete Hierarchie zu wählen und erst für eine konkrete Anwendung die Objekte *Zeit*, *Zeitskala* und *Sternzeit* als eigenständige Tools zu verknüpfen. Bei *Zeitskala* wäre dann UT1 explizit auszuwählen. Für Näherungsrechnungen könnte auch ganz auf *Zeitskala* verzichtet werden.

Zur Berechnung der *wahren Sternzeit* von Greenwich (θ_{true} , bzw. *GAST*) werden zusätzlich die Objekte Ekliptik (Ekliptikschiefe ϵ) und Nutation (Nutation in Länge $\Delta\psi$) benötigt:

$\theta_{true} = \theta_{mean} + \Delta\psi \cos \epsilon$

Für hochpräzise Berechnungen aufgrund der *IERS Conventions* ist seit dem 1.1.1997 zu θ_{true} noch der Term

0".00264 sin Ω + 0".000063 sin 2Ω

aufzuaddieren, mit Ω= mittlere Länge des Mondbahnknotens (siehe Sourcecode des Objektes Nutation).

```
/**
 *SiderealTime.java
 *
 * JulianTime plus corresponding mean sidereal time of Greenwich
 *
```

```
 * @see JulianTime
 *
 * @author Dieter Egger
 * @version  1.3.1    2001/01/10
 *
 */

package mie;

public class SiderealTime extends JulianTime
{
  private double theta;
  private double dtheta;


  public SiderealTime () { now (); calculateElements (); }
  public SiderealTime (double epoch)
      { super (epoch); calculateElements (); }
  public SiderealTime (JulianTime t)
      { super (t); calculateElements (); }

  public SiderealTime (SiderealTime st)
  {
    super (st);
    theta   = st.theta;
    dtheta = st.dtheta;
  }

  public void calculateElements ()
  {
    theta = b_Theta ();
    dtheta = b_DTheta ();
  }

  public JulianTime getSiderealTime () {
    return new
    JulianTime(0, theta*Aconst.Radian2Degree/15.0*3600.0-43200.0);
  }

  public void setJulianDate (double epoch)
  {
    super.setJulianDate (epoch);
    calculateElements ();
  }

  public void setJulianTime (JulianTime t)
  {
    super.setJulianTime (t);
    calculateElements ();
  }

  public void plus (JulianTime t)
  {
    super.plus (t);
    calculateElements ();
  }

  public void minus (JulianTime t)
  {
    super.minus (t);
```

```
    calculateElements ();
  }

  public double gmst       () { return theta; }
  public double gmstRate   () { return dtheta; }

  public double getTheta   () { return theta; }
  public double getDtheta  () { return dtheta; }

  private double b_Theta ()
  {
    double th, T = (julianDate () - Aconst.J2000)
                     / Aconst.JulianCentury;

    th = MyLib.horner
           (T, 67310.54841, 8640184.812866, 0.093104, -0.0000062);
    return ( MyLib.mod24
             (th/3600.0 + 876600.0*T) *15.0*Aconst.DegreeToRadian);
  }

  private double b_DTheta ()
  {
    double th, T = (julianDate () - Aconst.J2000)
                     / Aconst.JulianCentury;

    th =
    MyLib.horner (T, 8640184.812866, 0.186208, -0.0000186, 0.0);
    return ( (th/3600.0 + 876600.0) *15.0*Aconst.DegreeToRadian/
           (Aconst.JulianCentury*Aconst.SolarDay) );
  }

  public String toString ()
  {
    return super.toString()+
      "\t SiderealTime theta "+
      MyLib.degree2Dms (theta*Aconst.Radian2Degree/15.0,3);
  }
}
```

Sonnenzeit

Im Gegensatz zur Sternzeit, die sich an den Sternen orientiert, verfolgt die Sonnenzeit den Zweck, mit der Sonne, also Tag und Nacht, in Einklang zu bleiben. Mittags erreicht die Sonne ihren höchsten Stand und dann ist es traditionsgemäss 12 Uhr (lokale *wahre Sonnenzeit*, *wahrer Mittag*). Dies gilt für jeden Tag. Von einem wahren *Mittag* zum nächsten verstreichen 24 Stunden wahre Sonnenzeit. Auch die wahre Sonnenzeit verfliesst nicht gleichmässig. Zu den Erdrotationsschwankungen kommt noch die verschieden schnelle Umlaufbewegung der Erde um die Sonne hinzu. Um dennoch eine gleichmässig verstreichende Sonnenzeit zu realisieren, wird eine gedachte, *fiktive Sonne* eingesetzt, die in der Äquatorebene gleichmässig um die Erde läuft.

24 Stunden Sternzeit entsprechen nicht genau 24 Stunden Sonnenzeit. Während der 24 Stunden Sternzeit bewegt sich nämlich die Erde um ca. 1/366 auf ihrer Umlaufbahn weiter, so dass sie sich noch ein kleines Stückchen (fast ein Grad in knapp 4 Minuten) weiter drehen muss, damit die Sonne wieder in derselben Himmelsrichtung erscheint. Wegen der vorhandenen Bahnexzentrizität *(nicht kreisförmige Bahn)* und der Projektion der Bahn auf den Äquator kommt noch die sogenannte *Zeitgleichung* (wahre Zeit minus mittlere Zeit) ins Spiel. Sie kann bis zu achtzehn Minuten Anfang November und bis zu –14 Minuten Anfang Februar betragen [dtv-Atlas zur Astronomie].

Ein Sonnentag dauert 24 Stunden Sonnenzeit, ein Sterntag 24 Stunden Sternzeit (im Mittel). Man beachte, dass daher ursprünglich Sonnenzeitsekunde (im Mittel der SI-Sekunde entsprechend) und Sternzeitsekunde verschieden lang sind. Heutzutage wird aber einheitlich die SI-Sekunde verwendet, so dass zwar der Sonnentag davon ca. 86400 umfasst, der Sterntag aber nur etwa 86164.

Die Sonnenzeit wird derzeit nicht als eigenes Objekt unterstützt. Es wird vielmehr jeweils bei Zeitangaben UTC als Zonenzeit von Greenwich unterstellt. Lokale Rechnerzeiten werden aufgrund der internen TimeZone-Variablen nach UTC umgerechnet.

Ekliptik

Die fiktive Ebene, die von der Umlaufbewegung der Erde um die Sonne aufgespannt wird, nennt man Ekliptikebene. Sie ist gegenüber der Äquatorebene der Erde geneigt. Diesem Umstand verdanken wir unsere *Jahreszeiten*. Die Ekliptikebene schneidet die Himmelskugel in der Ekliptik, der scheinbaren Bahn der Sonne am Firmament.

Sie wird durch die Neigung zur Äquatorebene der Erde charakterisiert. Ihr Mittelwert ϵ_0, die *Schiefe der Ekliptik*, hängt geringfügig von der Zeit ab:

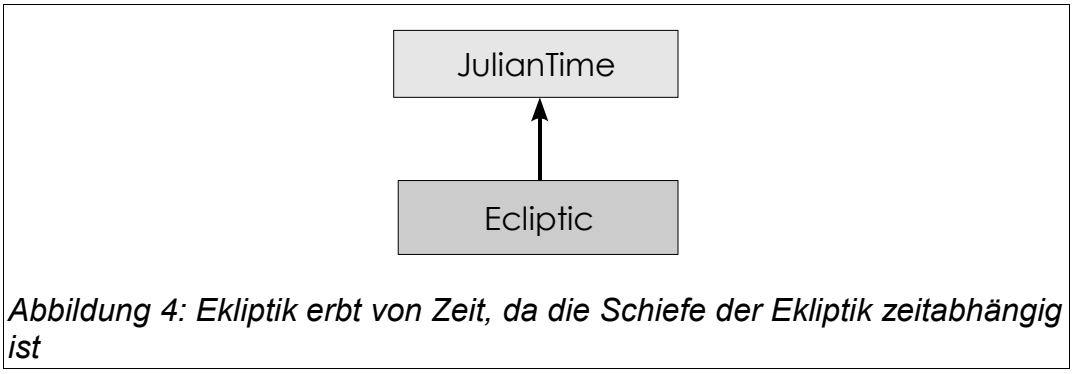

Abbildung 4: Ekliptik erbt von Zeit, da die Schiefe der Ekliptik zeitabhängig ist

ϵ_0 = 23°26'21“.448 - 46“.815 T - 0“.00059 T^2 + 0“.001813 T^3

mit

T = (JD - 2 451 545.0) / 36525

Zeitspanne seit J2000 in *julianischen Jahrhunderten* (dynamische Zeitskala, also TT oder TDB) .

Die *wahre Ekliptikschiefe* ϵ benötigt zusätzlich das Objekt *Nutation* (Nutation in Schiefe $\Delta\epsilon$)

$\epsilon = \epsilon_0 + \Delta\epsilon$

```
/**
 *Ecliptic.java
 *
 * JulianTime plus corresponding mean obliquity of ecliptic
 *
 * @see JulianTime
 *
 * @author Dieter Egger
 * @version  1.3.1    2001/01/10
 *
 */

package mie;

public class Ecliptic extends JulianTime   // is Object
{
  private double eps;
  private double deps;

  public Ecliptic () { now (); calculateEpsDeps (); }
  public Ecliptic (double epoch)
    { super (epoch); calculateEpsDeps (); }
  public Ecliptic (JulianTime t)
    { super (t); calculateEpsDeps (); }
  public Ecliptic (Ecliptic e)
    { super (e); eps = e.eps; deps = e.deps; }

  public void setJulianDate (double epoch)
    { super.setJulianDate (epoch); calculateEpsDeps (); }
  public void setJulianTime (JulianTime t)
    { super.setJulianTime (t); calculateEpsDeps (); }

  private void calculateEpsDeps ()
    { eps = obliquity (); deps = dObliquity (); }

  private double obliquity  ()
  {
    double T = (julianDate () - Aconst.J2000)
                / Aconst.JulianCentury;
    return
    ( MyLib.horner
      (T, Aconst.Epsilon2000, -46.815, -0.00059, 0.001813)
      / 3600.0 * Aconst.DegreeToRadian );
  }

  private double dObliquity ()
  {
    double T = (julianDate () - Aconst.J2000)
               / Aconst.JulianCentury;
    return
    ( MyLib.horner (T, -46.815, -0.00118, 0.005439, 0.0)
      / (Aconst.JulianCentury*Aconst.SolarDay*3600.0)
```

```
      *Aconst.DegreeToRadian );
  }

  public void plus (JulianTime t)
  {
    super.plus (t);
    calculateEpsDeps ();
  }

  public void minus (JulianTime t)
  {
    super.minus (t);
    calculateEpsDeps ();
  }

  public double getEps     () { return eps; }
  public double getDeps    () { return deps; }

  public String toString ()
  {
    return super.toString()+"\t Ecliptic eps = "+
    MyLib.degree2Dms (eps*Aconst.Radian2Degree,3);
  }
}
```

Nutation

Kurzperiodische Veränderungen der Rotationsachse der Erde und damit der Äquatorebene werden als Nutationseffekte beschrieben. Mond, Sonne und Planeten zeichnen hierfür verantwortlich. Die Hauptperiode liegt bei 18.6 Jahren, der Periode der Mondknotendrehung.

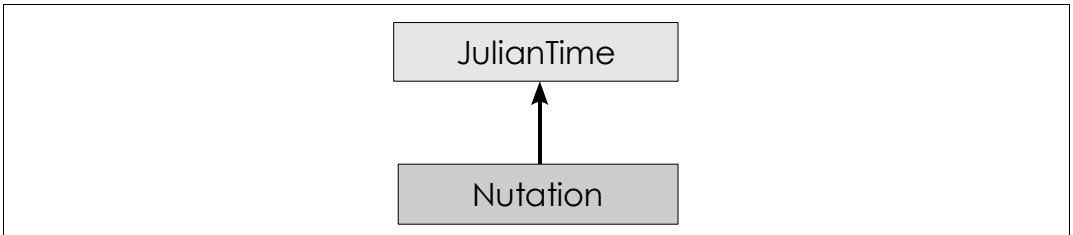

Abbildung 5: Nutation wird von Zeit abgeleitet

Nutation in Länge $\Delta\psi$ und *Nutation in Schiefe* $\Delta\epsilon$ sind dabei die erforderlichen Kenngrössen. Sie hängen über die zusätzlich zu berücksichtigenden Objekte Sonne und Mond in etwas komplizierter Form von der Zeit ab. Dennoch wird das Objekt Nutation direkt vom Objekt Zeit abgeleitet. Die notwendigen Daten von Mond und Sonne werden als Funktionen der Zeit eingeführt und nicht den Objekten Mond oder Sonne entnommen, um mit der traditionellen Nutationstheorie in Einklang zu bleiben.

Diese berechnet die *Nutationselemente* (siehe Abbildung 6) aus umfangreichen Reihenentwicklungen:

$$\Delta\psi = \sum_{i=1}^{n} S_i \sin A_i \quad , \quad \Delta\epsilon = \sum_{i=1}^{n} C_i \cos A_i \quad , \quad A_i = a_i l + b_i l' + c_i F + d_i D + e_i \Omega$$

Die mit dem Index i behafteten Koeffizienten sind Bestandteil einer *Nutationstheorie*. Beispielsweise enthält die IAU (1980) Reihe 106 Terme,

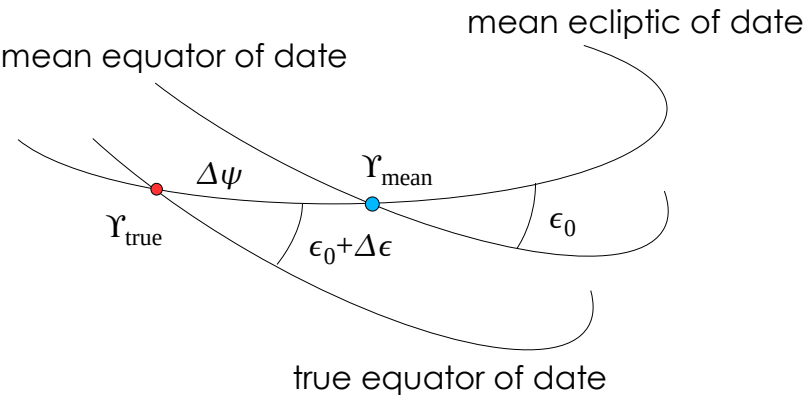

Abbildung 6: Bedeutung der Nutationselemente, die den Übergang zwischen mittlerem und wahrem Äquator vermitteln

die einer Tabelle [Seidelmann, S. 112,113] entnommen werden können. In dieser Arbeit findet eine gekürzte Fassung Eingang, die nur Terme > 0.001" berücksichtigt (vergleiche Objekt *Nutation*, Methode *calculateElements ()*).
Die restlichen Grössen definieren Objekteigenschaften von Mond und Sonne:

l	Mond, mittlere Anomalie
l'	Sonne, mittlere Anomalie
F	Mond, mittlere Länge, gezählt ab aufsteigendem Knoten
D	Mond - Sonne, mittlere Elongation
Ω	Mond, mittlere Länge des aufsteigenden Knotens

Sie werden durch kubische Funktionen der Zeitdifferenz T (jul.Jahrhunderte seit J2000, dynamische Zeitskala z.B. TDB) approximiert [Seidelmann, S.114]. Zur Berechnung siehe Objekt *Nutation*, Methode *NutVor ()*.

```
/**
 *Nutation.java
 *
 * JulianTime plus corresponding nutation elements
 * based upon a series approximation
 *
 * @see JulianTime
 *
 * @author Dieter Egger
 * @version  1.3.1    2001/01/10
 *
 */

package mie;
```

```
public class Nutation extends JulianTime
{
  private double w;
  private double s;
  private double f;
  private double d;
  private double o;

  private double dpsi;
  private double deps;

  public Nutation () { now (); calculateElements (); }
  public Nutation (double epoch) {
    super (epoch); calculateElements (); }
  public Nutation (JulianTime t){ super (t);calculateElements(); }

  public void setJulianDate (double epoch) {
    super.setJulianDate (epoch);
    calculateElements ();
  }

  public void setJulianTime (JulianTime t)
  {
    super.setJulianTime (t);
    calculateElements ();
  }

  public void plus (JulianTime t)
  {
    super.plus (t);
    calculateElements ();
  }

  public void minus (JulianTime t)
  {
    super.minus (t);
    calculateElements ();
  }

  public double deltaPsi () { return dpsi; }
  public double deltaEps () { return deps; }

  public double getDpsi () { return dpsi; }
  public double getDeps () { return deps; }

  private void NutVor ()
  /*-------------------------------------------------
    important values for nutation evaluation:
    w - mean anomaly of moon
    s - mean anomaly of sun
    f - mean longitude of moon minus o
    d - mean angular distance of the moon from the sun
    o - mean longitude of moon's ascending node
    -------------------------------------------------*/
  {
    double T =
      (julianDate () - Aconst.J2000)/ Aconst.JulianCentury;

     w =  MyLib.horner (T, 485866.733, 715922.633, 31.31, 0.064)
```

```
          /3600.0 + 1325.0*T*360.0;
    w =  MyLib.mod360 (w)*Aconst.DegreeToRadian;
    s =  MyLib.horner
          (T, 1287099.804, 1292581.224, -0.577, -0.012)
          /3600.0 + 99.0*T*360.0;
    s =  MyLib.mod360 (s)*Aconst.DegreeToRadian;
    f =  MyLib.horner (T, 335778.877, 295263.137, -13.257, 0.011)
          /3600.0 + 1342.0*T*360.0;
    f =  MyLib.mod360 (f)*Aconst.DegreeToRadian;
    d =  MyLib.horner
          (T, 1072261.307, 1105601.328, -6.891, 0.019)
          /3600.0 + 1236.0*T*360.0;
    d =  MyLib.mod360 (d)*Aconst.DegreeToRadian;
    o =  MyLib.horner (T, 450160.28, -482890.539, 7.455, 0.008)
          /3600.0 - 5.0*T*360.0;
    o =  MyLib.mod360 (o)*Aconst.DegreeToRadian;
  }

  private void calculateElements ()
  /*-------------------------------------------------
    series expansion of nutation elements
    T    - jul. centuries since J2000
    deps - nutation in obliquity
    dpsi - Nutation in longitude
    (WAHR's model "1980 IAU Theory of Nutation")
    only terms > 1 milliArcSeconds being considered.
    -------------------------------------------------*/
  {
    double T =
      (julianDate () - Aconst.J2000)/ Aconst.JulianCentury;

    NutVor ();

    dpsi= (-171996.0 - 174.2*T)*Math.sin(                  o);
    dpsi+=(   2062.0 +   0.2*T)*Math.sin(                2*o);
    dpsi+=( -13187.0 -   1.6*T)*Math.sin(        2*f-2*d+2*o);
    dpsi+=(   1426.0 -   3.4*T)*Math.sin(       s            );
    dpsi+=(   -517.0 +   1.2*T)*Math.sin(       s+2*f-2*d+2*o);
    dpsi+=(    217.0 -   0.5*T)*Math.sin(      -s+2*f-2*d+2*o);
    dpsi+=(    129.0 +   0.1*T)*Math.sin(         2*f-2*d  +o);
    dpsi+=(  -2274.0 -   0.2*T)*Math.sin(         2*f    +2*o);
    dpsi+=(    712.0 +   0.1*T)*Math.sin(   w               );
    dpsi+=(   -386.0 -   0.4*T)*Math.sin(         2*f      +o);
    dpsi+=(   -301.0          )*Math.sin(   w    +2*f    +2*o);
    dpsi+=(   -158.0          )*Math.sin(   w        -2*d    );
    dpsi+=(    123.0          )*Math.sin(  -w    +2*f    +2*o);
    dpsi+=(     63.0          )*Math.sin(             2*d    );
    dpsi+=(     63.0 +   0.1*T)*Math.sin(   w              +o);
    dpsi+=(    -58.0 -   0.1*T)*Math.sin(  -w              +o);
    dpsi+=(    -59.0          )*Math.sin(  -w    +2*f+2*d+2*o);
    dpsi+=(    -51.0          )*Math.sin(   w    +2*f      +o);
    dpsi+=(     46.0          )*Math.sin(-2*w    +2*f      +o);
    dpsi+=(     48.0          )*Math.sin( 2*w        -2*d    );
    dpsi+=(    -38.0          )*Math.sin(          2*f+2*d+2*o);
    dpsi+=(     11.0          )*Math.sin( 2*w    -2*f        );
    dpsi+=(    -22.0          )*Math.sin(          2*f-2*d    );
    dpsi+=(     17.0 -   0.1*T)*Math.sin(     2*s            );
    dpsi+=(    -15.0          )*Math.sin(       s          +o);
    dpsi+=(    -16.0 +   0.1*T)*Math.sin(     2*s+2*f-2*d+2*o);
    dpsi+=(    -12.0          )*Math.sin(      -s          +o);
```

```
    dpsi+=(     29.0          )*Math.sin( 2*w                 );
    dpsi+=(     29.0          )*Math.sin(   w    +2*f-2*d+2*o);
    dpsi+=(    -31.0          )*Math.sin( 2*w    +2*f    +2*o);
    dpsi+=(     26.0          )*Math.sin(         2*f         );
    dpsi+=(     21.0          )*Math.sin(  -w    +2*f      +o);
    dpsi+=(     16.0          )*Math.sin(  -w        +2*d  +o);
    dpsi+=(    -13.0          )*Math.sin(   w        -2*d  +o);
    dpsi+=(    -10.0          )*Math.sin(  -w    +2*f+2*d  +o);

    dpsi *= 0.0001/3600.0*Aconst.DegreeToRadian;

    deps= (  92025.0 +   8.9*T)*Math.cos(                   o);
    deps+=(   -895.0 +   0.5*T)*Math.cos(                 2*o);
    deps+=(   5736.0 -   3.1*T)*Math.cos(         2*f-2*d+2*o);
    deps+=(     54.0 -   0.1*T)*Math.cos(       s             );
    deps+=(    224.0 -   0.6*T)*Math.cos(       s+2*f-2*d+2*o);
    deps+=(    -95.0 +   0.3*T)*Math.cos(      -s+2*f-2*d+2*o);
    deps+=(    -70.0          )*Math.cos(         2*f-2*d  +o);
    deps+=(    977.0 -   0.5*T)*Math.cos(         2*f    +2*o);
    deps+=(    200.0          )*Math.cos(         2*f      +o);
    deps+=(    129.0 -   0.1*T)*Math.cos(   w    +2*f    +2*o);
    deps+=(    -53.0          )*Math.cos(  -w    +2*f    +2*o);
    deps+=(    -24.0          )*Math.cos(-2*w    +2*f      +o);
    deps+=(    -33.0          )*Math.cos(   w              +o);
    deps+=(     32.0          )*Math.cos(  -w              +o);
    deps+=(     26.0          )*Math.cos(  -w    +2*f+2*d+2*o);
    deps+=(     27.0          )*Math.cos(   w    +2*f      +o);
    deps+=(     16.0          )*Math.cos(         2*f+2*d+2*o);
    deps+=(    -12.0          )*Math.cos(   w    +2*f-2*d+2*o);
    deps+=(     13.0          )*Math.cos( 2*w    +2*f    +2*o);
    deps+=(    -10.0          )*Math.cos(  -w    +2*f      +o);

    deps *= 0.0001/3600.0*Aconst.DegreeToRadian;
  }

  public String toString ()
  {
    return super.toString()+"\t Nutation dpsi = "+
      MyLib.display (dpsi*Aconst.Radian2Degree*3600.0,3)+
      "\t deps = "+
      MyLib.display (deps*Aconst.Radian2Degree*3600.0,3);
  }
}
```

Präzession

Langperiodische Variationen der Erdrotationsachse werden als Präzession bezeichnet (analog zur Präzession eines Kreisels). Etwa alle 26 000 Jahre erfolgt ein kompletter Umlauf des Rotationspols der Erde um eine mittlere Achse, die senkrecht auf der Ekliptikebene steht.

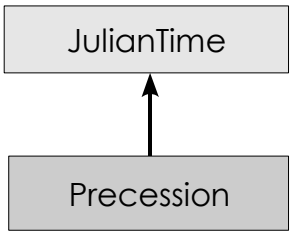

Abbildung 7: Die Präzession ist zeitabhängig, daher wird das Objekt Präzession vom Objekt Zeit abgeleitet

Ihre Kenndaten bilden die drei Grössen ζ_A, z_A und θ_A, die Winkelunterschiede zwischen äquatorialen, raumfesten Bezugssystemen zu verschiedenen Epochen darstellen (siehe Abbildung 8, ϒ=Frühlingspunkt, Richtung der x-Achse). Die Übergangsepoche ist J2000, von der aus die Zeitintervalle zu den beiden Epochen der Bezugssysteme gezählt werden. Zur Berechnung siehe [Seidelmann, S.104].

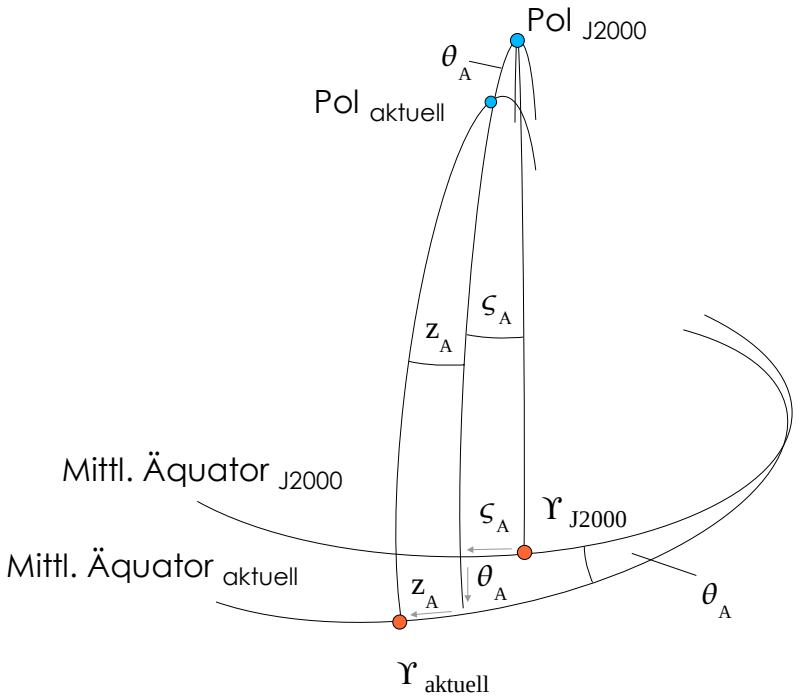

Abbildung 8: Bedeutung der Präzessionselemente, die den Zusammenhang zwischen Äquatorsystemen zu verschiedenen Epochen herstellen

Falls man vom *Bezugssystem zur Epoche J2000* ausgeht, vereinfachen sich die Formeln für die *Präzessionselemente* zu (Werte in Bogensekunden):

ζ_A = ((0.017998*T+0.30188)*T+2306.2181)*T

z_A = ((0.018203*T+1.09468)*T+2306.2181)*T

θ_A = ((-0.041833*T-0.42665)*T+2004.3109)*T

mit

T = (JD - J2000) / 36525

dynamische Zeitdifferenz (TDB oder TT).

```
/**
 *Precession.java
 *
 * JulianTime plus corresponding precession elements
 * based upon the IAU 1984 precession formulae
 *
 * @see JulianTime
 *
 * @author Dieter Egger
 * @version  1.3.1    2001/01/10
 *
 */

package mie;

public class Precession extends JulianTime // is Object
{
  private double z0;
  private double z;
  private double th;

  public Precession () { now (); calculateElements (); }
  public Precession (double epoch) {
    super (epoch); calculateElements (); }
  public Precession (JulianTime t) {
    super (t); calculateElements (); }

  public void setJulianDate (double epoch)
  {
    super.setJulianDate (epoch);
    calculateElements ();
  }

  public void setJulianTime (JulianTime t)
  {
    super.setJulianTime (t);
    calculateElements ();
  }

  public void plus (JulianTime t)
  {
    super.plus (t);
    calculateElements ();
  }

  public void minus (JulianTime t)
  {
    super.minus (t);
    calculateElements ();
  }

  public double zeta0 () { return z0; }
  public double zet   () { return z;  }
  public double theta () { return th; }
```

```
  public double getZ0 () { return z0; }
  public double getZ  () { return z;  }
  public double getTh () { return th; }


  private void calculateElements ()
  /*-------------------------------------------------------
    Berechnen der Praezessionselemente basierend auf J2000
    mit dem Startdatum J2000
    -------------------------------------------------------*/
  {
    double u =
      (julianDate () - Aconst.J2000)/ Aconst.JulianCentury;
    z0 = (( 0.017998*u+0.30188)*u+2306.2181)*u/3600.0
          *Aconst.DegreeToRadian;
    z  = (( 0.018203*u+1.09468)*u+2306.2181)*u/3600.0
          *Aconst.DegreeToRadian;
    th = ((-0.041833*u-0.42665)*u+2004.3109)*u/3600.0
          *Aconst.DegreeToRadian;
  }

  public String toString ()
  {
    return super.toString()+
      "\t Precession zeta0 "+
      MyLib.degree2Dms (z0*Aconst.Radian2Degree, 3)+
      "\t z "+MyLib.degree2Dms (z*Aconst.Radian2Degree, 3)+
      "\t theta "+MyLib.degree2Dms (th*Aconst.Radian2Degree, 3);
  }
}
```

Polbewegung

Die Rotation der Erde ist variabel, sowohl was die Achse als auch die Geschwindigkeit anbetrifft. Schuld daran sind Massenverlagerungen im Erdinnern, auf der Erdoberfläche, in den Ozeanen und in der Atmosphäre.

Die momentane Lage eines Beobachters in einem raumfesten Bezugssystem *(das die Erddrehung nicht mitmacht)* wird signifikant von der aktuellen Lage des *Rotationspols* beeinflusst.

Aktuelle *Erdrotationsparameter* müssen geschätzt werden, da sie immer erst im Anschluss an aktuelle Beobachtungen genau bestimmbar sind. Vor allem der aktuelle Wert von *UT1 - UTC* ist wichtig, da Beobachtungszeitpunkte in UTC und die aktuelle Lage der Erde im Raum durch *UT1* (als Eingangsparameter für die Sternzeitberechnung) festgelegt sind. Ignoriert man diesen zwar kleinen Unterschied von definitionsgemäss weniger als 0.9 Zeitsekunden (meist weniger als 0.5 s), so führt man doch Richtungsfehler bis zu 7.5 oder gar (im Extremfall) bis zu 13.5 Bogensekunden ein. Die Lage des aktuellen Rotationspols, beschrieben durch x- und y-Polkoordinaten, wirkt sich dagegen mit höchstens 0.6 Bogensekunden aus.

Die *Poldaten* sind dem *Bulletin B* (ftp://hpiers.obspm.fr/iers/bul/bulb/) zu entnehmen, das monatlich neu aufgelegt wird und endgültige Werte für den jeweils zurückliegenden und prädizierte Werte für den aktuellen und kurz bevorstehenden Zeitraum bereitstellt.

Die Lage des Rotationspols hängt auf recht komplexe Art und Weise von der Zeit ab. Mangels der genauen Kenntnis des Erdaufbaus können keine genauen Zeitfunktionen der Orientierung und der *Rotationsgeschwindigkeit* aufgestellt werden. Aber selbst dann wären die funktionalen Zusammenhänge ausgesprochen kompliziert, da auch die *Massenverteilung* in der Erde variabel ist.

Es ist günstiger, auf die nachträglich aus Beobachtungen bestimmten *Erdrotationsparameter* zurückzugreifen, die in tabellierter Form vorliegen (Bulletin B). Ebenfalls in Tabellenform stehen prädizierte Werte für die Gegenwart und die nähere Zukunft bereit (Auszug aus *Bulletin B*, ftp://hpiers.obspm.fr/iers/bul/bulb/bulletinb.dat):

```
Date    MJD       x       y    UT1R-UTC    UT1R-TAI
1998              "       "        s           s
   (0h UTC)

 Final Bulletin B values.

  JUL   5  50999  -.05785  .44549  -.102249  -31.102249
  JUL  10  51004  -.04506  .45253  -.104006  -31.104006
  JUL  15  51009  -.03282  .46047  -.105785  -31.105785
  JUL  20  51014  -.02240  .46507  -.107264  -31.107264
  JUL  25  51019  -.01171  .46919  -.108731  -31.108731

  JUL  30  51024   .00040  .47444  -.110958  -31.110958
  AUG   4  51029   .00978  .47778  -.113728  -31.113728

 Preliminary extension, to be updated weekly in Bulletin A and monthly
  in Bulletin B.

  AUG   9  51034   .01998  .47855  -.116568  -31.116568
  AUG  14  51039   .03346  .47853  -.119274  -31.119274
  AUG  19  51044   .04405  .47734  -.121474  -31.121474

  AUG  24  51049   .05422  .47674  -.123818  -31.123818
  AUG  29  51054   .06564  .47655  -.126872  -31.126872
  SEP   3  51059   .07601  .47804  -.130672  -31.130672
  SEP   8  51064   .08569  .47885  -.135605  -31.135605
  SEP  13  51069   .09516  .47931  -.141781  -31.141781
```

Text 1: Auszug aus dem Bulletin B des International Earth Rotation Service (IERS)

Werte für dazwischenliegende Zeitpunkte werden mittels Interpolation bestimmt.

Die *Erdrotation* wird durch die Daten x_P, y_P (Lage des Pols) und UT1R-UTC bzw. UT1R-TAI (aktueller Drehwinkel) bezüglich UTC oder TAI ausreichend genau festgelegt. *UT1R* kann sich von UT1 um einige Millisekunden unterscheiden. Sie enthält dafür keine kurzperiodischen (< 35 Tage) Abhängigkeiten mehr.

Das Koordinatensystem (siehe Abbildung 9), in dem die *Polkoordinaten* definiert sind, ist ein Linkssystem. Dadurch fallen die notwendigen

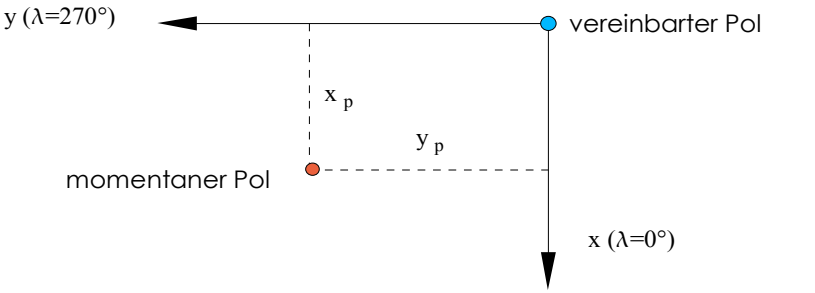

Abbildung 9: Der Übergang vom vereinbarten zum momentanen Pol wird durch die x- und y-Polkoordinaten vermittelt (λ = geogr. Länge)

Drehungen um die x- und y-Achse des erdfesten Äquatorsystems vorteilhafterweise im gleichen Drehsinn an.

```
/**
 *PolarData.java
 *
 * read earth orientation parameters from server or local machine
 * the file "data/bulletinb.dat" is supplied as URL or as String
 *
 * @see FileOfStrings
 * @see BullBFinal
 * @see BullBPrelim
 *
 * @author Dieter Egger
 * @version  1.3.1    2001/01/10
 *
 */

import java.applet.*;
import java.awt.*;
import java.util.StringTokenizer;
import java.net.*;

import mie.*;

public class PolarData extends FileOfStrings
{
  private BullBFinal bullbf;
  private BullBPrelim bullbp;

  public double jd, x, y, ut1_utc, ut1_tai;

  private double [] pV = new double [5];

  public PolarData (URL alpha)
  {
    super (alpha);
    bullbf = new BullBFinal (this);
    bullbp = new BullBPrelim (this);
  }

  public PolarData (String fn)
```

```
  {
    super (fn);
    bullbf = new BullBFinal (this);
    bullbp = new BullBPrelim (this);
  }

  public int sizePD () { return bullbf.size()+bullbp.size(); }

  public String line (int i)
  {
    if (i<0 || i>=sizePD()) return new String ("Out of bounds");
    else
    if (i<bullbf.size()) return (String)bullbf.elementAt (i);
    else
    return (String)bullbp.elementAt (i-bullbf.size());
  }

  public boolean parse (int i, double [] pV)
  {
    String s = line (i);
    StringTokenizer sd = new StringTokenizer (s);

    if (sd.countTokens()<5) return false;
    sd.nextToken();   // Month
    sd.nextToken();   // Day

    int j=0;
    while (sd.hasMoreTokens() && j<5)
    {
      pV [j++] = (new Double(sd.nextToken())).doubleValue ();
    }
    pV [0] += Aconst.JulianMJDZero;
    if (j<5) for (int k=j; k<5; k++) pV[k] = 0.0;
    return true;
  }

  public boolean actual (double juldat)
  // time intervals around leap second events are not handled properly
  // as well as time intervals at the beginning or at the end of the
data
  // only linear interpolation is used
  {
    double pvOld [] = new double [5];
    jd = juldat;
    x = y = ut1_utc = ut1_tai = 0.0;
    if (!parse (0, pvOld)) return false;
    if (juldat < pvOld [0]) return false;
    if (!parse (sizePD()-1, pV)) return false;
    if (juldat > pV [0]) return false;

    for (int i=1; i<sizePD(); i++)
    {
      if (!parse (i, pV)) return false;
      if (juldat < pV [0])
      {
        double ta = (juldat-pvOld[0])/(pV[0]-pvOld[0]);
        for (int j=1; j<5; j++)
            pV [j] = pvOld [j] + ta*(pV[j]-pvOld[j]);
        x = pV[1];
        y = pV[2];
```

```
        ut1_utc = pV[3];
        ut1_tai = pV[4];
        return true;
      }
      else for (int j=0; j<5; j++) pvOld [j] = pV [j];
    }
    return false;
  }
}
```

FileOfStrings

Wann immer Textdateien vom Server oder auch vom lokalen Rechner benötigt werden, bietet das Objekt *FileOfStrings* die geeignete Hilfestellung. Die zwei Konstruktormethoden werden abhängig davon aufgerufen, ob ein Applet oder eine Application vorliegen:

```
/**
 *FileOfStrings.java
 *
 * get ascii file contents from server or from local machine
 * every line is represented as String and collected in Vector
 *
 * @see URL
 * @see String
 * @see Vector
 *
 * @author Dieter Egger
 * @version  1.3.1    2001/01/10
 *
 */

import java.net.*;
import java.io.*;
import java.util.*;

public class FileOfStrings extends Vector
{
  public FileOfStrings () { super (10,10); }

  public FileOfStrings (URL alpha)
  { super (10,10); getData (alpha); }

  public FileOfStrings (String fn)
  { super (10,10); getData (fn); }

  public void getData (URL alpha)
  {
    try
    {
      URLConnection alphaConnection = alpha.openConnection ();

      BufferedReader dis = new BufferedReader
      (new InputStreamReader(alphaConnection.getInputStream ()));
      String inputLine;
      while ((inputLine = dis.readLine ()) != null)
      {
        addElement (inputLine);
      }
      dis.close ();
```

```
    }
    catch (MalformedURLException me)
    {
      addElement (new String ("MalformedURLException: " + me));
    }
    catch (IOException ioe)
    {
      addElement (new String ("IOException: " + ioe));
    }
  }

  public void getData (String fn)
  {
    try
    {
      BufferedReader dis = new BufferedReader(new FileReader (fn));
      String inputLine;
      while ((inputLine = dis.readLine ()) != null)
      {
        addElement (inputLine);
      }
      dis.close ();
    }
    catch (FileNotFoundException fne)
    {
      addElement (new String ("FileNotFoundException: " + fne));
    }
    catch (IOException ioe)
    {
      addElement (new String ("IOException: " + ioe));
    }
  }
}
```

BullBFinal

Aus einem Objekt *FileofStrings*, das den Text des Bulletin B enthält, werden die *final values* herausgefiltert:

```
/**
 *BullBFinal.java
 *
 * extract final values from Bulletin B (given in PolarData)
 *
 * @see FileOfStrings
 * @see PolarData
 * @see BullBPrelim
 *
 * @author Dieter Egger
 * @version  1.3.1    2001/01/10
 *
 */

import java.awt.*;
import java.util.*;

public class BullBFinal extends Vector
{
  public BullBFinal (FileOfStrings svf) { extractFinal (svf); }
```

```
  public void extractFinal (FileOfStrings svf)
  {
    boolean store = false;

    for (int i=0; i<svf.size(); i++)
    {
      String s = new String ((String) svf.elementAt (i));
      if (!store)
      {
        store = s.indexOf ("Final Bulletin B values") >= 0;
      }
      else
      {
        store = s.indexOf ("Preliminary extension") < 0;
        if (!store) break;
        if (s.trim ().length () > 60)
            addElement (svf.elementAt (i));
      }
    }
  }
}
```

Die Schlüsselworte zum Auffinden der entsprechenden Textpassagen müssen im Sourcecode angeglichen werden, falls sie sich im Bulletin B ändern sollten.

Das Gleiche gilt auch für das folgende Objekt *BullBPrelim* zur Extraktion der vorläufigen, also prädizierten Werte.

BullBPrelim

Ganz analog zu *BullBFinal* können aus einem *FileofStrings* auch die für den aktuellen Zeitraum und die nähere Zukunft vorhergesagten Werte extrahiert werden:

```
/**
 *BullBPrelim.java
 *
 * extract preliminary values from Bulletin B (given in PolarData)
 *
 * @see FileOfStrings
 * @see PolarData
 * @see BullBFinal
 *
 * @author Dieter Egger
 * @version  1.3.1    2001/01/10
 *
 */

import java.awt.*;
import java.util.*;

public class BullBPrelim extends Vector
{
  public BullBPrelim (FileOfStrings svf) { extractPrelim (svf); }

  public void extractPrelim (FileOfStrings svf)
  {
```

```
    boolean store = false;

    for (int i=0; i<svf.size(); i++)
    {
      String s = (String) svf.elementAt (i);
      if (!store)
      {
        store = s.indexOf ("Preliminary extension") >= 0;
      }
      else
      {
        store = s.indexOf ("Note.") < 0;
        if (!store) break;
        if (s.trim ().length () > 60)
            addElement (svf.elementAt (i));
      }
    }
  }
}
```

Auch hier bedeutet Vector das in Java vordefinierte Objekt, das eine Liste von beliebigen Objekten verwalten kann. In diesem Fall handelt es sich um Strings.

Bezugssysteme

Fixiert man ein Koordinatensystem im Raum, so kann es als *Bezugssystem* dienen, um beispielsweise Satellitenorte und –geschwindigkeiten anzugeben. Kennt man die Lage des Ursprungs, die Richtung der x-Achse und die Richtung der y- oder z-Achse, so machen Koordinatenangaben einen Sinn . Anstelle der y- oder z-Achse findet man häufig die Ebene, in der sich x- und y-Achse befinden, als weitere Angabe.

In der Astronomie unterscheiden wir beispielsweise helio- (Ursprung in der Sonne) und geozentrische (Ursprung in der Erde) Bezugssysteme, deren x-Achse entweder *raumfest* (immer zum Frühlingspunkt zeigend) oder *körperfest* (im Fall der Erde stets zum Schnittpunkt des Nullmeridians mit dem Äquator zeigend) gelagert ist. x- und y-Achse liegen meist in der Ekliptik- oder der Äquatorebene, bei topozentrischen Systemen in der Horizontebene.

Kurz-bez.	Bezugssystem	Ursprung	x-Achse	Ebene
BE	Barycentric Ecliptical	Baryzentrum	Frühlingspunkt	Ekliptik
HE	Heliocentric Ecliptical	Heliozentrum	Frühlingspunkt	Ekliptik
GE	Geocentric Ecliptical	Geozentrum	Frühlingspunkt	Ekliptik
SF	Space-Fixed Equatorial	Geozentrum	Frühlingspunkt	Äquator
EF	Earth-Fixed Equatorial	Geozentrum	Meridian von Greenwich	Äquator
TE	Topocentric Equatorial	Topozentrum	Meridian von Greenwich	Parallel zum Äquator
TH	Topocentric Horizon	Topozentrum	Süden, y-Achse nach Osten	Horizont

Tabelle 3: Die gebräuchlichsten Bezugssysteme

Zusätze wie *mittleres*, *wahres*, *momentanes* oder *vereinbartes* oder *zur Epoche J2000* oder *zur aktuellen Epoche*, verfeinern die grobe Unterteilung der Referenzsysteme.

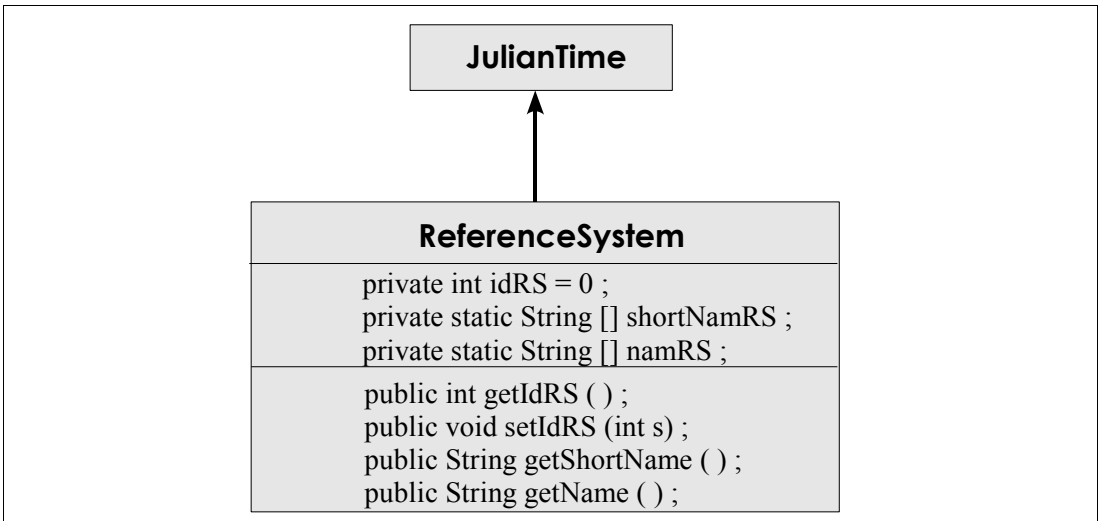

Abbildung 10: ReferenceSystem erbt von JulianTime. Damit wird die zeitliche Veränderlichkeit eines Bezugssystems unterstrichen.

Zwei moderne *Referenzsysteme* werden mit ICRS (International Celestial Reference System) und ITRS (International Terrestrial Reference System) bezeichnet.

ICRS entspricht einem baryzentrischen, mittleren, raumfesten Äquatorsystem der Epoche J2000.0, das aufgrund hochgenau bestimmter Positionen (< 0.1 Millibogensekunden) ausgewählter Quasare realisiert wird. Sternpositionen, die vom Satelliten Hipparcos vermessen worden sind, finden ebenfalls Eingang in die praktische Realisierung des ICRS, dem ICRF (International Celestial Reference Frame). Seine Genauigkeit ist um wenigstens zwei Grössenordnungen höher als beim FK5-System.

ITRS ist ein vereinbartes, erdfestes Äquatorsystem, das ausgewählten Beobachterpositionen auf der Erde Referenzcharakter zuweist.

Die Verknüpfung von ICRS und ITRS geschieht über Präzession, Nutation, wahre Sternzeit von Greenwich und die Erdrotationsparameter (siehe auch Seite 39).

Die üblicherweise in der Astronomie und Himmelsmechanik eingesetzten Bezugs- oder Referenzsysteme verändern ihre Lage im Vergleich zu einem an Quasaren orientierten *Bezugssystem*.

Wegen der Wanderung des *Frühlingspunktes* entlang der Ekliptik im Uhrzeigersinn bleiben weder ekliptikale noch äquatoriale „raumfeste“ Bezugssysteme von Variationen verschont. Bei erdgebundenen Systemen ist die Zeitabhängigkeit wegen der Erddrehung ohnehin offensichtlich.

Da es keine absoluten Bezugssysteme gibt, werden die zeitlichen Abhängigkeiten als gegenseitige Abhängigkeiten beschrieben, die natürlich erst dann zum Tragen kommen, wenn von einem System ins andere transformiert werden soll.

Welches System als Ausgangssystem für die Transformationen gewählt wird, ist Definitionssache. Es ist natürlich naheliegend, ein Bezugssystem zu wählen, das möglichst wenig Änderungen unterliegt und auch in der Praxis gut realisiert werden kann. War es früher das *FK5-System* (noch früher FK4, FK=Fundamentalkatalog), ein mit sehr genau vermessenen Sternpositionen und –bewegungen aufgebautes Referenzsystem, ist es heutzutage das ICRS (International Celestial Reference System). Das *ICRS* basiert auf hochgenau vermessenen *Quasarpositionen* (mit VLBI bestimmt). Zusätzlich sind die ebenfalls hochgenau vermessenen Sternpositionen und –bewegungen des *Hipparcos*-Systems eingebettet worden (mit dem Satelliten Hipparcos bestimmt). Im Genauigkeitsrahmen des *FK5-Systems* stimmen FK5-System und ICRS überein. ICRS ist jedoch wesentlich genauer (vergl. [Zebhauser]).

Abbildung 11 soll helfen, die Zusammenhänge zwischen den verschiedenen Bezugssystemen zu verstehen. Dabei werden auch die Objekte angeführt, die für die Übergänge verantwortlich zeichnen.

Moderne *Planetentheorien* bauen auf dem baryzentrischen *Ekliptiksystem* zur Standardepoche J2000 auf. Positionen von FK5-Sternen werden in einem baryzentrischen *Äquatorsystem* zur selben Epoche definiert.

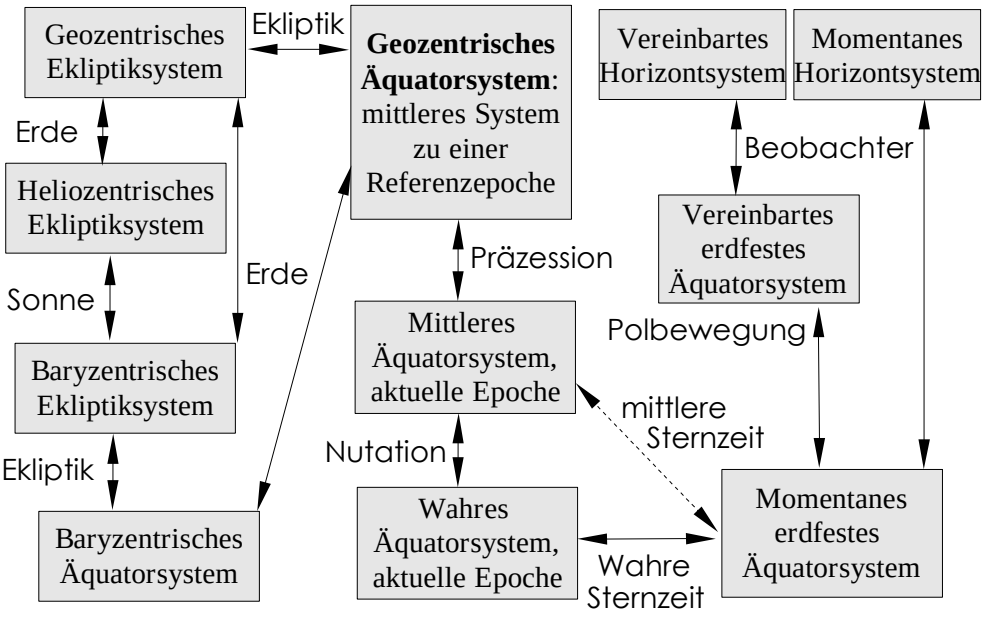

Abbildung 11: Die gebräuchlichen Bezugssysteme in der Astronomie und Himmelsmechanik und ihre Zusammenhänge. Wünscht man den Einfluss der Nutation zu berücksichtigen, so ist auch die wahre Sternzeit einzusetzen; ansonsten genügt die mittlere Sternzeit für den Übergang vom mittleren raumfesten in das momentane erdfeste Äquatorsystem.

Ist das mittlere baryzentrische Äquatorsystem zur Referenzepoche J2000 gemeint, so entspricht es dem *ICRS* (International Celestial Reference System). Das vereinbarte, erdfeste Äquatorsystem entspricht dem *ITRS* (International Terrestrial Reference System).

Für geringere Genauigkeitsansprüche (z.B. eine halbe Bogenminute) kann die Nutation vernachlässigt werden. Dann ist adäquat die mittlere Sternzeit zu verwenden, um ins erdfeste System zu gelangen. Von dort kann, bei kompletter Vernachlässigung der Polbewegung, sofort in das (momentane) Horizontsystem des Beobachters transformiert werden. Falls man sich nahe an der Standardepoche J2000 befindet, kann auch die Präzession vernachlässigt werden.

```
/**
 *ReferenceSystem.java
 *
 * JulianTime plus a corresponding reference system
 * chosen out of a collection of widely used ref.systems
 *
 * @see JulianTime
 *
 * @author Dieter Egger
 * @version  1.3.1    2001/01/10
 *
 */

package mie;

public class ReferenceSystem extends JulianTime
{
  private int idRS = 1;
```

```
public final static int numRS = 10;

public final static int BE = 0;
public final static int HE = 1;
public final static int GE = 2;
public final static int SFJ2k = 3;
public final static int SF = 4;
public final static int SFT = 5;
public final static int EF = 6;
public final static int EFC = 7;
public final static int TE = 8;
public final static int TH = 9;

private static String namRS [] = {
  "Barycentric Ecliptical",
  "Heliocentric Ecliptical",
  "Geocentric Ecliptical",
  "Space-Fixed Equatorial J2000",
  "Space-Fixed Equatorial",
  "Space-Fixed Equatorial True",
  "Earth-Fixed Equatorial",
  "Earth-Fixed Equatorial Conventional",
  "Topocentric Equatorial",
  "Topocentric Horizon"
  };

private static String [] shortNamRS =
  { "BE", "HE", "GE",
    "SFJ2k", "SF", "SFT",
    "EF", "EFC",
    "TE", "TH" };

private static String [] sXYZ = { "X", "Y", "Z" };
private static String [] sRPhiTheta = { "R", "Phi", "The" };
private static String [] sRLongLat = { "R", "Lon", "Lat" };
private static String [] sRRaDec = { "R", "RA", "Dec" };
private static String [] sRAzEl = { "R", "Az", "El" };

private static String [][] labelRS =
{ sRLongLat, sRLongLat, sRLongLat,
  sRRaDec, sRRaDec, sRRaDec,
  sRRaDec, sRRaDec,
  sRLongLat, sRAzEl };

public ReferenceSystem () { idRS = 1; }
public ReferenceSystem (int s) { setIdRS (s); }
public ReferenceSystem (String sys) { setIdRS (sys); }
public ReferenceSystem (JulianTime jt, int s)
{ super (jt); setIdRS (s); }
public ReferenceSystem (JulianTime jt, String sys)
{ super (jt); setIdRS (sys); }
public ReferenceSystem (ReferenceSystem rs)
{ super (rs); setIdRS (rs.getIdRS ()); }

public void setReferenceSystem (ReferenceSystem rs)
{ setJulianTime (rs); setIdRS (rs.getIdRS ()); }

public void setIdRS (int s) { if (s>=0 && s<numRS) idRS = s; }
public void setIdRS (String sys)
```

```
  {
    int i;
    for (i=0; i<numRS; i++)
      if (sys.equals (shortNamRS [i])) { idRS = i; break; }
  }
  public boolean isRS (String sys)
  {
    int i;
    for (i=0; i<numRS; i++)
      if (sys.equals (shortNamRS [i])) { return true; }
    return false;
  }

  public int getIdRS () { return idRS; }
  public String getShortName () { return shortNamRS [idRS]; }
  public String getName () { return namRS [idRS]; }
  public String [] getLabel () { return labelRS [idRS]; }
  public String [] getLabelXYZ () { return sXYZ; }
  public String [] getLabelPolar () { return sRPhiTheta; }

  public String toString ()
  {
    return "ReferenceSystem  "+getShortName()+"  "+getName()
            +"\n  ReferenceEpoch "
            +super.toString ();
  }
}
```

Orbit

Keplersche Bahnelemente

Ein Satz von 6 Grössen, die **Form** (*Exzentrizität e* und *Grosse Halbachse a*) und **Lage** (*aufsteigender Knoten* Ω, *Inklination i* und *Perizentrum* ω) einer *Bahnellipse*, sowie die momentane **Stellung** des umlaufenden Körpers entlang der Bahn (gezählt ab dem Perizentrum: wahre Anomalie ν für die tatsächliche Position, mittlere *Anomalie M* für die mittlere Position bei einer gleichförmig gedachten Bewegung), eindeutig festlegen.

Die Keplerschen Bahnelemente definieren die sogenannte *Keplerellipse*, eine idealisierte Bahn im *Zweikörperproblem* (punktförmige Massen, wobei die Masse des umlaufenden Körpers komplett vernachlässigt wird).

```
/**
 *OrbitElements.java
 *
 * the six Kepler elements of an orbit:
 * semi-major axis, eccentricity, inclination
 * pericentre, ascending node, mean anomaly
 *
 * @author Dieter Egger
 * @version  1.3.1    2001/01/10
 *
 */

package mie;
```

```
public class OrbitElements extends Object
{
  private double A;
  private double e;
  private double i;
  private double w;
  private double O;
  private double M;

  public OrbitElements () { A = e = i = w = O = M = 0.0; }

  public OrbitElements
       (double A, double e, double i, double w, double O, double M)
  {
    setAeiwOM (A, e, i, w, O, M);
  }

  public OrbitElements (OrbitElements B) { setOrbitElements (B); }

  public void setElements
(double pA, double pe, double pi, double pw, double pO, double pM)
  {
    A = pA;
    e = pe;
    i = pi;
    w = pw;
    O = pO;
    M = pM;
  }

  public void setAeiwOM
(double pA, double pe, double pi, double pw, double pO, double pM)
  {
    setA (pA);
    setE (pe);
    setI (pi);
    setW (pw);
    setO (pO);
    setM (pM);
  }

  public void setOrbitElements (OrbitElements B)
  {
    A = B.getA();
    e = B.getE();
    i = B.getI();
    w = B.getW();
    O = B.getO();
    M = B.getM();
  }

  public void setMeanAnomaly (double ma) { setM (ma); }
  public void increaseMeanAnomaly (double incr) { setM (M+incr); }

  // Access Data

  public double getA () { return A; }
  public double getE () { return e; }
  public double getI () { return i; }
```

```
  public double getW () { return w; }
  public double getO () { return O; }
  public double getM () { return M; }

  public void setA (double d) { A = Math.abs (d); }
  public void setE (double d) { if (d>=0.0 && d<=1.0) e = d; }
  public void setI (double d) { i = MyLib.modulo (d, Aconst.Pi); }
  public void setW (double d) { w = MyLib.mod2Pi (d); }
  public void setO (double d) { O = MyLib.mod2Pi (d); }
  public void setM (double d) { M = MyLib.mod2Pi (d); }

  public String toString ()
  {
    return "OrbitElements A [km] "+MyLib.display(A*0.001,6)+
         "\t e= "+MyLib.display(e,6)+
         "\t i= "+MyLib.display(i*Aconst.Radian2Degree,4)+
         "\t w= "+MyLib.display(w*Aconst.Radian2Degree,4)+
         "\t O= "+MyLib.display(O*Aconst.Radian2Degree,4)+
         "\t M= "+MyLib.display(M*Aconst.Radian2Degree,4);
  }
}
```

Orbit-Vektor

Orts- und Geschwindigkeitsvektor des umlaufenden Körpers werden zum Orbit-Vektor zusammengefasst. Im Falle von Keplerellipsen existiert eine eindeutige Zuordnung zwischen Keplerschen Bahnelementen und Orbit-Vektoren.

Inhomogene Massenanordnungen und/oder weitere Massenkörper erlauben im allgemeinen keine einfachen Lösungen der *Bewegungsgleichungen* mehr, so dass die Orbit-Vektoren zwar in Bahnelemente umgerechnet werden können, die resultierende Ellipse jedoch nicht mehr der tatsächlichen Umlaufbewegung entspricht. Lediglich für einen kurzen Zeitraum repräsentiert diese *oskulierende Ellipse* den Bahnverlauf.

```
/**
 *OrbitVector.java
 *
 * Vector3D representing the position plus
 * Vector3D representing the velocity
 *
 * @see  Vector3D
 *
 * @author Dieter Egger
 * @version  1.3.1    2001/01/10
 *
 */

package mie;

public class OrbitVector extends Vector3D
{
  protected Vector3D vel;

  // Constructors
```

```
  public OrbitVector ()
  {
    vel = new Vector3D ();
  }

  public OrbitVector (Vector3D r)
  {
    super (r);
    vel = new Vector3D ();
  }

  public OrbitVector (Vector3D r, Vector3D v)
  {
    super (r);
    vel = new Vector3D (v);
  }

  public OrbitVector (OrbitVector B)
  {
    super (B);
    vel = new Vector3D (B.getVel());
  }

  public void setPosVel (Vector3D r, Vector3D v)
  {
    setVector3D (r);
    vel.setVector3D (v);
  }

  public void setOrbitVector (OrbitVector B)
  {
    setVector3D (B);
    vel.setVector3D (B.getVel());
  }

  // Access Data

  public Vector3D R () { return this; }
  public Vector3D V () { return vel; }

  public void setR (Vector3D r) {     setVector3D (r); }
  public void setV (Vector3D v) {     vel.setVector3D (v); }

  public Vector3D getPos () { return this; }
  public Vector3D getVel () { return vel; }

  public void setPos (Vector3D r) { setVector3D (r); }
  public void setVel (Vector3D v) { vel.setVector3D (v); }

  public String toString ()
  {
    return "OrbitVector\n R   [km] "+Vector3D.times(0.001,this)+
        "\n V [km/s] "+Vector3D.times(0.001,vel);
  }
}
```

Beschreibung der Umlaufbahn

Wahlweise ein Orbit-Vektor oder 6 Bahnelemente legen die *Umlaufbahn* eines Satelliten oder Planeten fest.

Bei Satellitenbahnen unterstellt man meist stillschweigend ein raumfestes Bezugssystem, dessen Ursprung im Massenmittelpunkt der Erde liegt, dessen x-Achse zum *Frühlingspunkt* zeigt und dessen Bezugsebene die *Äquatorebene* der Erde ist.

Bei Planeten und Kometen des Sonnensystems impliziert man meist ein heliozentrisches Bezugssystem mit der *Ekliptikebene* als Bezugsebene und ebenfalls der Richtung zum Frühlingspunkt als x-Richtung.

Orbit-Vektor oder Bahnelemente gelten jeweils für einen bestimmten Zeitpunkt, die sogenannte *Bahnepoche*. Ausserdem gelten sie in einem Bezugssystem, das ebenfalls für einen bestimmten Zeitpunkt, die sogenannte Bezugssystem-Epoche definiert ist. Die beiden Epochen müssen nicht notwendigerweise übereinstimmen.

Die Angabe der *Bezugssystem-Epoche* ist notwendig, da sich sowohl die Richtung zum Frühlingspunkt, als auch die Lage der Äquatorebene laufend verändern (Präzession, Nutation, Polbewegung). Schliesslich muss noch die Masse des *Zentralkörpers*, bzw. ihr Produkt mit der NEWTONschen Gravitationskonstanten bekannt sein, wenn man Bahnelemente und Orbit-Vektoren ineinander umrechnen möchte.

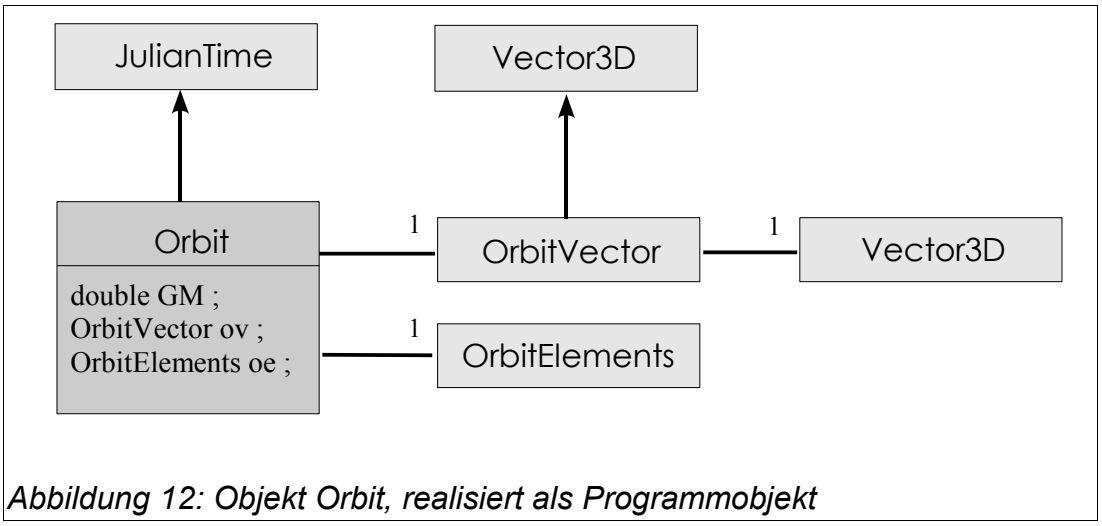

Abbildung 12: Objekt Orbit, realisiert als Programmobjekt

Im Falle einer ungestörten *Keplerbahn* bleiben Form und Lage der Bahnellipse über die Zeit hinweg erhalten. Nur die *Mittlere Anomalie* verändert sich gleichförmig und kann daher leicht für neue Zeitpunkte berechnet werden. Bei *Kreisbahnen* fallen mittlere und wahre *Anomalie* zusammen. Die wahre Anomalie beschreibt stets die tatsächliche Bahnposition. Bei exzentrischen Bahnen ist sie von der mittleren Anomalie

verschieden. Sie verändert sich gemäss dem zweiten Keplerschen Gesetz. Zwischen *Mittlerer Anomalie* und *Exzentrischer Anomalie* wird durch die *Keplergleichung* ein Zusammenhang hergestellt. Von der exzentrischen Anomalie kann die wahre Anomalie einfach abgeleitet werden (siehe Seite 49).

Da sich Himmelskörper bewegen, ist dieses Objekt ganz offensichtlich zeitabhängig. Die zugehörige Zeitskala ist dabei einer dynamischen Zeit zuzuordnen, in der die Bewegungsgleichungen formuliert sind.

Die einfachste und damit auch idealisierte Form der Orbits bildet die *Ellipse*, gekennzeichnet durch ihre Grosse *Halbachse* a und ihre *Exzentrizität* e (siehe Abbildung 13).

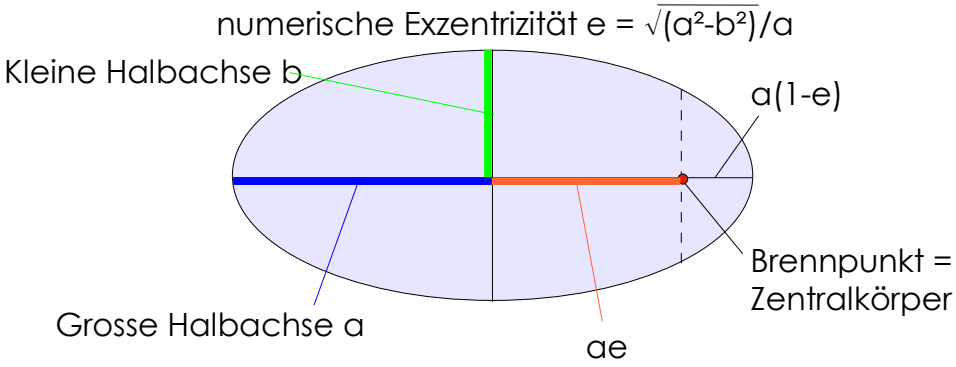

Abbildung 13: Beschreibung der Gestalt der Ellipse durch die beiden Gestaltparameter Grosse Halbachse a und Exzentrizität e

Der Umlauf des Himmelskörpers um den *Brennpunkt* wird durch die Keplerschen Gesetze geregelt. Zur Kennzeichnung des momentanen Punktes auf der *Umlaufbahn* (siehe Abbildung 14) dient die *wahre* Anomalie *v,* die wegen ihres ungleichmässigen Ablaufs gerne durch die *exzentrische* Anomalie *E* bzw. durch die gleichmässig verstreichende *mittlere* Anomalie *M* ersetzt wird:

$$M = n(t - \tau)$$

mit τ Zeitpunkt, zu dem der Himmelskörper dem *Brennpunkt* am nächsten kommt (*Perizentrumsdurchgang*) und

$$n = \sqrt{\frac{GM}{a^3}}$$

mittlere Bewegung, *GM = Gravitationskonstante * Masse des Zentralkörpers;*

Berechnung von *E* über die *Keplergleichung*, die iterativ zu lösen ist

$$E = M + e\sin(E)$$

und schliesslich quadrantengerecht die wahre Anomalie mittels der Arcus-Tangens Funktion mit zwei Argumenten [Schneider, Satellitengeodäsie].

$$v = atan2\left(\frac{\sqrt{1-e^2}\sin(E)}{1-e\cos(E)}, \frac{\cos(E)-e}{1-e\cos(E)}\right)$$

Die mittlere Anomalie *M* entspricht einem gedachten Körper, der in der gleichen Zeit gleichmässig auf dem äusseren Kreis umläuft, wie der reale Körper auf der Ellipse. *M* wird, wie *E* vom Ellipsenzentrum aus positiv gegen

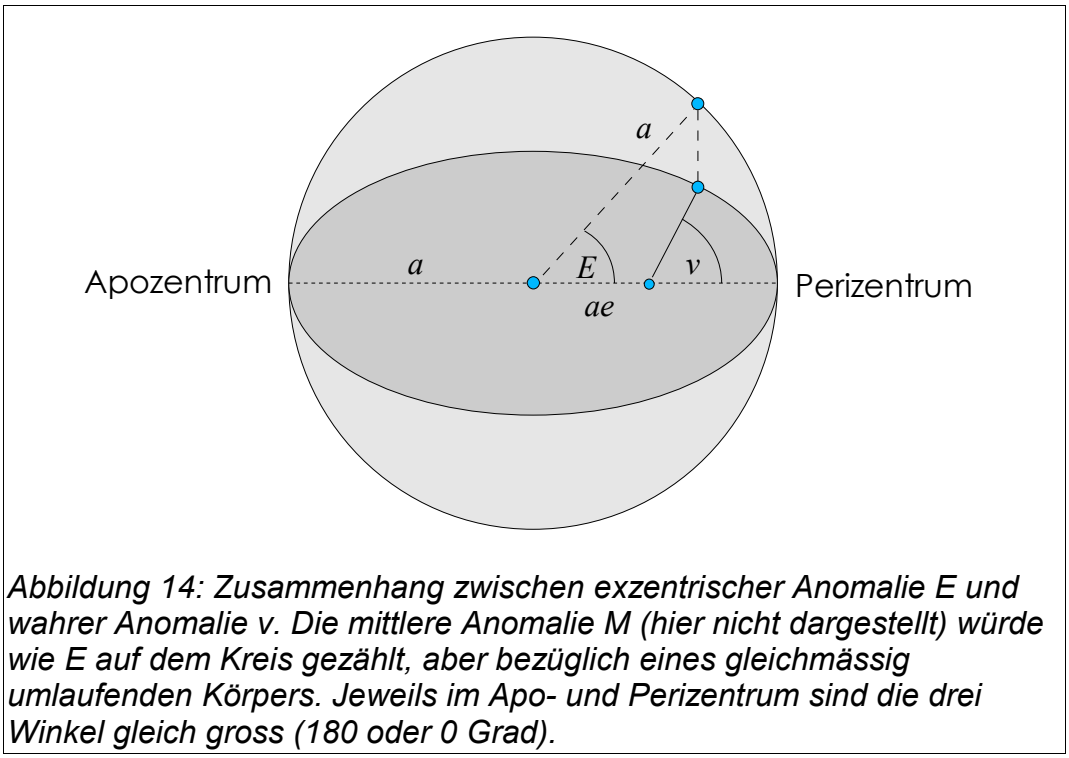

Abbildung 14: Zusammenhang zwischen exzentrischer Anomalie E und wahrer Anomalie v. Die mittlere Anomalie M (hier nicht dargestellt) würde wie E auf dem Kreis gezählt, aber bezüglich eines gleichmässig umlaufenden Körpers. Jeweils im Apo- und Perizentrum sind die drei Winkel gleich gross (180 oder 0 Grad).

den Uhrzeigersinn gezählt. Wie auch bei *v* liegt der Zählbeginn auf der grossen Halbachse in Richtung *Perizentrum*. Jeweils im Peri- (der kleinsten Entfernung vom Zentralkörper) und im *Apozentrum* (der grössten Entfernung vom Zentralkörper) stimmen *v*, E und M überein. Bei einer *Kreisbahn* sind die drei Winkel stets identisch.

Zur Lagebeschreibung der *Ellipse* im Raum (siehe Abbildung 15) braucht man drei weitere Winkel, die relativ zu einem Bezugssystem angegeben werden: Von der x-Achse aus wird in mathematisch positiver Richtung der Winkel Ω gezählt, der zur sogenannten *Knotenlinie* zeigt. Es wird der Teil der Knotenlinie betrachtet, der zum *aufsteigenden* Bahnknoten zeigt. Gemeint ist der Punkt in der x-y-Ebene, der vom Himmelskörper von unten nach oben durchstossen wird.

Von der Knotenlinie aus wird der Winkel ω gezählt, der zum zentrumsnächsten Punkt der Bahnellipse weist.

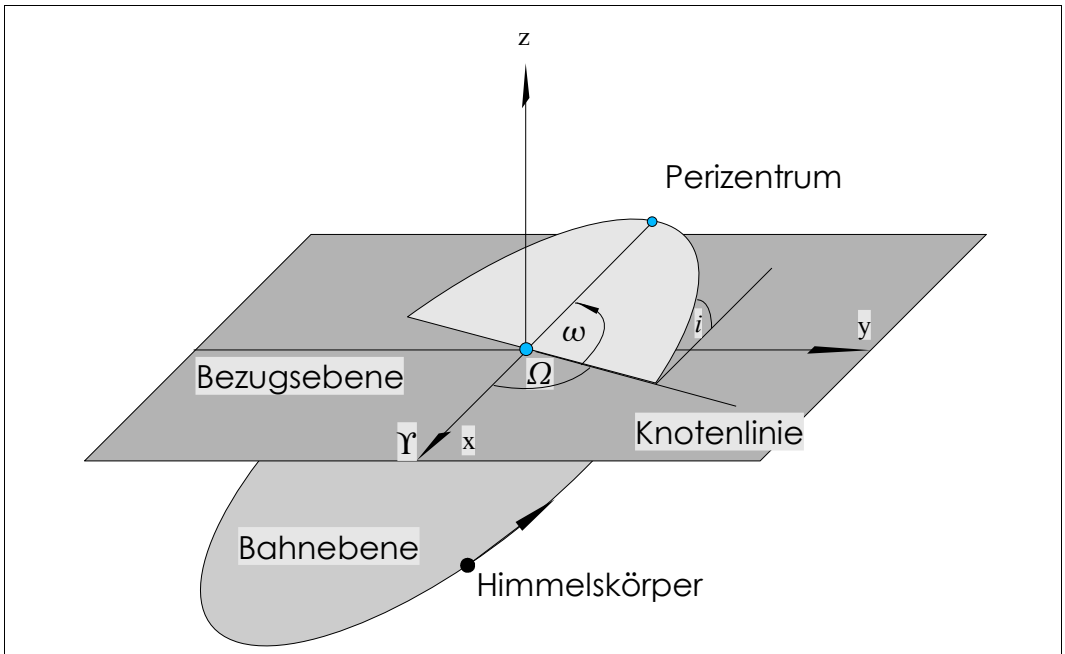

Abbildung 15: Drei Kenngrössen legen die Lage der Ellipse im Raum fest

Die *Bahnneigung* selbst wird durch die *Inklination i* beschrieben. Sie kann 0 bis 180 Grad betragen.

```
/**
 *Orbit.java
 *
 * JulianTime plus OrbitVector and OrbitElements
 * and the product of mass and gravitational constant (GM)
 * of the central body
 *
 * @see JulianTime
 * @see OrbitVector
 * @see OrbitElements
 *
 * @author Dieter Egger
 * @version  1.3.1    2001/01/10
 *
 */

package mie;

public class Orbit extends JulianTime
{
  protected double GM;                      // grav.constant times mass
  protected OrbitVector   orbitVector;      // position and velocity
  protected OrbitElements orbitElements;// Kepler elements

  private void init ()
  {
    orbitVector = new OrbitVector ();
    orbitElements = new OrbitElements ();
    orbitElements.setA (42.0e6);
    elements2vector ();
  }

  public Orbit ()
  {
```

```
    super ();
    GM = Aconst.GM_Earth;
    init ();
  }

  public Orbit (JulianTime t)
  {
    super (t);
    GM = Aconst.GM_Earth;
    init ();
  }

  public Orbit (JulianTime t, double GM)
  {
    super (t);
    this.GM = GM;
    init ();
  }

  public Orbit (JulianTime t, double GM, OrbitVector ov)
  {
    super (t);
    this.GM = GM;
    orbitVector = new OrbitVector (ov);
    orbitElements = new OrbitElements ();
    vector2elements ();
  }

  public Orbit (JulianTime t, double GM, OrbitElements oe)
  {
    super (t);
    this.GM = GM;
    orbitVector = new OrbitVector ();
    orbitElements = new OrbitElements (oe);
    elements2vector ();
  }

  public Orbit (Orbit orb)
  {
    super (orb);
    this.GM = orb.getGM ();
    orbitVector = new OrbitVector (orb.getOrbitVector());
    orbitElements = new OrbitElements (orb.getOrbitElements());
  }

  // Access Data

  public JulianTime    getOrbitEpoch ()    { return this; }
  public double        getGM ()            { return GM; }
  public OrbitVector   getOrbitVector ()   { return orbitVector; }
  public OrbitElements getOrbitElements ()
  { return orbitElements; }

  public void setOrbitEpoch (JulianTime t_epoch)
    { super.setJulianTime (t_epoch); }
  public void setGM (double GM) { this.GM = GM; }
  public void setOrbitVector   (OrbitVector ov)
    { orbitVector.setOrbitVector (ov);   vector2elements (); }
  public void setOrbitElements (OrbitElements oe)
    { orbitElements.setOrbitElements (oe); elements2vector (); }
```

```
public void setPosVel (Vector3D pos, Vector3D vel)
  { orbitVector.setPosVel (pos, vel); vector2elements (); }

public void setOrbit (Orbit orb)
{
  setOrbitEpoch (orb.getOrbitEpoch());
  setGM (orb.getGM ());
  orbitVector.setOrbitVector (orb.getOrbitVector ());
  orbitElements.setOrbitElements (orb.getOrbitElements ());
}

public Vector3D getPos () { return orbitVector.getPos (); }
public Vector3D getVel () { return orbitVector.getVel (); }
public double getA () { return orbitElements.getA (); }
public double getE () { return orbitElements.getE (); }
public double getI () { return orbitElements.getI (); }
public double getW () { return orbitElements.getW (); }
public double getO () { return orbitElements.getO (); }
public double getM () { return orbitElements.getM (); }

// Change time

public void setJulianDate (double epoch)
{
  double delta_t = epoch - julianDate ();
  super.setJulianDate (epoch);
  orbitElements.increaseMeanAnomaly
                   (delta_t* Aconst.SolarDay* meanMotion());
  elements2vector ();
}

public void setJulianTime (JulianTime t)
{
  JulianTime delta_t = new JulianTime (t);
  delta_t.minus (this);
  super.setJulianTime (t);
  orbitElements.increaseMeanAnomaly
           (delta_t.julianDate()* Aconst.SolarDay* meanMotion());
  elements2vector ();
}

public void plus  (JulianTime delta_t)
{
  super.plus (delta_t);
  orbitElements.increaseMeanAnomaly
           (delta_t.julianDate()* Aconst.SolarDay* meanMotion());
  elements2vector ();
}

public void minus (JulianTime delta_t)
{
  super.minus (delta_t);
  orbitElements.increaseMeanAnomaly
           (-delta_t.julianDate()* Aconst.SolarDay* meanMotion());
  elements2vector ();
}

// Additional Stuff
```

```
  public double meanMotion ()
  { double A = orbitElements.getA();
    return Math.sqrt (GM/(A*A*A)); }

  public static double meanMotion (double GM, double A)
  { return Math.sqrt (GM/(A*A*A)); }

  public double timeOfRevolution ()
  { return Aconst.TwoPi/meanMotion (); }

  public double timeOfPericentre ()
  { return
julianDate()-orbitElements.getM()/meanMotion()/Aconst.SolarDay; }

  public static double KeplerEq (double M, double e)
  /*-----------------------------------------------
    Solution of Kepler's equation  E = M + e*sin (E)
    by means of Newton-Iteration. Calculation of
    excentric anomaly from mean anomaly
    -----------------------------------------------*/
  {
    double Ei, Ej;
    int i;

    i = 0;
    Ej = M;
    if (e>0.8) Ej = Aconst.Pi;
    do
    {
      Ei = Ej;
      Ej = Ei + (M + e*Math.sin (Ei) - Ei)/(1 - e*Math.cos (Ei));
      i++;
    }
    while ((( (Ej-Ei)>0 ? (Ej-Ei) : (Ei-Ej) ) > 1E-14) && (i<20));
    return (Ej);
  }


  private void elements2vector ()
  // ------------------------ Orbital Elements to Orbit Vector
  {
    double cw, cO, ci, sw, sO, si;
    double EA, cE, sE;

    cw = Math.cos (orbitElements.getW());
    sw = Math.sin (orbitElements.getW());
    cO = Math.cos (orbitElements.getO());
    sO = Math.sin (orbitElements.getO());
    ci = Math.cos (orbitElements.getI());
    si = Math.sin (orbitElements.getI());

    Vector3D p =
      new Vector3D ( cw*cO - ci*sO*sw,  cw*sO + ci*cO*sw, si*sw);
    Vector3D q =
      new Vector3D (-sw*cO - ci*sO*cw, -sw*sO + ci*cO*cw, si*cw);

    EA = KeplerEq (orbitElements.getM(), orbitElements.getE());
    cE = Math.cos (EA);
    sE = Math.sin (EA);
```

```
    double f1 = cE - orbitElements.getE();
    double f2 =
    Math.sqrt (1.0-orbitElements.getE()*orbitElements.getE())*sE;
    Vector3D hp = new Vector3D (p);
    hp.times (f1);
    Vector3D hq = new Vector3D (q);
    hq.times (f2);
    Vector3D ort = new Vector3D (hp);
    ort.plus (hq);
    ort.times (orbitElements.getA());

    f1 = Math.sqrt (GM/orbitElements.getA())
          / (1.0-orbitElements.getE()*cE);
    f2 = Math.sqrt
          (1.0-orbitElements.getE()*orbitElements.getE()) *cE;
    hp.setVector3D (p);
    hq.setVector3D (q);
    hp.times (sE);
    hq.times (f2);
    Vector3D ges = new Vector3D (hq);
    ges.minus (hp);
    ges.times (f1);

    orbitVector.setPosVel (ort, ges);
  }


  private void vector2elements ()
  // ------------------------ Orbit Vector to Orbital Elements
  {
    Vector3D c0, k0, p0, q0, r0, hv;
    double bc, vq, br, cv, sv, p, ea, wa;
    double bA, be, bi, bw, bO, bM;

    c0 =
      new Vector3D (Vector3D.cross (orbitVector.getPos(),
                                    orbitVector.getVel()));
    bc = Vector3D.abs (c0);
    c0.setUnit();

    bi = Math.acos (c0.x[2]);                    /* inclination */
    if (c0.x[1] == 0.0) bO = 0.0;
    else bO = Math.atan2 (c0.x[0], -c0.x[1]);/* ascending node  */

    r0 = new Vector3D ((orbitVector.getPos()).unit());
    br = Vector3D.abs (orbitVector.getPos());
    vq =
      Vector3D.dot (orbitVector.getVel(), orbitVector.getVel());
    p = bc*bc/GM;

    bA = br*GM/(2.0*GM - br*vq);                /* semimajor axis */
    be = Math.sqrt (1.0 - p/bA);                /* eccentricity   */

    cv = (p - br)/(be*br);
    sv = p *
          Vector3D.dot(orbitVector.getPos(),
                       orbitVector.getVel())/(be*br*bc);
    hv = new Vector3D (Vector3D.cross (c0, r0));
    p0 = new Vector3D (r0);
    p0.times (cv);
```

```
    q0 = new Vector3D (hv);
    q0.times (sv);
    p0.minus (q0);

    Vector3D hh = new Vector3D (hv);
    hh.times (cv);
    q0.setVector3D (r0);
    q0.times (sv);
    q0.plus (hh);
    k0 = new Vector3D (Math.cos (bO), Math.sin (bO), 0.0);

    bw = Math.atan2 (-Vector3D.dot (k0, q0),
                      Vector3D.dot (k0, p0));/* pericenter */

    wa = Math.atan2 (sv, cv);
    ea = 2.0*
    Math.atan (Math.tan (wa/2.0)*Math.sqrt ((1.0-be)/(1.0+be)));

    bM = ea - be*Math.sin (ea);            /* mean anomaly */
    if (bM < 0.0) bM += Aconst.TwoPi;
    orbitElements.setAeiwOM (bA, be, bi, bw, bO, bM);
  }
}
```

Nimmt man ein Referenzsystem hinzu, in dem der Orbit gilt, so „landet“ man beim RefOrbit:

```
/**
 *RefOrbit.java
 *
 * Orbit in a given reference system
 *
 * @see Orbit
 * @see ReferenceSystem
 *
 * @author Dieter Egger
 * @version  1.3.1    2001/01/10
 *
 */

package mie;

public class RefOrbit extends Orbit
{
  protected ReferenceSystem refSys;

  public RefOrbit ()
  {
    refSys = new ReferenceSystem ();
    refSys.setJulianTime (JulianTime.actualNoon ());
  }

  public RefOrbit (Orbit v)
  {
    super (v);
    refSys = new ReferenceSystem ();
    refSys.setJulianTime (JulianTime.actualNoon ());
  }

  public RefOrbit (Orbit v, ReferenceSystem rf)
```

```
  {
    super (v);
    refSys = new ReferenceSystem (rf);
  }

  public RefOrbit (RefOrbit old)
  {
    refSys = new ReferenceSystem ();
    setRefOrbit (old);
  }

  public void setRefOrbit (RefOrbit ro)
  {
    setOrbit (ro);
    refSys.setReferenceSystem (ro.refSys);
  }

  public ReferenceSystem getRefSys () { return refSys; }

  public void setRefSys (ReferenceSystem rs)
  { refSys.setReferenceSystem (rs); }

  public void setIdRS (String rs) { refSys.setIdRS (rs); }

  public void setJulianTimeRS (JulianTime t)
  { refSys.setJulianTime (t); }
}
```

Himmelskörper

Alle Körper des Sonnensystems können als Objekte beschrieben werden, die gravitativ wechselwirken und um einen gemeinsamen Schwerpunkt (das *Baryzentrum*) „kreisen“.

In dieser Arbeit werden Himmelskörper ausschliesslich durch ihre Orbit-Eigenschaften charakterisiert.

Himmelskörper bewegen sich und sind daher als zeitabhängige Objekte einzuführen, zumindest was ihre Position und ihre Geschwindigkeit anbelangt. Weitere Eigenschaften können sowohl konstanter als auch zeitabhängiger Natur sein. Je nach gewünschter Genauigkeit wären sie entsprechend zu modellieren.

Nur die einfachsten *Bewegungsgleichungen* liefern Bahnellipsen als Lösungen. Bei geringen Störungen können ebenfalls noch mittlere Ellipsen zur Beschreibung der Bewegung dienen, deren Elemente nun aber zeitabhängig zu formulieren sind.

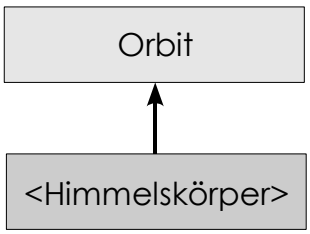

Abbildung 16: Das Objekt Himmelskörper wird vom Objekt Orbit abgeleitet und verfügt dadurch über alle Kenngrössen, die seine Umlaufbahn bestimmen

Planeten

Planeten und *Kometen* werden durch ihre Orbits festgelegt. Zu diesem Zweck greifen sie auf Objekte zurück, die Orbit-Elemente für gewünschte Zeitpunkte bereitstellen können. Die Umrechnung in Orts- und Geschwindigkeitsvektor erfolgt im Sinne einer *Keplerbahn* mit der Sonne als *Zentralkörper*.

Durch die Einführung eigener Objekte für die *Bahndaten* gewinnt man an Flexibilität. Sowohl mittlere Bahnelemente (s. MEEUS) mit ihren Zeitabhängigkeiten können verwendet werden, als auch beispielsweise hochgenaue, tabellierte *Ephemeriden* oder Verfahren zur numerischen Integration. Entscheidend ist, dass zum gewünschten Zeitpunkt die Bahninformation in Form der sechs Keplerelemente vorliegt.

Ausgehend von sogenannten mittleren Bahnelementen zur jeweils aktuellen Epoche werden die Orbits der Planeten Merkur bis Neptun berechnet [Meeus]. *Mittlere* Bahnelemente geben im Mittel über längere Zeiträume die Bahnbewegungen recht gut wieder. Höchsten Ansprüchen an die Positionsgenauigkeit können sie jedoch nicht genügen.

Für den äussersten bekannten Planeten, *Pluto*, sind keine brauchbaren mittleren Bahnelemente verfügbar. Er ist erst 1930 entdeckt worden und sein Umlauf um die Sonne dauert mehr als 250 Jahre. Zu seiner Modellierung kommen sogenannte *oskulierende Bahnelemente* zum Einsatz [Astronomical Almanac], die nur für jeweils kurze Zeiträume die Bahnpositionen recht genau beschreiben. An ein kurzes, gut bestimmtes Bahnstück wird praktisch eine komplette Ellipse bestens angeschmiegt.

Die grossen Halbachsen der Planeten bleiben gleich und signalisieren dadurch eine gleichbleibende Bahnenergie. Die anderen Elemente e, i, ω, Ω und M werden mit kubischen Funktionen approximiert.

Am Beispiel *Mars* sei die Vorgehensweise erläutert:

Die grosse Halbachse beträgt 1.5236883 Astronomische Einheiten (zur Umrechnung in Meter siehe Astronomische Konstante, S.7). Die Koeffizienten der kubischen Funktionen für die weiteren *Bahnelemente* lauten [Meeus]:

Element	Konst. Term k_0	Lin. Term k_1	Quadrat. Term k_2	Kub. Term k_3
e	0.0933129	0.000092064	-0.000000077	0.0
i	1.850333	-0.000675	0.0000126	0.0
ω	285.431761	1.0697667	0.0001313	0.00000414
O≙Ω	48.786442	0.7709917	-0.0000014	-0.00000533
M	293.737334	19141.69551	0.0003107	0.0

Tabelle 4: Mittlere Bahnelemente des Planeten Mars und ihre Zeitabhängigkeiten

Mit der Zeitdifferenz T = (t - J1900)/36525 in julianischen Jahrhunderten seit *J1900*=2415020.0 (31.12.1899, 12 Uhr) ergeben sich die aktuellen Bahnelemente zu:

Element $_{aktuell} = k_0 + k_1*T + k_2*T^2 + k_3*T^3$

bzw. unter Ausnutzung des *Horner-Schemas* (Einsparung von Multiplikationen)

Element $_{aktuell} = ((k_3*T + k_2)*T + k_1)*T + k_0$

Die aktuell ermittelten Bahnelemente beschreiben die Planetenpositionen im jeweils aktuellen heliozentrischen *Ekliptiksystem*. Somit sind die Präzessionseinflüsse bereits in den Koeffizienten k_i enthalten.

```
/**
 *PlanetData.java
 *
 * read orbit elements for planets from server or local machine
 * the file "data/planet.dat" is supplied as URL or as String
 *
 * @see OrbitElements
 * @see FileOfStrings
 * @see PlanetNames
 * @see MyLib
 *
 * @author Dieter Egger
 * @version  1.3.1    2001/01/10
 *
 */

import java.applet.*;
import java.awt.*;
import java.util.StringTokenizer;
import java.net.*;

import mie.*;

public class PlanetData extends OrbitElements
{
  FileOfStrings planetFile;
  final static int linesPerPlanet = 7;
  PlanetNames planetNames;
  String planetSemiA;
  String planetE;
  String planetI;
  String planetW;
  String planetO;
  String planetM;
```

```
public final static double EpochPluto = 2450640.5;

public PlanetData (URL alpha)
{
  planetFile = new FileOfStrings (alpha);
  planetNames = new PlanetNames (planetFile);
}

public PlanetData (String fn)
{
  planetFile = new FileOfStrings (fn);
  planetNames = new PlanetNames (planetFile);
}

public int numOfPlanets () { return planetNames.size(); }

public int indexOf (String name)
{
  for (int i=0; i<numOfPlanets(); i++)
    if (name.equals ((String)planetNames.elementAt (i)))
        return i;
  return -1;
}

public String getName (int i)
{
  if (i<0 || i>=numOfPlanets())
      return new String ("Out of bounds!");
  else return (String)planetNames.elementAt (i);
}

public String getSemiA (int i)
{
  int j = i*linesPerPlanet + 1;
  if (j<0 || j>=planetFile.size())
      return new String ("Out of bounds!");
  else return (String)planetFile.elementAt (j);
}

public String getE (int i)
{
  int j = i*linesPerPlanet + 2;
  if (j<0 || j>=planetFile.size())
      return new String ("Out of bounds!");
  else return (String)planetFile.elementAt (j);
}

public String getI (int i)
{
  int j = i*linesPerPlanet + 3;
  if (j<0 || j>=planetFile.size())
      return new String ("Out of bounds!");
  else return (String)planetFile.elementAt (j);
}

public String getW (int i)
{
  int j = i*linesPerPlanet + 4;
  if (j<0 || j>=planetFile.size())
      return new String ("Out of bounds!");
```

```
  else return (String)planetFile.elementAt (j);
}

public String getO (int i)
{
  int j = i*linesPerPlanet + 5;
  if (j<0 || j>=planetFile.size())
      return new String ("Out of bounds!");
  else return (String)planetFile.elementAt (j);
}

public String getM (int i)
{
  int j = i*linesPerPlanet + 6;
  if (j<0 || j>=planetFile.size())
      return new String ("Out of bounds!");
  else return (String)planetFile.elementAt (j);
}

public double elementValue (String s, double dt)
{
  double [] v = new double [4];
  StringTokenizer sd = new StringTokenizer (s);

  if (sd.countTokens()<4) for (int i=0; i<4; i++) v[i]=0.0;

  int j=0;
  while (sd.hasMoreTokens() && j<4)
  {
    v [j++] = (new Double(sd.nextToken())).doubleValue ();
  }
  return MyLib.horner (dt, v[0], v[1], v[2], v[3]);
}

public void setOrbitElements (String name, JulianTime t)
{
  int j = indexOf (name);
  double dt;
  double A, e, i, w, O, M;

  if (j == -1) return;

  if ("Pluto".equals (name))
     dt = (t.julianDate () - EpochPluto)/Aconst.JulianCentury;
  else dt =
        (t.julianDate () - Aconst.JD19)/Aconst.JulianCentury;

  A = elementValue (getSemiA (j), dt);
  e = elementValue (getE (j), dt);
  i = elementValue (getI (j), dt);
  i *= Aconst.DegreeToRadian;
  w = elementValue (getW (j), dt);
  w = MyLib.mod360 (w);
  O = elementValue (getO (j), dt);
  O = MyLib.mod360 (O);
  M = elementValue (getM (j), dt);

  if ("Moon".equals (name)) A *= 1000.0;
  else A *= Aconst.AstronomicalUnit;
```

```
    M -= (O + w);
    M  = MyLib.mod360 (M) * Aconst.DegreeToRadian;
    O *= Aconst.DegreeToRadian;
    w *= Aconst.DegreeToRadian;
    setAeiwOM (A, e, i, w, O, M);
  }
}
```

Diese Daten fliessen in den PlanetOrbit ein, der von RefOrbit abgeleitet wird:

```
/**
 *PlanetOrbit.java
 *
 * define RefOrbit using mean elements from Meeus
 * (planet orbits and orbit of the Moon)
 *
 * @see RefOrbit
 * @see PlanetData
 * @see OrbitElements
 *
 * @author Dieter Egger
 * @version  1.3.1    2001/01/10
 *
 */

import java.net.*;

import mie.*;

public class PlanetOrbit extends RefOrbit
{
  private PlanetData pd;

  public PlanetOrbit (URL alpha)
  {
    pd = new PlanetData (alpha);
  }

  public PlanetOrbit (String fn)
  {
    pd = new PlanetData (fn);
  }

  public PlanetOrbit (URL alpha, String name, JulianTime t)
  {
    pd = new PlanetData (alpha);
    setPlanetOrbit (name, t);
  }

  public PlanetOrbit (String fn, String name, JulianTime t)
  {
    pd = new PlanetData (fn);
    setPlanetOrbit (name, t);
  }

  public PlanetData getPlanetData () { return pd; }

  public void setPlanetOrbit (int index, JulianTime t)
  {
    setPlanetOrbit (pd.getName (index), t);
```

```
  }

  public void setPlanetOrbit (String name, JulianTime t)
  {
    setOrbitEpoch (t);
    setJulianTimeRS (t);
    if ("Moon".equals (name))
    {
      setGM (Aconst.GM_Earth);
      setIdRS ("GE");
    }
    else
    {
      setGM (Aconst.GM_Sun);
      setIdRS ("HE");
    }
    pd.setOrbitElements (name, t);
    setOrbitElements (pd);
    if ("Earth".equals (name))
        setPosVel (improveEarth (), getVel ());
  }


  private Vector3D improveEarth ()
  /*-----------------------------
    Verbesserung der Erdposition (siehe MEEUS)
    ------------------------------------------*/
  {
     double a, b, c, d, e, h;
     double laenge, theta, radius;

     double t = (julianDate () - Aconst.JD19)
                     / Aconst.JulianCentury;
     Vector3D rlt = new Vector3D (getPos ().toPolar ());

     a = MyLib.mod360 (153.23 + 22518.7541*t)
           *Aconst.DegreeToRadian;
     b = MyLib.mod360 (216.57 + 45037.5082*t)
           *Aconst.DegreeToRadian;
     c = MyLib.mod360 (312.69 + 32964.3577*t)
           *Aconst.DegreeToRadian;
     d = MyLib.mod360 (350.74 + (-0.00144*t + 445267.1142)*t)
           *Aconst.DegreeToRadian;
     e = MyLib.mod360 (231.19 + 20.20*t)
           *Aconst.DegreeToRadian;
     h = MyLib.mod360 (353.40 + 65928.7155*t)
           *Aconst.DegreeToRadian;

     rlt.x[1]+=(  0.00134 * Math.cos (a) + 0.00154 * Math.cos (b)
            + 0.00200 * Math.cos (c) + 0.00179 * Math.sin (d)
            + 0.00178 * Math.sin (e) ) * Aconst.DegreeToRadian;

     rlt.x[0]+=(0.00000543*Math.sin(a) + 0.00001575 * Math.sin (b)
            + 0.00001627 * Math.sin (c) + 0.00003076 * Math.cos (d)
            + 0.00000927 * Math.sin (h) ) * Aconst.AstronomicalUnit;

     return rlt.toVector ();
  }
}
```

Kometen

Anstelle der Grossen Halbachse *a* ist bei Kometen meist der Wert für die *Periheldistanz q* und anstelle der Mittleren Anomalie *M* meist der Zeitpunkt τ des *Periheldurchgangs* tabelliert. Beide lassen sich leicht in die klassischen Bahnelemente überführen:

$$a = \frac{q}{(1-e)}$$

mit e Exzentrizität ($0 \leq e < 1$), und

$$M(t) = (t-\tau)n \quad \text{, wobei} \quad n = \sqrt{\frac{GM}{a^3}}$$

die mittlere Bewegung in Radian pro Sekunde bedeutet.

Bis auf diesen kleinen Unterschied entsprechen sich PlanetData und CometData:

```
/**
 *CometData.java
 *
 * read orbit elements for comets from server or local machine
 * the file "data/comet.dat" is supplied as URL or as String
 *
 * @see OrbitElements
 * @see FileOfStrings
 * @see CometNames
 * @see MyLib
 *
 * @author Dieter Egger
 * @version  1.3.1    2001/01/10
 *
 */

import java.applet.*;
import java.awt.*;
import java.net.*;

import mie.*;

public class CometData extends OrbitElements
{
  FileOfStrings cometFile;
  final static int linesPerComet = 7;
  CometNames cometNames;
  String cometSemiA;
  String cometE;
  String cometI;
  String cometW;
  String cometO;
  String cometM;

  public CometData (URL alpha)
  {
    cometFile = new FileOfStrings (alpha);
```

```
    cometNames = new CometNames (cometFile);
  }

  public CometData (String fn)
  {
    cometFile = new FileOfStrings (fn);
    cometNames = new CometNames (cometFile);
  }

  public int numOfComets () { return cometNames.size(); }

  public int indexOf (String name)
  {
    for (int i=0; i<numOfComets(); i++)
      if (name.equals ((String)cometNames.elementAt (i))) return i;
    return -1;
  }

  public String getName (int i)
  {
    if (i<0 || i>=numOfComets())
      return new String ("Out of bounds!");
    else return (String)cometNames.elementAt (i);
  }

  public String getSemiA (int i)
  {
    int j = i*linesPerComet + 1;
    if (j<0 || j>=cometFile.size())
      return new String ("Out of bounds!");
    else return (String)cometFile.elementAt (j);
  }

  public String getE (int i)
  {
    int j = i*linesPerComet + 2;
    if (j<0 || j>=cometFile.size())
      return new String ("Out of bounds!");
    else return (String)cometFile.elementAt (j);
  }

  public String getI (int i)
  {
    int j = i*linesPerComet + 3;
    if (j<0 || j>=cometFile.size())
      return new String ("Out of bounds!");
    else return (String)cometFile.elementAt (j);
  }

  public String getW (int i)
  {
    int j = i*linesPerComet + 4;
    if (j<0 || j>=cometFile.size())
      return new String ("Out of bounds!");
    else return (String)cometFile.elementAt (j);
  }

  public String getO (int i)
  {
    int j = i*linesPerComet + 5;
```

```
    if (j<0 || j>=cometFile.size())
      return new String ("Out of bounds!");
    else return (String)cometFile.elementAt (j);
  }

  public String getM (int i)
  {
    int j = i*linesPerComet + 6;
    if (j<0 || j>=cometFile.size())
      return new String ("Out of bounds!");
    else return (String)cometFile.elementAt (j);
  }

  public double elementValue (String s)
  {
    double v = 0.0;
    try
    {
      v = (new Double(s)).doubleValue ();
    }
    catch (NumberFormatException nfe) {}

    return v;
  }

  public void setOrbitElements (String name, JulianTime t)
  {
    int j = indexOf (name);
    double dt;
    double A, e, i, w, O, M;

    if (j == -1) return;


    A = elementValue (getSemiA (j));// Periapsis distance
    e = elementValue (getE (j));
    i = elementValue (getI (j));
    i *= Aconst.DegreeToRadian;
    w = elementValue (getW (j));
    w = MyLib.mod360 (w);
      w *= Aconst.DegreeToRadian;
    O = elementValue (getO (j));
    O = MyLib.mod360 (O);
      O *= Aconst.DegreeToRadian;
    M = elementValue (getM (j));
      // Periapsis crossing time as julian day + frac.

    dt = (t.julianDate () - M)*Aconst.SolarDay;
      // difference in seconds

    if (e < 1.0) A /= (1.0 - e);      // now semi-major axis
    A *= Aconst.AstronomicalUnit;
   M  = dt;   // time difference to periapsis crossing time [s]
    setElements (A, e, i, w, O, M);
      // Pseudo-Elements because of M = time difference
  }

  public String toString ()
  {
    return "CometData A "
```

```
        +MyLib.display(getA()/Aconst.AstronomicalUnit,6)+
        "\t e "+MyLib.display(getE(),6)+
        "\t i "+MyLib.display(getI()*Aconst.Radian2Degree,4)+
        "\t w "+MyLib.display(getW()*Aconst.Radian2Degree,4)+
        "\t O "+MyLib.display(getO()*Aconst.Radian2Degree,4)+
        "\t M "+MyLib.display(getM()*Aconst.Radian2Degree,6);
  }
}
```

CometOrbit macht Gebrauch von diesen CometData und ist selbst von RefOrbit abgeleitet:

```
/**
 *CometOrbit.java
 *
 * define RefOrbit using astrometric elements for epoch J2000
 * (comet orbits)
 *
 * @see RefOrbit
 * @see CometData
 * @see OrbitElements
 *
 * @author Dieter Egger
 * @version  1.3.1    2001/01/10
 *
 */

import java.net.*;

import mie.*;

public class CometOrbit extends RefOrbit
{
  private CometData pd;

  public CometOrbit (URL alpha)
  {
    pd = new CometData (alpha);
  }

  public CometOrbit (String fn)
  {
    pd = new CometData (fn);
  }

  public CometOrbit (URL alpha, String name, JulianTime t)
  {
    pd = new CometData (alpha);
    setCometOrbit (name, t);
  }

  public CometOrbit (String fn, String name, JulianTime t)
  {
    pd = new CometData (fn);
    setCometOrbit (name, t);
  }

  public CometData getCometData () { return pd; }

  public void setCometOrbit (int index, JulianTime t)
```

```
  {
    setCometOrbit (pd.getName (index), t);
  }

  public void setCometOrbit (String name, JulianTime t)
  {
    setOrbitEpoch (t);
    setJulianTimeRS (new JulianTime (Aconst.J2000));
    setIdRS ("HE");
    setGM (Aconst.GM_Sun);

    pd.setOrbitElements (name, t);
    double a = pd.getA();
    double n = Math.sqrt (getGM()/(a*a*a));
    double ma = pd.getM()*n;
    OrbitElements oe = new OrbitElements
      (pd.getA(), pd.getE(), pd.getI(), pd.getW(), pd.getO(), ma);
    setOrbitElements (oe);
  }
}
```

Satelliten

Der Einfachheit halber werden *Satelliten* ausschliesslich durch ihre Orbits beschrieben. Die Bahnelemente variieren recht schnell, so dass auf häufig aktualisierte Two Line Elements, eine kompakte Formulierung der Bahnelemente in zwei Zeilen, zurückgegriffen wird.

Abweichungen von reinen *Keplerbahnen* ergeben sich bei Satelliten wegen einer Reihe von *Bahnstörungen*:

1) Anisotropie des Gravitationsfeldes der Erde (Hauptterm = Polabplattung)
2) Gravitationsfelder von Sonne, Mond, Planeten und Asteroiden
3) Reibungswiderstand der Hochatmosphäre
4) Strahlungsdruck der Sonne
5) Strahlungsdruck der Erde (hauptsächlich Infrarot)
6) relativistischen Effekten

Eine *präzise Bahnbestimmung* ist nur mit Hilfe der numerischen Integration der nach-Newtonschen Bewegungsgleichungen zu erreichen. Sie ist sehr rechenaufwendig. Und sie stellt eine grosse Herausforderung an die Theorie der Satellitenbahnen dar [Schneider, Seeber].

Um dennoch mit wenig Aufwand brauchbare *Satellitenbahnen* berechnen zu können, bieten sich die bereits erwähnten *Two-Line-Elements* an, die mehrmals pro Woche für eine Vielzahl von Satelliten neu berechnet und per Internet zur Verfügung gestellt werden.

Am Beispiel des Hubble Space Teleskop (HST) (siehe Text 2) werden Aufbau und Bedeutung näher erläutert:

```
HST
1 20580U 90037B   98273.16532806  .00001888  00000-0  18987-3 0  1470
2 20580  28.4686 277.6154 0013928  39.8809 320.2796 14.86982652263060
```

Bedeutung der Zeile 1:

```
Spalte     Beschreibung
-----------------------
 01-01     Zeilennummer der Two-Line-Elements
 03-07     Satellitennummer
 10-11     Internationale Kennzeichnung
           (die beiden letzten Ziffern des Startjahres)
 12-14     Internationale Kennzeichnung (Startnummer des Jahres)
 15-17     Internationale Kennzeichnung (Stückbezeichnung)
 19-20     Epoche: Jahr (die beiden letzten Ziffern des Jahres)
 21-32     Epoche: Tag des Jahres mit Tagesbruchteil
 34-43     Erste Zeitableitung der mittleren Bewegung (n-punkt)
         oder Ballistischer Koeffizient (je nach Ephemeriden-Typ)
 45-52     Zweite Zeitableitung der mittleren Bewegung (n-punkt-punkt)
           (Dezimalpunkt wird unterstellt; leer falls unbekannt)
 54-61     BSTAR Reibungsterm bei GP4-Störungstheorie.
           Sonst Strahlungsdruck-Koeffizient.(Dez.punkt unterstellt)
 63-63     Ephemeriden-Typ
 65-68     Nummerierung der Two-Line-Elements
 69-69     Check Summe (Modulo 10)
           (Buchstaben, Blanks, Punkte = 0; Minuszeichen = 1; Pluszeichen
= 2)
```

Bedeutung der Zeile 2:

```
Spalte     Beschreibung
-----------------------
 01-01     Zeilennummer der Two-Line-Elements
 03-07     Satellitennummer
 09-16     Inklination [Grad]
 18-25     Rektaszension aufsteigender Knoten (Knotenlage) [Grad]
 27-33     Exzentrizität (Dezimalpunkt wird unterstellt)
 35-42     Perigäumslage [Grad]
 44-51     Mittlere Anomalie [Grad]
 53-63     "Mittlere Bewegung" [Umläufe/Tag]
 64-68     Nummer des aktuellen Umlaufs zur Epoche [Umläufe]
 69-69     Check Summe (Modulo 10)
```

Text 2: Two Line Elements am Beispiel des Hubble Space Teleskop und ihre Bedeutung

Programmtechnisch werden die TLEs in Listen der beiden Zeilen und der Namen aufgespalten:

```
/**
 *TleData.java
 *
 * read two line elements for satellites from server or local machine
 * the files "tle/*.txt" are chosen by URL or String
 *
 * @see FileOfStrings
 * @see TleNames
 * @see TleLine1
 * @see TleLine2
 *
 * @author Dieter Egger
 * @version  1.3.1    2001/01/10
 *
 */
```

```
import java.applet.*;
import java.awt.*;
import java.util.StringTokenizer;
import java.net.*;

import mie.*;

public class TleData extends Object
{
  FileOfStrings tleFile;
  TleNames tlenames;
  TleLine1 tleline1;
  TleLine2 tleline2;

  public TleData (URL alpha)
  {
    tleFile = new FileOfStrings (alpha);
    tlenames = new TleNames (tleFile);
    tleline1 = new TleLine1 (tleFile);
    tleline2 = new TleLine2 (tleFile);
  }

  public TleData (String fn)
  {
    tleFile = new FileOfStrings (fn);
    tlenames = new TleNames (tleFile);
    tleline1 = new TleLine1 (tleFile);
    tleline2 = new TleLine2 (tleFile);
  }

  public int size () { return tlenames.size(); }

  public int indexOf (String name)
  {
    for (int i=0; i<size(); i++)
      if (name.equals ((String)tlenames.elementAt (i))) return i;
    return -1;
  }

  public String getName (int i)
  {
    if (i<0 || i>=size()) return new String ("Out of bounds!");
    else return (String)tlenames.elementAt (i);
  }

  public String getLine1 (int i)
  {
    if (i<0 || i>=size()) return new String ("Out of bounds!");
    else return (String)tleline1.elementAt (i);
  }

  public String getLine2 (int i)
  {
    if (i<0 || i>=size()) return new String ("Out of bounds!");
    else return (String)tleline2.elementAt (i);
  }

}
```

Diese Daten fliessen in den TleOrbit ein:

```
/**
 *TleOrbit.java
 *
 * define RefOrbit using two line elements
 * (satellite orbits)
 *
 * @see RefOrbit
 * @see TleData
 * @see OrbitElements
 *
 * @author Dieter Egger
 * @version  1.3.1    2001/01/10
 *
 */

import java.applet.*;
import java.awt.*;
import java.util.StringTokenizer;
import java.net.*;

import mie.*;

public class TleOrbit extends RefOrbit
{
  private TleData tleData;
  private String actualName;
  private String actualLine1;
  private String actualLine2;
  private int actualIndex = -1;
  private double an, anp, anr;
    // mean motion, dot (mean motion), nr of rev.
  private boolean valid = false;

  public TleOrbit (URL alpha)
  {
    tleData = new TleData (alpha);
  }

  public TleOrbit (String fn)
  {
    tleData = new TleData (fn);
  }

  public int getActualIndex () { return actualIndex; }
  public String getActualName () { return actualName; }
  public String getActualLine1 () { return actualLine1; }
  public String getActualLine2 () { return actualLine2; }

  public TleData getTleData () { return tleData; }

  public boolean isValid () { return valid; }

  double getDouble (String s)
  {
    Double d = new Double (s);
    return d.doubleValue ();
  }

  int getInt (String s) { return Integer.parseInt(s); }
```

```
public void setTleOrbit (int i)
/*---------------------------------------------
  extract orbit information for ith satellite in tle data
  ---------------------------------------------*/
{
  int year;
  double jdfr, m_b;
  double bA, be, bi, bw, bO, bM;

  if (i == actualIndex) return;
  if (i<0 || i>tleData.size())
    { valid = false; actualIndex = -1; return; }

  actualName = new String (tleData.getName (i));
  actualLine1 = new String (tleData.getLine1 (i));
  actualLine2 = new String (tleData.getLine2 (i));

  year = getInt (actualLine1.substring ( 18,  20)) + 2000;
  JulianTime dat = new JulianTime (""+year+" 1 0", "00:00:00");

  jdfr = getDouble (actualLine1.substring ( 20, 32));

  dat.plus (new JulianTime (jdfr));

  setOrbitEpoch (dat);

  anp = getDouble (actualLine1.substring ( 33, 43))
        * Aconst.TwoPi;

  bi = getDouble (actualLine2.substring (  8, 16))
        * Aconst.DegreeToRadian;
  bO = getDouble (actualLine2.substring ( 17, 25))
        * Aconst.DegreeToRadian;
  be = getDouble (actualLine2.substring ( 26, 33)) * 1.0e-7;
  bw = getDouble (actualLine2.substring ( 34, 42))
        * Aconst.DegreeToRadian;
  bM = getDouble (actualLine2.substring ( 43, 51))
        * Aconst.DegreeToRadian;

  an = getDouble (actualLine2.substring ( 52, 63))
        * Aconst.TwoPi;
  m_b = an / Aconst.SolarDay;
  bA = Math.exp (Math.log (getGM()/(m_b * m_b)) / 3.0);

  anr = getDouble (actualLine2.substring ( 63, 68));
  setOrbitElements (new OrbitElements (bA, be, bi, bw, bO, bM));
  setIdRS ("SF");
  setJulianTimeRS (dat);
  actualIndex = i;
  valid = true;
}

public void setTleOrbit (JulianTime tnew)
{
  if (!isValid()) return;
  JulianTime delta = new JulianTime (tnew);
  delta.minus (this);
  double dt = delta.julianDate ();
  if (Math.abs (dt) < Aconst.EpsilonDouble*4.0) return;
  double deltaM = (an )*dt;
```

```
    OrbitElements orbel = new OrbitElements (getOrbitElements ());
    orbel.increaseMeanAnomaly (deltaM);
    setOrbitElements (orbel);
    setOrbitEpoch (tnew);
    setJulianTimeRS (tnew);
  }
}
```

Sterne

Sterne sind weit ausserhalb unseres Sonnensystems beheimatet und behalten auch über lange Zeiträume ihre relativen Positionen zu anderen Sternen bei. Daher rührt die alte Bezeichnung Fix-Sterne. Sie scheinen sich nur deshalb zu bewegen, weil die Erde rotiert und wir sie daher im Osten auf- und im Westen untergehen sehen.

Genauere Beobachtungen zeigen allerdings, dass auch Sterne einer Dynamik unterliegen (*Stellardynamik*). Sie führt zwar meist nur zu geringen Bewegungen im Bereich von wenigen Bogensekunden pro Jahrhundert, ist aber für genaue astronomische Ortsbestimmungen nicht vernachlässigbar.

Sternpositionen und *Eigenbewegungen* werden in *Sternkatalogen* verzeichnet. Ihre Werte beziehen sich beim *FK5* (5. *Fundamentalkatalog*) auf das mittlere, baryzentrische Äquatorsystem zur Epoche J2000, beim HIPPARCOS-Katalog auf die Epoche J1991.25. Mit den entsprechenden Transformationen können sie leicht in aktuelle Referenzsysteme umgerechnet werden.

Der FK5 beschreibt die ca. 1500 *Fundamentalsterne* mit ihren Rektaszensionen und Deklinationen, den zugehörigen Eigenbewegungen und bei nahen Sternen auch mit ihren Parallaxen. Die Entfernungseinheit 1 *Parsec* besagt, dass der Erdbahnradius aus dieser Distanz unter dem Winkel von 1 Bogensekunde gesehen wird. Das sind etwas über 3 Lichtjahre, einer weiteren, in der Astronomie gebräuchlichen Entfernungseinheit, die die Lichtstrecke während eines Jahres kennzeichnet. Ein Stern, der unter der *Parallaxe* 0.3“ erscheint, ist mithin 3.3 Parsec bzw. gut 10 Lichtjahre entfernt.

Programmtechnisch wird beispielsweise ein Fk5-Stern mit der Objektstruktur

```
/**
 *Fk5Star.java
 *
 * define object Fk5Star
 * (actual data contained in corresponding file(s))
 * the epoch of reference is preferably J2000
 *
 * @see Fk5StarData
 *
 * @author Dieter Egger
 * @version  1.3.1    2001/01/10
 *
 */

public class Fk5Star extends Object
{
```

```
  private int number;                 // FK5-number
  private int magnitude;              // mag * 100
  private String spectralClass; // O B A F G K M + number
  private int rightAscension;         // seconds of time * 1000
  private int pmRightAscension;
    // seconds of time * 1000 / JulianCentury
  private int declination;      // seconds of arc * 100
  private int pmDeclination;
    // seconds of arc * 100 / JulianCentury
  private int parallax;               // seconds of arc * 1000
  private int radialVelocity;         // km/s * 10
  public int x, y;                    // for drawing purposes only

  public Fk5Star ()
  {
    number = 0;
    magnitude = 0;
    spectralClass = "F5";
    rightAscension = 0;
    pmRightAscension = 0;
    declination = 0;
    pmDeclination = 0;
    parallax = 0;
    radialVelocity = 0;
  }

  public void setNumber (String s)
    { number = Integer.parseInt(s); }
  public void setMagnitude (String s)
    { magnitude = Integer.parseInt(s); }
  public void setSpectralClass (String s)
    { spectralClass = new String (s); }
  public void setRightAscension (String s)
    { rightAscension = Integer.parseInt(s); }
  public void setRightAscension (int ra)  { rightAscension = ra; }
  public void setPmRightAscension (String s)
    { pmRightAscension = Integer.parseInt(s); }
  public void setDeclination (String s)
    { declination = Integer.parseInt(s); }
  public void setDeclination (int dec)     { declination = dec; }
  public void setPmDeclination (String s)
    { pmDeclination = Integer.parseInt(s); }
  public void setParallax (String s)
    { parallax = Integer.parseInt(s); }
  public void setRadialVelocity (String s)
    { radialVelocity = Integer.parseInt(s); }

  public int getNumber ()             { return number; }
  public int getMagnitude ()               { return magnitude; }
  public String getSpectralClass ()   { return spectralClass; }
  public int getRightAscension ()     { return rightAscension; }
  public int getPmRightAscension () { return pmRightAscension; }
  public int getDeclination ()        { return declination; }
  public int getPmDeclination ()           { return pmDeclination; }
  public int getParallax ()                { return parallax; }
  public int getRadialVelocity ()          { return radialVelocity; }

  public String toString () { return ("Fk5Star #"+number); }
};
```

über seine Daten, die aus einer Datei stammen

```
/**
 *Fk5StarData.java
 *
 * read data for Fk5Star from server or local machine
 * the file "data/fk5.txt", "data/fk5sup.txt"
 * or "data/star2000.txt" is supplied as URL or as String
 *
 * @see Fk5Star
 * @see FileOfStrings
 *
 * @author Dieter Egger
 * @version  1.3.1    2001/01/10
 *
 */

import java.applet.*;
import java.awt.*;
import java.util.StringTokenizer;
import java.net.*;

public class Fk5StarData extends FileOfStrings
{
  public Fk5StarData () {}

  public Fk5StarData (URL alpha)
  {
    super (alpha);
  }

  public Fk5StarData (String fn)
  {
    super (fn);
  }

  public void set (URL alpha) { getData (alpha); }
  public void set (String fn) { getData (fn); }

  public String line (int i)
  {
    if (i<0 || i>=size()) return new String ("Out of bounds");
    else
    return (String) elementAt (i);
  }

  public boolean getFk5Star (int i, Fk5Star st)
  {
    String s = line (i);
    StringTokenizer sd = new StringTokenizer (s);

    int nrtok = sd.countTokens();
    if (sd.countTokens()<9) return false;
    st.setNumber (sd.nextToken());
    st.setMagnitude (sd.nextToken());
    st.setSpectralClass (sd.nextToken());

    if (nrtok == 9)
      // one value for right ascension (1000*seconds of time)
      st.setRightAscension (sd.nextToken());
```

```
    else if (nrtok == 11) // two values: h and minutes of time
    {
      int ra = Integer.parseInt (sd.nextToken())*60
               + Integer.parseInt (sd.nextToken());
      ra*=60000;
      st.setRightAscension (ra);
    }
    else if (nrtok == 13)
      // three values: h m and seconds of time
    {
      int ra = Integer.parseInt (sd.nextToken())*3600
               + Integer.parseInt (sd.nextToken())*60
        + Integer.parseInt (sd.nextToken());
      ra*=1000;
      st.setRightAscension (ra);
    }
    st.setPmRightAscension (sd.nextToken());

    if (nrtok == 9)
      // one value for right ascension (100*seconds of arc)
      st.setDeclination (sd.nextToken());
    else if (nrtok == 11) // two values: deg and minutes of arc
    {
      int deg = Integer.parseInt (sd.nextToken());
      int min = Integer.parseInt (sd.nextToken());
      boolean neg = deg<0 || min<0;
      int dec = Math.abs(deg)*3600+Math.abs(min)*60;
      if (neg) dec = -dec;
      dec*=100;
      st.setDeclination (dec);
    }
    else if (nrtok == 13)
    // three values: deg m and seconds of arc
    {
      int deg = Integer.parseInt (sd.nextToken());
      int min = Integer.parseInt (sd.nextToken());
      int sec = Integer.parseInt (sd.nextToken());
      boolean neg = deg<0 || min<0 || sec<0;
      int dec = Math.abs(deg)*3600+Math.abs(min)*60+Math.abs(sec);
      if (neg) dec = -dec;
      dec*=100;
      st.setDeclination (dec);
    }
    st.setPmDeclination (sd.nextToken());
    st.setParallax (sd.nextToken());
    st.setRadialVelocity (sd.nextToken());
    return true;
  }

  public Fk5Star getFk5Star (int snumber)
  {
    Fk5Star st = new Fk5Star();

    for (int i=0; i<size(); i++)
    {
      if (!getFk5Star (i, st)) continue;
      if (st.getNumber () == snumber) return st;
      if (i > snumber) break;
    }
    return new Fk5Star();
```

```
  }
}
```

als kompletter Katalog im Speicher gehalten, wobei jeweils die Grenzmagnituden zur Informationsreduktion vorgegeben werden können:

```
/**
 *Fk5StarCatalog.java
 *
 * collect desired Fk5Star objects in a Vector
 * filter criterion is the magnitude range
 *
 * @see Fk5Star
 * @see Fk5StarData
 *
 * @author Dieter Egger
 * @version  1.3.1    2001/01/10
 *
 */

import java.awt.*;
import java.util.*;

public class Fk5StarCatalog extends Vector
{
  public Fk5StarCatalog () { super (10,10); }

  public Fk5StarCatalog
           (Fk5StarData sd, int magStart, int magStop)
  { buildCatalog (sd, magStart, magStop); }

  public void buildCatalog
           (Fk5StarData sd, int magStart, int magStop)
  {
    int mag;
    int unten =  magStart, oben = magStop;
    if (unten > oben) { unten = magStop; oben = magStart; }
    if (unten == oben) { unten = -1000; oben = 1000; }

    if (size() != 0) removeAllElements ();
    for (int i=0; i<sd.size(); i++)
    {
      Fk5Star fk5Star = new Fk5Star();
      if (sd.getFk5Star (i, fk5Star) == false) break;
      mag = fk5Star.getMagnitude ();
      if (mag > unten && mag <= oben)
      {
        addElement (fk5Star);
      }
    }
  }

  public Fk5Star getFk5Star (int fk5nr)
  {
    Fk5Star star, foundStar = null;
    for (int i=0; i<size(); i++)
    {
      star = (Fk5Star) elementAt (i);
      if (star.getNumber () == fk5nr) { foundStar = star; break; }
      else
```

```
      if (star.getNumber () > fk5nr) break;
    }
    return foundStar;
  }
}
```

Transformationen

Ein Koordinatensystem wird relativ zu einem anderen Koordinatensystem festgelegt. Durch eine Transformation können Koordinaten eines Vektors, gegeben im ersten Koordinatensystem, in Koordinaten bezüglich des zweiten Koordinatensystems umgerechnet werden. Im allgemeinen sind dazu Translationen, Rotationen und eventuell Spiegelungen erforderlich.

Vektoren

Gemeint sind dreidimensionale Vektoren, die geeignet sind, einen Punkt im Raum oder eine gerichtete Geschwindigkeit im Raum festzulegen.

$$\boldsymbol{x} = \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} \quad oder \quad \dot{\boldsymbol{x}} = \begin{pmatrix} \dot{x}_1 \\ \dot{x}_2 \\ \dot{x}_3 \end{pmatrix}$$

```
/**
 *Vector3D.java
 *
 *double array representing 3-dim-vector
 *
 * @author Dieter Egger
 * @version  1.3.1    2001/01/10
 *
 */

package mie;

public class Vector3D extends Object       // Object 3-Vector
{
  public double x [];                            // 3 double values

  // Constructors

  public Vector3D ()
  { x = new double [3]; x[0] = x[1] = x[2] = 0.0; }

  public Vector3D (double x1, double x2, double x3)
  // 3-Vector with Initialisation
  { x = new double [3]; setX123 (x1, x2, x3); }

  public Vector3D (Vector3D v)
  { x = new double [3]; setVector3D (v); }

  // Methods

  public void setX123 (double x1, double x2, double x3)
  { x [0] = x1;  x [1] = x2;    x [2] = x3;     }
```

```
public void setVector3D (Vector3D v)
{ for (int i=0; i<3; i++) x[i] = v.x[i]; }

public double getX (int i)
{ if (i<0 || i>2) return 0.0; else return x[i]; }

public void setX (int i, double d)
{ if (i<0 || i>2) return; else x[i] = d; }

public String toString ()
{
  return "Vector3D "+MyLib.display (x[0],3)+", "
      +MyLib.display (x[1],3)+", "+MyLib.display (x[2],3)+";";
}

public double abs ()                        // length
{
  return Math.sqrt (x[0]*x[0]+x[1]*x[1]+x[2]*x[2]);
}

public static double abs (Vector3D v)  // length
{
  return Math.sqrt (v.x[0]*v.x[0]+v.x[1]*v.x[1]+v.x[2]*v.x[2]);
}

public Vector3D toPolar ()               // to Polar Coordinates
/*-----------------------------------------------
  Vektor3D ---> Polarkoordinaten (in Vector3D res)
  -----------------------------------------------*/
{
  double r = abs ();
  double phi, theta;
  double go = 4.0*
           (Math.abs (x[0]) + Math.abs (x[1]) + Math.abs (x[2]));
  if (Math.abs (x [0]) <= Aconst.EpsilonDouble*go)
  {
    if (Math.abs (x[1]) <= Aconst.EpsilonDouble*go)
        phi = 0.0;
    elsephi = Aconst.HalfPi;
    if (x[1] < -Aconst.EpsilonDouble*go) phi = -phi;
  }
  else phi = Math.atan2 (x [1], x [0]);
  if (phi < 0.0) phi += Aconst.TwoPi;
  if (r <= Aconst.EpsilonDouble*go) theta = Aconst.HalfPi;
  else theta =  Math.acos (x [2] / r);
  Vector3D res = new Vector3D (r, phi, theta);
    // res is Polar Vector!!
  return res;
}

public Vector3D toLongLat ()  // to Longitude/Latitude
/*-----------------------------------------------
  Vektor3D ---> R, Long., Lat. (in Vector3D h)
  -----------------------------------------------*/
{
  Vector3D h = new Vector3D (toPolar ());
  h.x[2] = Aconst.HalfPi - h.x[2];
  return h;
}
```

```
public Vector3D toRaDec ()    // to Right Ascension/Declination
/*-------------------------------------------
  Vektor3D ---> R, RA, Dec (in Vector3D h)
  -------------------------------------------*/
{
  Vector3D h =  new Vector3D (toLongLat ());
  h.x[1] /= 15.0;
  return h;
}

public Vector3D toAzEl ()          // to Azimuth/Elevation
/*-------------------------------------------
  Vektor3D ---> R, Az, El. (in Vector3D h)
  -------------------------------------------*/
{
  Vector3D h = new Vector3D (toPolar ());
  h.x[2] = Aconst.HalfPi - h.x[2];
  h.x[1] = Aconst.Pi - h.x[1];
  if (h.x[1] < 0.0) h.x[1] += Aconst.TwoPi;
  return h;
}

public Vector3D fromLongLat (double r, double lon, double lat)
// from Longitude/Latitude
/*-------------------------------------------
  Vector3D <--- R, Long., Lat.
  -------------------------------------------*/
{
  lat = Aconst.HalfPi - lat;
  fromPolar (r, lon, lat);
  return this;
}

public Vector3D fromRaDec (double r, double ra, double dec)
// from Right Ascension/Declination
/*-------------------------------------------
  Vector3D <--- R, RA, Dec
  -------------------------------------------*/
{
  dec = Aconst.HalfPi - dec;
  ra *= 15.0;
  fromPolar (r, ra, dec);
  return this;
}

public Vector3D fromAzEl (double r, double az, double el)
// from Azimuth/Elevation
/*-------------------------------------------
  Vector3D <--- R, Az, El.
  -------------------------------------------*/
{
  el = Aconst.HalfPi - el;
  az = Aconst.Pi - az;
  if (az < 0.0) az += Aconst.TwoPi;
  fromPolar (r, az, el);
  return this;
}
```

```
public Vector3D toVector ()
{
  Vector3D res = new Vector3D ();
  res.toVector (x[0], x[1], x[2]);
  return res;
}

public void fromPolar (double r, double phi, double theta)
{
  toVector (r, phi, theta);
}

public void toVector (double r, double phi, double theta)
  // polar coordinates to Vector
/*-----------------------------------------
  Polarkoordinaten ---> Vektor
  -----------------------------------------*/
{
  double d = r * Math.sin (theta);
  x[0] = d * Math.cos (phi);
  x[1] = d * Math.sin (phi);
  x[2] = r * Math.cos (theta);
  clean ();
}

public void clean ()      // set near to zero values to zero
//------------------
{
  double go = abs ()*Aconst.EpsilonDouble*4.0;
  for (int i=0; i<3; i++) if (Math.abs (x[i]) < go) x[i] = 0.0;
}

// Vektoroperationen

public Vector3D unit ()
//---------------------
{
  Vector3D temp = new Vector3D (this);
  temp.over (abs ());
  return temp;
}

public void setUnit ()
//---------------------
{
  over (abs ());
}

public static Vector3D plus   (Vector3D a, Vector3D b)
{
   Vector3D help = new Vector3D ();
   for (int i=0; i<3; i++) help.x [i] = a.x [i] + b.x [i];
   return help;
}

public void plus (Vector3D a)
{
   for (int i=0; i<3; i++) x [i] += a.x [i];
}
```

```
public static Vector3D minus  (Vector3D a, Vector3D b)
{
   Vector3D help = new Vector3D ();
   for (int i=0; i<3; i++) help.x [i] = a.x [i] - b.x [i];
   return help;
}

public void minus (Vector3D a)
{
   for (int i=0; i<3; i++) x [i] -= a.x [i];
}

public static Vector3D cross (Vector3D a, Vector3D b)
{
   Vector3D help = new Vector3D ();
   help.x [0] =  a.x [1] * b.x [2] - a.x [2] * b.x [1];
   help.x [1] =  a.x [2] * b.x [0] - a.x [0] * b.x [2];
   help.x [2] =  a.x [0] * b.x [1] - a.x [1] * b.x [0];
   return help;
}

public static double dot (Vector3D a, Vector3D b)
{
   double sum = 0.0;
   for (int i=0; i<3; i++) sum += a.x [i] * b.x [i];
   return sum;
}

public double times (Vector3D b) { return dot (this, b); }

public void times (double f)     // Vector multiplied by scalar
{
   for (int i=0; i<3; i++) x [i] *= f;
}

public void over (double f)      // Vector divided by scalar
{
   for (int i=0; i<3; i++) x [i] /= f;
}

public static Vector3D times (double f, Vector3D v)
{
  Vector3D h = new Vector3D (v);
  h.times (f);
  return h;
}

public void aberration (Vector3D v) // linear approximation
{
  Vector3D r0 = new Vector3D (this);
  Vector3D v0 = new Vector3D (v);
  double r_abs = abs ();
  double v_abs = v.abs ();
  if (v_abs < 1e-6) return;
  r0.over (r_abs);
  v0.over (v_abs);
  Vector3D del = new Vector3D (v0);
  Vector3D h = new Vector3D (r0);
  h.times (dot (r0, v0));
  del.minus (h);
```

```
    del.times (v_abs/Aconst.SpeedOfLight);
    r0.minus (del);
    r0.setUnit ();
    r0.times (r_abs);
    setVector3D (r0);
  }
}
```

Matrix

Passend zu den dreidimensionalen Vektoren werden 3 mal 3 dimensionale Matrizen eingeführt, die im wesentlichen die räumliche Drehung von Vektoren im Raum abbilden sollen. Dabei werden nicht wirklich die Vektoren gedreht, sondern die Koordinatensysteme, in denen die Vektorkomponenten quantitativ beschrieben sind.

$$\boldsymbol{M} = \begin{pmatrix} m_{11} & m_{12} & m_{13} \\ m_{21} & m_{22} & m_{23} \\ m_{31} & m_{32} & m_{33} \end{pmatrix}$$

```
/**
 *Matrix3D.java
 *
 * 3 Vector3Ds representing a 3x3 matrix
 *
 * @see  Vector3D
 *
 * @author Dieter Egger
 * @version  1.3.1    2001/01/10
 *
 */

package mie;

public class Matrix3D extends Object  // Object 3,3-Matrix
{
  public Vector3D v[];                     // 3 rowVector3D's

  // Constructors

  public Matrix3D () { init (); }

  public Matrix3D (Vector3D v1, Vector3D v2, Vector3D v3)
  { init (); setRowVectors (v1, v2, v3); }

  public Matrix3D (Matrix3D m) { init (); setMatrix3D (m); }

  void init ()
  {
    v = new Vector3D [3];
    for (int i=0; i<3; i++) v[i] = new Vector3D ();
  }

  // Methods

  public void setRowVectors
              (Vector3D v1, Vector3D v2, Vector3D v3)
```

```
{
  v[0].setVector3D (v1);
  v[1].setVector3D (v2);
  v[2].setVector3D (v3);
}

public void setMatrix3D (Matrix3D m)
{ for (int i=0; i<3; i++) v[i].setVector3D (m.v[i]); }

public void setUnit ()
{
  v[0].setX123 (1.0, 0.0, 0.0);
  v[1].setX123 (0.0, 1.0, 0.0);
  v[2].setX123 (0.0, 0.0, 1.0);
}

public static Matrix3D rotationMatrix (int axis, double angle)
/*-------------------------------------------
  Calculation of Rotation Matrix R_axis (angle)
  axis == 1 : x-axis
  axis == 2 : y-axis
  axis == 3 : z-axis
  angle given in radians
  -------------------------------------------*/
{
  double c, s;
  Matrix3D M = new Matrix3D ();

  M.setUnit ();
  c = Math.cos (angle); s = Math.sin (angle);
  switch (axis)
  {
  case 1:
      M.v[1].setX123 (0.0,  c,  s);
      M.v[2].setX123 (0.0, -s,  c);
      break;
  case 2:
      M.v[0].setX123 ( c, 0.0, -s);
      M.v[2].setX123 ( s, 0.0,  c);
      break;
  case 3:
      M.v[0].setX123 ( c,  s, 0.0);
      M.v[1].setX123 (-s,  c, 0.0);
      break;
  default: break;
  }
  return (M);
}


public double det ()                    // Determinant of Matrix
//------------------
{
   double h;
   h =  v [0].x[0] * v [1].x[1] * v [2].x[2]
    + v [0].x[1] * v [1].x[2] * v [2].x[0]
    + v [0].x[2] * v [1].x[0] * v [2].x[1]
    - v [2].x[0] * v [1].x[1] * v [0].x[2]
    - v [2].x[1] * v [1].x[2] * v [0].x[0]
    - v [2].x[2] * v [1].x[0] * v [0].x[1];
```

```
   return h;
}

public void clean ()      // set near to zero values to zero
//------------------
{
  double go = det ()*Aconst.EpsilonDouble*4.0;
  for (int j=0; j<3; j++)
  for (int i=0; i<3; i++)
    if (Math.abs (v[j].x[i]) < go) v[j].x[i] = 0.0;
}

// Vector operations

public static Matrix3D plus   (Matrix3D a, Matrix3D b)
{
   Matrix3D help = new Matrix3D ();
   for (int i=0; i<3; i++)
      help.v [i].setVector3D (Vector3D.plus (a.v [i], b.v [i]));
   return help;
}

public void plus (Matrix3D a)
{
   for (int i=0; i<3; i++)
      v [i].setVector3D (Vector3D.plus (this.v[i], a.v [i]));
}

public static Matrix3D minus (Matrix3D a, Matrix3D b)
{
   Matrix3D help = new Matrix3D ();
   for (int i=0; i<3; i++)
      help.v [i].setVector3D (Vector3D.minus (a.v [i], b.v [i]));
   return help;
}

public void minus (Matrix3D a)
{
   for (int i=0; i<3; i++)
      v [i].setVector3D (Vector3D.minus (this.v[i], a.v [i]));
}

public Vector3D getColumnVector (int i)
{
  if (i>=0 && i<3)
    return new Vector3D (v[0].x[i], v[1].x[i], v[2].x[i]);
  else return new Vector3D ();
}

public Vector3D getRowVector (int i)
{
  if (i>=0 && i<3) return v[i];
  else return new Vector3D ();
}

public void setColumnVector (int i, Vector3D cv)
{
  if (i>=0 && i<3)
  {
    for (int j=0; j<3; j++)   v[j].x[i] = cv.x[j];
```

```
    }
  }

  public void setRowVector (int i, Vector3D rv)
  {
    if (i>=0 && i<3) v[i].setVector3D (rv);
  }

  public double getElement (int i, int j)
  {
    if (i>=0 && i<3) return v[i].getX (j); else return 0.0;
  }

  public void setElement (int i, int j, double w)
  {
    if (i>=0 && i<3) v[i].setX (j, w);
  }

  public static Matrix3D times (Matrix3D a, Matrix3D b)
  {
     Matrix3D h = new Matrix3D ();

     for (int i=0; i<3; i++)
      for (int j=0; j<3; j++)
      h.v[i].x[j] =
           Vector3D.dot (a.v [i],  b.getColumnVector (j));
     return h;
  }

  public static Vector3D times (Matrix3D a, Vector3D b)
  {
     Vector3D h = new Vector3D ();

     for (int i=0; i<3; i++) h.x[i] = Vector3D.dot (a.v [i], b);
     return h;
  }

  public Vector3D times (Vector3D b) { return times (this, b); }

  public void times (double f)  // Matrix multiplied by scalar
  {
     for (int i=0; i<3; i++) v[i].times (f);
  }

  public void over (double f)              // Matrix divided by scalar
  {
     for (int i=0; i<3; i++) v[i].over (f);
  }

  public Matrix3D transposed ()       // Transpose of Matrix
  {
     Matrix3D h = new Matrix3D ();

     for (int i=0; i<3; i++)
        h.v[i].setVector3D (getColumnVector (i));
     return h;
  }

  public String toString ()
  {
```

```
    return   v[0].toString()+"\n"+
         v[1].toString()+"\n"+
         v[2].toString();
  }
}
```

Rotation

Umrechnung der Vektorkomponenten $\boldsymbol{x}$ von einem Referenzsystem in Vektorkomponenten $\boldsymbol{x'}$ eines dazu verdrehten Referenzsystems. Eine sogenannte *Rotationsmatrix* $\boldsymbol{R}$ vermittelt den Übergang. Sie ist eine Orthonormalmatrix, die im Sinne der Matrix-Vektor-Multiplikation (Zeile mal Spalte, alles im 3-dimensionalen Raum) den Ausgangsvektor umrechnet:

$$\boldsymbol{x'} = \boldsymbol{R}_i(\alpha)\boldsymbol{x}, \quad i \in [1,2,3]$$

Drehung um die Achse $\boldsymbol{e}_i$ um den Winkel α gegen den Uhrzeigersinn (mathematisch positiv). Der Positionsvektor $\boldsymbol{x}$' zeigt zwar nach wie vor zum selben Ort, hat aber im verdrehten Koordinatensystem $\boldsymbol{e}_i$' anders lautende Komponenten als im System $\boldsymbol{e}_i$. Die beiden Vektoren sind qualitativ gleich, unterscheiden sich aber in ihren Komponenten.

```
/**
 *Rotation.java
 *
 * abstract class for rotation of a Vector3D
 *
 * @author Dieter Egger
 * @version  1.3.1    2001/01/10
 *
 */

package mie;

public abstract class Rotation extends Object
{
  Matrix3D tM;

  public Rotation () { tM = new Matrix3D (); }

  public void setRotationMatrix (Matrix3D tm)
  { tM.setMatrix3D (tm); }

  public Vector3D forward (Vector3D b)
  { return Matrix3D.times (tM, b); }
  public Vector3D back    (Vector3D b)
  { return Matrix3D.times (tM.transposed (), b); }
}
```

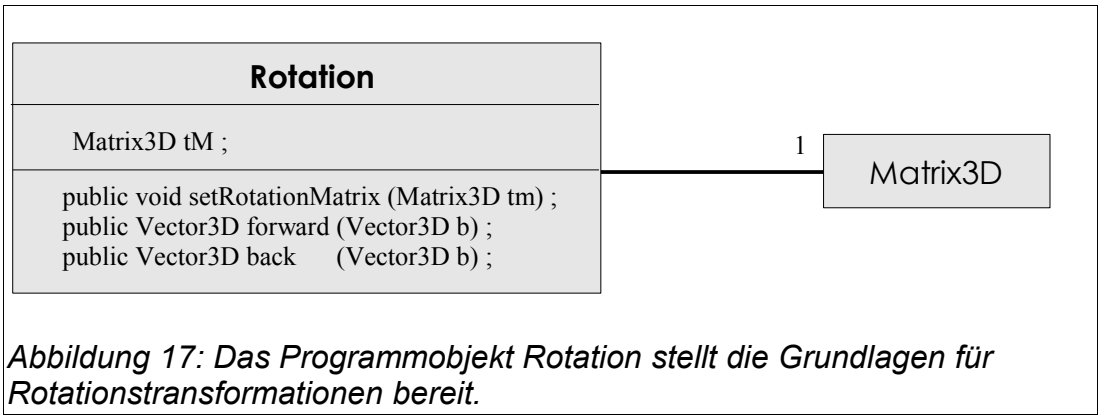

Abbildung 17: Das Programmobjekt Rotation stellt die Grundlagen für Rotationstransformationen bereit.

Translation

Vektorielle Verschiebung des Koordinatenursprungs bei gleichbleibender Orientierung der Koordinatenachsen (*Parallelverschiebung*).

$$x' = x - v$$

Sie wird durch den *Translationsvektor* v beschrieben, der vom „alten“ zum „neuen“ Koordinatenursprung zeigt. x' und x zeigen zwar zum selben Ort, aber von verschiedenen Ursprüngen aus. Die beiden Vektoren sind somit auch qualitativ verschieden.

```
/**
 *Translation.java
 *
 * abstract class for translation of position and velocity
 *
 * @author Dieter Egger
 * @version  1.3.1    2001/01/10
 */

package mie;
public abstract class Translation extends Object
{
  Vector3D tR, tV;

  public Translation ()
  { tR = new Vector3D (); tV = new Vector3D (); }

  public void setTranslationVector (Vector3D dR, Vector3D dV)
  { tR.setVector3D (dR); tV.setVector3D (dV);}
  public Vector3D forward (Vector3D b)
  { return Vector3D.minus (b, tR); }
  public Vector3D back    (Vector3D b)
  { return Vector3D.plus  (tR, b); }
  public Vector3D forwardV (Vector3D b)
  { return Vector3D.minus (b, tV); }
  public Vector3D backV    (Vector3D b)
  { return Vector3D.plus  (tV, b); }
}
```

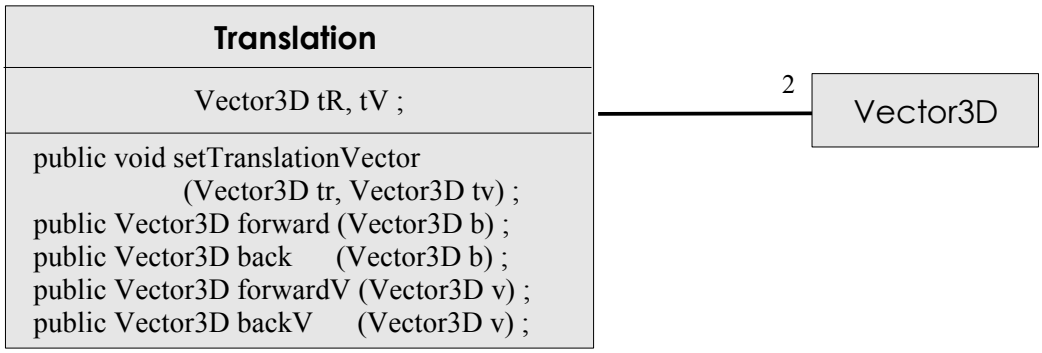

Abbildung 18: Translationen verlangen eine Unterscheidung, ob Orte oder ob Geschwindigkeiten transformiert werden sollen. Aus diesem Grund enhält das Objekt Translation zwei weitere Methoden.

Rotationstransformationen

Eine Reihe von Übergängen zwischen den oben erwähnten Referenzsystemen werden durch Rotationen beschrieben:

Übergang	Zusammenhang
Ekliptiksystem --> Äquatorsystem	Ekliptikschiefe
Mittl. Äqu.s.(Epoche J2000) --> mittl. raumfestes Äqu.s.(aktuell)	Präzession
Mittleres raumfestes Äqu.s. --> wahres raumfestes Äqu.s.	Nutation
Mittleres raumfestes Äqu.s. --> erdfestes Äquatorsystem	Mittlere Sternzeit
Wahres raumfestes Äqu.s. --> erdfestes Äquatorsystem	Wahre Sternzeit
Momentanes erdfestes --> vereinbartes erdfestes Äqu.s.	x, y Pol
Äquatorsystem --> Horizontsystem	λ, φ Beobachter

Tabelle 5: Zusammenhänge zwischen Bezugssystemen, die mit Rotationen ineinander überführt werden können

Die erreichbare Genauigkeit hängt von der Genauigkeit der Parameter ab, die die Zusammenhänge zwischen den verschiedenen Systemen beschreiben. Sowohl ihre Modellierung als auch ihre präzise Messung sind dabei wichtig.

Beispiel:

```
/**
 *Ecliptic2Equator.java
 *
 * define the object Ecliptic2Equator which actually
 * performs the transformation from the ecliptical system
 * to the equatorial system at the actual epoch (JulianTime)
 *
 * @see Rotation
 * @see Ecliptic
 * @see Matrix3D
 * @see JulianTime
 *
```

```
 * @author Dieter Egger
 * @version  1.3.1    2001/01/10
 */

import mie.*;

public class Ecliptic2Equator extends Rotation
{
  Ecliptic ecl;

  public Ecliptic2Equator (JulianTime t)
  {
    ecl = new Ecliptic (t);
    setRotationMatrix
        (Matrix3D.rotationMatrix (1, -ecl.getEps ()));
  }
}
```

Translationstransformationen

Einige Übergänge zwischen den oben erwähnten Referenzsystemen erfordern Translationen:

Übergang	Zusammenhang
Baryzentrisches --> geozentrisches System	Baryzentrum--> Erde
Heliozentrisches --> geozentrisches System	Sonne --> Erde
Geozentrisches --> topozentrisches System	Erde --> Beobachter

Tabelle 6: Zusammenhänge zwischen Bezugssystemen, die mit Translationen ineinander überführt werden können

Die erreichbare Genauigkeit hängt von der genauen Bestimmung der Position der Erde im Sonnensystem und von der genauen Bestimmung der Position des Beobachters auf der Erdoberfläche ab (relativ zum Massenmittelpunkt der Erde).

Beispiel:

```
/**
 *Geo2Topocenter.java
 *
 * define the object Geo2Topocenter which actually
 * performs the transformation from the earth-centered system
 * to the observer-centered system
 *
 * @see Translation
 * @see Observer
 * @see Vector3D
 *
 * @author Dieter Egger
 * @version  1.3.1    2001/01/10
 *
 */

import mie.*;

public class Geo2Topocenter extends Translation
{
```

```
  public Geo2Topocenter (Observer obs)
  {
    setTranslationVector (obs.getLocation (),
    Vector3D.cross( new Vector3D
      (0,0,Aconst.TwoPi/Aconst.SiderealDay), obs.getLocation()) );
  }
}
```

Gängige Transformationen

Die gegenseitige Lagerung gebräuchlicher Referenzsysteme wurde so gewählt, dass keine Spiegelungen anfallen (siehe Tabelle 7). Bis auf das Horizontsystem des Beobachters scheint dies durchaus gerechtfertigt. Steht der Astronom im Vordergrund, so wird er die x-Achse nach Süden orientieren, die y-Achse nach Westen und die z-Achse zum Zenit, wodurch sich ein Linkssystem ergibt. Gleichermassen wird der Geodät die x-Achse nach Norden, die y-Achse nach Osten und die z-Achse zum Zenit zeigen lassen, was wiederum ein Linkssystem ergibt. Letztlich sind für beide aber nicht die Vektorkomponenten von Bedeutung, sondern zwei Winkel, die als Azimut und Elevation bezeichnet werden. Hier unterscheiden sich Astronom und Geodät aber immer noch. Zwar ist die Zählrichtung bei beiden im Uhrzeigersinn (von oben bzw. vom Zenit, also der Spitze der z-Achse aus betrachtet), jedoch beginnt der Astronom im Süden und der Geodät im Norden zu zählen. In dieser Arbeit und somit auch in der Astro-Toolbox, wird das Nordazimut verwendet, also die Sichtweise des Geodäten bevorzugt, allerdings nicht die zugehörige Vektordarstellung gewählt. Um das Linkssystem zu vermeiden, wird die x-Achse nach Süden, die y-Achse nach Osten und die z-Achse zum Zenit ausgerichtet.

Wenn Geschwindigkeiten im Spiel sind, werden sie separat behandelt. Zwar ändert sich nichts bei einer Rotation, die einfach mit der gleichen Drehmatrix durchgeführt werden kann, aber sehr wohl ändert sich die Vorgehensweise bei Translationen, da nun nicht mehr der Orts-Translationsvektor anzuwenden ist, sondern der Geschwindigkeits-Translationsvektor.

Günstigerweise führt man wegen der Zeitabhängigkeit sowohl der Orte und der Geschwindigkeiten, als auch der zugrunde liegenden Referenzsysteme, neue Objekte ein, die beispielsweise für einen Satelliten jeweils eindeutig festlegen, in welchem Bezugssystem einer bestimmten Epoche (der Referenzepoche), Position und Geschwindigkeit zu einer bestimmten Zeit (der Orbit- oder Bahnepoche) gelten. Die beiden Zeiten müssen nicht notwendigerweise identisch sein.

RefTimeOrbitVector

RefTimeOrbitVector ist das zentrale Objekt für alle anfallenden Transformationen. Selbst wenn bei einer Transformation nicht das vollständige Objekt vorliegt, so wird es doch intern dazu ergänzt. Wegen der Vererbungshierarchie bereitet es keine Schwierigkeiten, jeweils den gleichen Objekttyp weiterzugeben, der als Eingabe vorgelegen hatte. Wie aus Abbildung 19 hervorgeht, entspricht es vier Objekten gleichzeitig. Wenn man *Java.Object* hinzunimmt (von dem *Vector3D* abgeleitet ist), zeigt es sich sogar in 5 Gesichtern. Es ist ausgesprochen polymorph.

Der OrbitVector ist bereits im Abschnitt Orbit abgedruckt (siehe Seite 46).

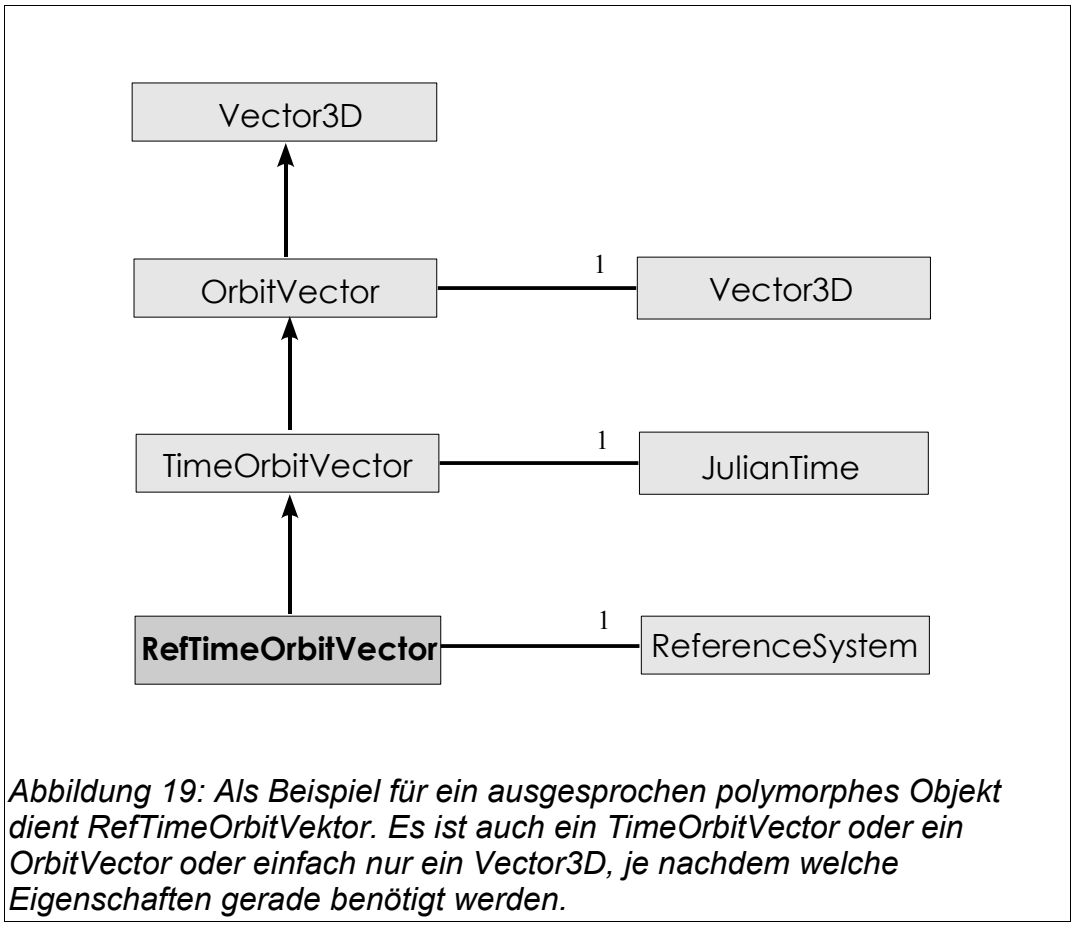

Abbildung 19: Als Beispiel für ein ausgesprochen polymorphes Objekt dient RefTimeOrbitVektor. Es ist auch ein TimeOrbitVector oder ein OrbitVector oder einfach nur ein Vector3D, je nachdem welche Eigenschaften gerade benötigt werden.

Ergänzt man ihn um eine Zeitangabe, die festlegt, wann ein Himmelsobjekt diesen Ort und diese Geschwindigkeit einnimmt, so erhält man einen TimeOrbitVector. Wenn man nun auch noch sagt, in welchem Bezugssystem die Werte gelten, ist man bei einer vollständigen Beschreibung angelangt. Das entsprechende Objekt ist ein RefTimeOrbitVector.

```
/**
 *RefTimeOrbitVector.java
 *
 * TimeOrbitVector in a given reference system
 *
 * @see  TimeOrbitVector
 *
```

```
 * @author Dieter Egger
 * @version  1.3.1    2001/01/10
 *
 */

package mie;

public class RefTimeOrbitVector extends TimeOrbitVector
{
  protected ReferenceSystem refSys;

  public RefTimeOrbitVector ()
  {
    refSys = new ReferenceSystem ();
    refSys.setJulianTime (JulianTime.actualNoon ());
  }

  public RefTimeOrbitVector (TimeOrbitVector v)
  {
    super (v);
    refSys = new ReferenceSystem ();
    refSys.setJulianTime (JulianTime.actualNoon ());
  }

  public RefTimeOrbitVector
    (ReferenceSystem rf, TimeOrbitVector v)
  {
    super (v);
    refSys = new ReferenceSystem (rf);
  }

  public RefTimeOrbitVector (RefTimeOrbitVector old)
  {
    refSys = new ReferenceSystem ();
    setRefTimeOrbitVector (old);
  }

  public void setRefTimeOrbitVector (RefTimeOrbitVector ro)
  {
    setTimeOrbitVector (ro);
    refSys.setReferenceSystem (ro.getRefSys ());
  }

  public ReferenceSystem getRefSys () { return refSys; }

  public void setRefSys (ReferenceSystem rs)
  { refSys.setReferenceSystem (rs); }

  public void setIdRS (String rs) { refSys.setIdRS (rs); }
  public void setJulianTimeRS (JulianTime t)
  { refSys.setJulianTime (t); }

  public void setReferenceId (String rs) { refSys.setIdRS (rs); }
  public void setReferenceEpoch (JulianTime t)
  { refSys.setJulianTime (t); }

  public String getReferenceId ()
  { return refSys.getShortName(); }
  public JulianTime getReferenceEpoch () { return refSys; }
```

```
  public String toString ()
  {
    return "RefTimeOrbitVector\n "+refSys.toString ()
               +"\n "+super.toString();
  }
}
```

Der Vollständigkeit halber sei auch das Objekt TimeOrbitVector in seiner Quellenform vorgestellt:

```
/**
 *TimeOrbitVector.java
 *
 * OrbitVector at given time
 *
 * @see  OrbitVector
 *
 * @author Dieter Egger
 * @version  1.3.1    2001/01/10
 *
 */

package mie;

public class TimeOrbitVector extends OrbitVector
{
  protected JulianTime epoch;

  // Constructors

  public TimeOrbitVector ()
  {
    epoch = new JulianTime (JulianTime.actualNoon ());
  }

  public TimeOrbitVector (OrbitVector o)
  {
    super (o);
    epoch = new JulianTime (JulianTime.actualNoon ());
  }

  public TimeOrbitVector (JulianTime t, OrbitVector o)
  {
    super (o);
    epoch = new JulianTime (t);
  }

  public TimeOrbitVector (TimeOrbitVector B)
  {
    super (B.getPos (), B.getVel ());
    epoch = new JulianTime (B.getEpoch ());
  }

  public void setTimeOrbitVector (TimeOrbitVector B)
  {
    setOrbitVector (B);
    setEpoch (B.getEpoch ());
  }

  // Access Data
```

```
  public void setEpoch (JulianTime ep)
  {
    epoch.setJulianTime (ep);
  }

  public JulianTime getEpoch () { return epoch; }

  public String toString ()
  {
    return "TimeOrbitVector\n Epoch "+epoch.toString ()
               +"\n "+super.toString();
  }
}
```

Doch nun zum Überblick über die gängigsten Transformationen:

Transformation	Übergangsobjekt	Formel
BE ⇒ GE	Erde, baryzentrisch	$\boldsymbol{x}_{GE} = \boldsymbol{x}_{BE} - Erde_{BE}$
HE ⇒ GE	Erde, heliozentrisch	$\boldsymbol{x}_{GE} = \boldsymbol{x}_{HE} - Erde_{HE}$
GE ⇒ SF	Ekliptik	$\boldsymbol{x}_{SF} = \boldsymbol{R}_1(-\epsilon_0)\,\boldsymbol{x}_{GE}$
SF_J2000 ⇒ mSF	Präzession	$\boldsymbol{x}_{mSF} = \boldsymbol{R}_3(-z_A)\,\boldsymbol{R}_2(\theta_A)\,\boldsymbol{R}_3(-\zeta_A)\,\boldsymbol{x}_{J2000}$
mSF ⇒ tSF	Nutation	$\boldsymbol{x}_{tSF} = \boldsymbol{R}_1(-\epsilon)\,\boldsymbol{R}_3(-\Delta\psi)\,\boldsymbol{R}_1(\epsilon_0)\,\boldsymbol{x}_{mSF}$
tSF ⇒ EF	wahre Sternzeit θ_{true}	$\boldsymbol{x}_{EF} = \boldsymbol{R}_3(\theta_{true})\,\boldsymbol{x}_{tSF}$
mSF ⇒ EF	mittlere Sternzeit θ_{mean}	$\boldsymbol{x}_{EF} = \boldsymbol{R}_3(\theta_{mean})\,\boldsymbol{x}_{mSF}$ (ohne Nutation!)
EF ⇒ cEF	Polbewegung	$\boldsymbol{x}_{cEF} = \boldsymbol{R}_2(-x_P)\,\boldsymbol{R}_1(-y_P)\,\boldsymbol{x}_{EF}$
cEF ⇒ TE	Beobachter	$\boldsymbol{x}_{TE} = \boldsymbol{x}_{cEF} - Beobachter_{cEF}$
TE ⇒ TH	Beobachter	$\boldsymbol{x}_{TH} = \boldsymbol{R}_2(\frac{\pi}{2} - \phi)\,\boldsymbol{R}_3(\lambda)\,\boldsymbol{x}_{TE}$

Tabelle 7: Transformationsvorschriften für die Übergange zwischen verschiedenen Bezugssystemen

die Abkürzungen bedeuten:

BE	baryzentrisches Ekliptiksystem
HE	heliozentrisches Ekliptiksystem
GE	geozentrisches Ekliptiksystem
SF	Space-Fixed = raumfestes, geozentrisches Äquatorsystem
SF_J2000	mittleres Äquatorsystem zur Epoche J2000 (z.B. ICRS)
mSF	mean space-fixed = mittleres Äquatorsystem zur aktuellen Epoche
tSF	true space-fixed = wahres Äquatorsystem zur aktuellen Epoche
EF	earth-fixed = momentanes, mit der Erde rotierendes Äquatorsystem (auch als erdfest bezeichnet)
cEF	conventional earth-fixed = vereinbartes, mit der Erde rotierendes Äquatorsystem (z.B. ITRS)
TE	topocentric equator = Äquatorsystem des Beobachters
TH	topocentric horizon = Horizontsystem des Beobachters

Tabelle 8: Bedeutung der Abkürzungen für die verschiedenen Bezugssysteme

Anmerkungen:

Die mittlere Sternzeit θ_{mean} wird auch als *GMST* (Greenwich Mean Sidereal Time), die wahre Sternzeit θ_{true} als *GAST* (Greenwich Apparent Sidereal Time) bezeichnet.

Momentanes und vereinbartes Äquatorsystem sind *erdfeste Systeme*, die sich durch die Lage des Pols unterscheiden. Das erste beruht auf dem aktuellen, also tatsächlichen Pol, während das zweite auf einem *vereinbarten* Pol aufbaut. Im vereinbarten System bleiben die Koordinaten von Beobachtungsstationen bis auf die Kontinentaldrift erhalten.

Mit den eben beschriebenen Transformationen sind die Übergänge noch nicht vollständig erfasst. Sie beruhen nämlich auf der Annahme, dass die Referenzsysteme bewegungs- und gravitationsfrei sind. Dass dem nicht so ist, dürfte offensichtlich genug sein:

die Sonne ist sehr massereich,

die Erde bewegt sich um das Baryzentrum,

die Erde rotiert,

um die wichtigsten Eigenschaften zu nennen, die obige Voraussetzung nicht erfüllen.

Hätte man die Transformationen von vornherein relativistisch abgefasst, wäre man aus dem Schneider. Allerdings zu einem recht hohen Preis, denn die komplizierten Formeln erschweren das Verständnis.

Somit müssen die kleinen Abweichungen in Form von sogenannten *relativistischen* Korrekturen angebracht werden [Seidelmann, Taff]:

Beim Übergang vom Baryzentrum (oder Heliozentrum) zum Geozentrum:

1) Gravitationslichtablenkung (Masse der Sonne)
2) jährliche Aberration (Geschwindigkeit der Erde)

Beim Übergang vom Geozentrum zum Topozentrum:

3) tägliche Aberration (Geschwindigkeit des Beobachters)

Bei bewegten Objekten zusätzlich

4) Lichtzeitkorrektur (z.B. Geschwindigkeit eines Planeten)

2) bis 4) waren schon vor EINSTEIN bekannt.

Lichtablenkung wegen Gravitation der Sonne

Sehr weit entfernte *Sterne* erscheinen nahezu in derselben Richtung, egal, ob sie von der Sonne oder von der Erde aus betrachtet werden. In diesem Fall vereinfacht sich die Berechnung der gravitativen Lichtablenkung zu [Seidelmann]

$$\boldsymbol{p}_1 = \boldsymbol{p} + g\frac{\boldsymbol{e} - (\boldsymbol{p}\cdot\boldsymbol{e})\,\boldsymbol{p}}{1 + \boldsymbol{p}\cdot\boldsymbol{e}} \quad , \qquad g = \frac{2GM}{c^2}\cdot\frac{1}{E}$$

mit $\boldsymbol{p}$ Richtungsvektor zum Stern, $\boldsymbol{e}$ Richtungsvektor von der Sonne zur Erde, *GM* Gravitationskonstante mal Masse der *Sonne*, *c* Lichtgeschwindigkeit, *E* Entfernung der Erde von der Sonne und schliesslich $\boldsymbol{p}_1$ neuer Richtungsvektor zum Stern.

```
/**
 *LightDeflection.java
 *
 * define the object LightDeflection which corrects
 * for the gravitational influence of the Sun
 * valid for the ecliptical system J2000
 *
 * @see EarthBary
 * @see Vector3D
 *
 * @author Dieter Egger
 * @version  1.3.1    2001/01/10
 *
 */

import mie.*;

public class LightDeflection extends Object
{
  EarthBary b;
  Vector3D eh, eh1;

  public LightDeflection (JulianTime t)
  {
    b = new EarthBary (t);
    eh = new Vector3D (b.pos_earth_helio ());
    eh1 = new Vector3D (eh.unit ());
  }
```

```
  public Vector3D gravOfSun (Vector3D v)
  {
    Vector3D v1 = new Vector3D (v.unit ());
    double g = 2.0*Aconst.GM_Sun
               /MyLib.sqr (Aconst.SpeedOfLight)/eh.abs();
    Vector3D h = new Vector3D (eh1);
    h.plus (Vector3D.times (-Vector3D.dot (v1, eh1), v1));
    h.times (g/(1.0+Vector3D.dot (v1, eh1)));
    h.plus (v1);
    h.times (v.abs ());
    return h;
  }
}
```

Lichtablenkung wegen bewegtem Beobachter

Wenn wir nur geringe Geschwindigkeiten $\boldsymbol{v}$ eines Beobachters ($v << c$) zulassen, können wir die resultierende *Lichtablenkung* linear annähern. Die *Aberration*, wie dieser Effekt allgemein bezeichnet wird, führt dann zu einem veränderten Richtungsvektor $\boldsymbol{p}_1$, der stets in Bewegungsrichtung verschoben erscheint [Seidelmann]

$$\boldsymbol{p}_1 = \frac{\boldsymbol{p} + \boldsymbol{v}/c}{|\boldsymbol{p} + \boldsymbol{v}/c|}$$

$\boldsymbol{p}$ entspricht dem Richtungsvektor, den ein ruhender Beobachter wahrnehmen würde.

```
/**
 *TLinearAberration.java   ---Excerpt---
 *
 * tool which corrects for the moving object/observer
 * linear approximation valid for velocity << speed of light
 * valid for any system
 *
 * @see Tool
 * @see RefTimeOrbitVector
 * @see Vector3D
 *
 * @author Dieter Egger
 * @version  1.3.1    2001/01/10
 *
 */

import java.applet.*;
import java.awt.*;
import java.awt.event.*;

import mie.*;

public class TLinearAberration extends Tool implements ActionListener
{
  RefTimeOrbitVector actualRefTOV;
  String transferType = "RefTimeOrbitVector";

  ........
  void update ()
```

```
  {
    Vector3D h = new Vector3D (actualRefTOV.getPos ());
    h.aberration (actualRefTOV.getVel ());
    actualRefTOV.setPos (h);
    ta.setText (actualRefTOV.getRefSys ().toString ());
    ta.append ("\n");
    ta.append (actualRefTOV.toString ());
    processActionEvent
       (new ActionEvent
           (this, ActionEvent.ACTION_PERFORMED, transferType));
    footer.setText (transferType+" ready");
  }
  ........
}
```

Beobachter

Um die Position eines Beobachters auf der Erdoberfläche festzulegen, bedarf es der genauen Kenntnis der *Erdform*, die zwar grob einem (Rotations-)Ellipsoid entspricht, bei näherem Hinschauen aber doch davon abweicht. Zu verschiedenen Zeiten wurden verschiedene Ellipsoide als geometrische Näherungsformen verwendet [Sigl].

Zur Festlegung des geozentrischen Ortsvektors muss die Lage des *Geozentrums*[1] bestimmbar sein. Ebenfalls bekannt sein muss die Lagerung des Koordinatensystems. Wegen der *Polschwankungen* hat man sich auf einen mittleren Pol geeinigt, den sogenannten *vereinbarten Pol*, damit die Koordinaten des Beobachters bis auf plattentektonische Bewegungen erhalten bleiben. In diese Richtung zeigt nun die z-Achse. Die x-Achse zeigt zum Schnittpunkt „Äquator – Meridan von Greenwich“.

Der Beobachter wird schliesslich durch (ellipsoidische) Länge, Breite und Höhe oder den entsprechenden geozentrischen Ortsvektor fixiert. Bei bekanntem *Ellipsoid* (definiert durch den *Äquatorradius* der Erde und die *Abplattung*) können die Angaben ineinander umgerechnet werden.

Ausgehend vom *Äquatorradius* a und der *Abplattung* $f=(a-b)/a$, mit b *Polradius* können über

$$e^2 = \frac{a^2 - b^2}{a^2}$$

und

$$N = \frac{a}{\sqrt{1 - e^2 \sin^2 \phi}}$$

[1]Das Massenzentrum der Erde kann beispielsweise durch die Laserentfernungsmessung zu Satelliten bestimmt werden.

leicht die *geodätischen* Koordinaten *Länge* λ, *Breite* φ und *Höhe* h in *geozentrische Koordinaten* überführt werden:

$$\begin{pmatrix} x \\ y \\ z \end{pmatrix} = \begin{pmatrix} (N+h)\cos\phi\cos\lambda \\ (N+h)\cos\phi\sin\lambda \\ (N(1-e^2)+h)\sin\phi \end{pmatrix}$$

Etwas komplizierter gestaltet sich die Umkehrung, die vorzugsweise iterativ vollzogen wird [Seeber]:

$$\lambda = \arctan\frac{y}{x}$$

$$\phi = \arctan\frac{z}{\sqrt{x^2+y^2}}\left(1-e^2\frac{N}{N+h}\right)^{-1}$$

$$h = \frac{\sqrt{x^2+y^2}}{\cos\phi} - N$$

Wie in [Seidelmann] und [Hekimoglu] ausgeführt, kann die womöglich entstehende Nullstellenproblematik (in Polnähe) beherrscht werden.

```
/**
 *Observer.java
 *
 * Vector3D plus ellipsoidal coordinates
 * representing an observer's position,
 * and the name of the observer's place
 *
 * @see  Vector3D
 *
 * @author Dieter Egger
 * @version  1.3.1    2001/01/10
 *
 */

package mie;

public class Observer extends Vector3D
{
  private String name;
  private double longitude;
  private double latitude;
  private double height;
  private double radius = Aconst.EarthEquatorialRadius;
  private double oblateness = Aconst.EarthOblateness;

  public Observer ()
  {
    name = new String ("North Pole");
    setLonLatH ( 0.0, 0.5*Aconst.Pi, 0.0);
  }
```

```
public Observer (String who, double l, double b, double h)
{
  name = new String (who);
  setLonLatH (l, b, h);
}

public Observer (String who, Vector3D b)
{
  name = new String (who);
  setLocation (b);
}

public Observer (Observer obs)
{
  setObserver (obs);
}

public void setObserver (Observer obs)
{
  setVector3D (obs);
  name = new String (obs.getName());
  longitude = obs.getLongitude();
  latitude  = obs.getLatitude();
  height    = obs.getHeight();
  radius = obs.getRadius();
  oblateness = obs.getOblateness();
}

public void resetEllipsoid ()
{
  radius = Aconst.EarthEquatorialRadius;
  oblateness = Aconst.EarthOblateness;
}

public void setLonLatH (double l, double b, double h)
{
  longitude = l;
  latitude = b;
  height  = h;
  lonLatH2Location ();
}

public String getName          () { return name; }
public double getLongitude     () { return longitude; }
public double getLatitude      () { return latitude; }
public double getHeight        () { return height; }
public Vector3D getLocation    () { return this; }
public double getLocation (int i) { return this.getX (i); }
public double getRadius        () { return radius; }
public double getOblateness    () { return oblateness; }

public String toString ()
{
  return "Observer "+name+"\t Long "
       +MyLib.display(longitude*Aconst.RadianToDegree,4)
       +"\t Lat "
       +MyLib.display(latitude*Aconst.RadianToDegree,4)
       +"\t Height "
       +MyLib.display(height,3);
}
```

```
public void setName (String s) { name = new String (s); }
public void setLongitude (double lon)
    { longitude = lon; lonLatH2Location (); }
public void setLatitude  (double lat)
    { latitude = lat; lonLatH2Location (); }
public void setHeight    (double h)
    { height = h; lonLatH2Location (); }
public void setLocation  (Vector3D loc)
    { setVector3D (loc); location2LonLatH (); }
public void setRadius    (double r)      { radius = r; }
public void setOblateness(double o)      { oblateness = o; }

private void location2LonLatH ()
/*-------------------------------------------
  geocentric location vector --->
  ell. Longitude, Latitude, Height
  -------------------------------------------*/
{
  double nq, hc, hh, eps = 10.0*Aconst.EpsilonDouble;

  double a = radius;
  double xq = x [0];
  double yq = x [1];
  double zq = x [2];

  double sh = Math.sqrt (MyLib.sqr (xq) + MyLib.sqr (yq));
  double b = a * (1.0 - oblateness);
  double e2 = (a*a - b*b)/(a*a);
  height = hc = 0.0;

  if (Math.abs (xq) < eps)
  {
    if (Math.abs (yq) < eps) longitude = 0.0;
    else if (yq < 0.0) longitude = -Aconst.HalfPi;
    else longitude = Aconst.HalfPi;
  }
  else longitude = Math.atan2 (yq, xq);

  if (sh < eps)
  {
    if (Math.abs (zq) < eps) latitude = 0.0;
    else if (zq < eps) latitude = -Aconst.HalfPi;
    else latitude = Aconst.HalfPi;
  }
  else latitude = Math.atan2 (zq, sh);

  do
  {
      nq = a/Math.sqrt(1.0 - e2*MyLib.sqr (Math.sin (latitude)));
      if (Math.abs (latitude)
          < Aconst.HalfPi-1.0/3600.0*Aconst.DegreeToRadian)
        height = sh/Math.cos (latitude) - nq;
      else height = Math.abs (zq) - b;

      if (Math.abs (height - hc) < 0.0001) break;

      if (sh > eps)
      {
        hh = zq/sh * 1.0/(1.0 - e2*(nq/(nq+height)));
```

```
          latitude = Math.atan (hh);
        }
        hc = height;
    }
    while (true);
  }

  private void lonLatH2Location ()
  /*-------------------------------------------
    ell. Longitude, Latitude, Height --->
    geocentric location vector
    -------------------------------------------*/
  {
    double a, b, c, s, y;

    a = radius;
    b = a*(1.0-oblateness);
    a *= a;
    b *= b;
    c = Math.cos (latitude);
    s = Math.sin (latitude);
    y = a/Math.sqrt (a*c*c+b*s*s);
    x [0] = (y+height)*c*Math.cos (longitude);
    x [1] = (y+height)*c*Math.sin (longitude);
    x [2] = (y*b/a + height)*s;
  }
}
```

Wetter

Luftdruck, Temperatur und Luftfeuchtigkeit der *Erdatmosphäre* wirken sich auf alle Beobachtungen aus, die von der Erde aus vorgenommen werden[2]. Kennt man die aktuellen Werte nicht, nimmt man zweckmässigerweise eine *Normatmosphäre* von 15 Grad Celsius und 1013.25 Hektopascal an. Die Luftfeuchte wird in der Normatmosphäre allerdings nicht erfasst. Ein Wert um 30% mag angebracht erscheinen, wenn man trockenes Wetter in Betracht zieht.

Die *Atmosphäre* wirkt sowohl refraktiv, als auch dispersiv und extinktiv. Ihr jeweiliger, momentaner Aufbau ist meist ungenügend bekannt. Zum einen liegt es an turbulenten Luftströmungen und zum anderen daran, dass nicht jederzeit an allen Punkten der durchquerten Atmosphäre aktuelle Messdaten von Temperatur T, Luftdruck P und Partialdruck des Wasserdampfes verfügbar sind.

Man greift deshalb auf theoretische Modelle zurück, die bei bekannten atmosphärischen Daten auf der Beobachtungsstation den kompletten Aufbau der restlichen Atmosphäre ableiten (vgl. [Egger]).

Für *Elevationen* e grösser 15, bzw. $z=90-e$ kleiner 75 Grad, genügt die einfache Formel [Mueller]

[2] in 45 Grad Elevation verursacht die Refraktion eine vertikale Richtungsabweichung von etwa einer Bogenminute.

$$\Delta z = \frac{P}{T}\left(16.293 \tan z - 0.0187 \tan^3 z\right)$$

zur Beschreibung der *Refraktion*, mit z *Zenitdistanz*, P Luftdruck in Hektopascal und T Temperatur in Kelvin. Um Δz Bogensekunden liegt beispielsweise ein Stern tiefer als beobachtet.

Für geringere Elevationen e (< 15 Grad) ist der Zusammenhang, wie bei [Seidelmann] angegeben, vorzuziehen:

$$\Delta e = \frac{P}{T}\left(\frac{0.1594 + 0.0196\,e + 0.00002\,e^2}{1.0 + 0.505\,e + 0.0845\,e^2}\right)$$

e und Δe verstehen sich hier im Gradmass.

```
/**
 *Weather.java
 *
 * keep track of meteorological data and influences
 *
 * @author Dieter Egger
 * @version  1.3.1    2001/01/10
 *
 */

package mie;

public class Weather extends Object
{
  private double temperature;   // Kelvin, only for internal use
  private double pAir;          // HectoPascal
  private double pWatervapor;   // HectoPascal

  public Weather () { setTPe (15.0, 1013.25, 13.0); }

  public Weather (double tC, double pA, double pe)
      { setTPe (tC, pA, pe); }

  public void setTPe (double tC, double pA, double pe)
      // Degree Celsius
  { pAir = pA; pWatervapor = pe; temperature = tC + 273.15; }

  public void setWeather (Weather w)
  {
    temperature = w.temperature;
    pAir = w.pAir;
    pWatervapor = w.pWatervapor;
  }

  /**
   * compute astronomical refraction
   *
   * @param z is the zenith angle in radian
   * @return influence of atmospheric refraction in radian
   *
   * @author Dieter Egger
```

```
   * @version  1.3.1  2001/01/10
   *
   */
  public double refraction (double z)
  {
    double d, f, g;
    double hpt = pAir/temperature;

    g = 90.0 - z*Aconst.RadianToDegree;
    if (g>15.0)
    {
      d = Math.tan (z);
      f = hpt *
          (16.293*d - 0.0187*d*d*d)/3600.0*Aconst.DegreeToRadian;
    }
    else
    if (g>=-0.6)
    {
      f = hpt * (0.1594 + 0.0196*g + 0.00002*g*g)/
        (1.0    + 0.505 *g + 0.0845 *g*g) *Aconst.DegreeToRadian;
    }
    else f = 0.0;
    return (f);
  }

  public double getTemperature ()
      { return temperature - 273.15; } // Degree Celsius
  public double getPAir () { return pAir; }
  public double getPWatervapor () { return pWatervapor; }

  public void setTemperature (double tC) // Degree Celsius
  { if (tC > -273.15) temperature = tC + 273.15; }
  public void setPAir         (double pa) { pAir = pa; }
  public void setPWatervapor (double pw) { pWatervapor = pw; }

  public String toString ()
  { return "Weather T "+MyLib.display(temperature-273.15,3)
             +"C\t p="+MyLib.display(pAir,3)
             +"HP\t e="+MyLib.display(pWatervapor,3)+"HP"; }
}
```

Literatur

Balzert H.: *Lehrbuch der Software-Technik – Software-Entwicklung*, Spektrum Akademischer Verlag, Heidelberg, 1996 (mit CD-ROM)

Bastos A.: *Celestial Objects and Satellite Astronomy*. ESRO SP-89, October 1974

Brumberg V.A.: *Essential Relativistic Celestial Mechanics*, Adam Hilger, Bristol, 1991

Doberenz W.: *JAVA*, Carl Hanser, München, 1996 (mit CD-ROM)

Egger D.: *Systemanalyse der Laserentfernungsmessung.* DGK, Reihe C, Nr. 311, München, 1985

Egger D.: *Neuronales Netz prädiziert Erdrotationsparameter.* Allgemeine Vermessungsnachrichten (AVN) 11/12, 517-524, 1992

Egger D.: *Objektorientierte Modellierung eines Teilbereichs der Astronomie und Himmelsmechanik mit Implementierung in Java.* Shaker Verlag, Aachen, 2000

Epstein L.C.: *Relativitätstheorie – anschaulich dargestellt*, Birkhäuser, Basel, 1985

Fliessbach T.: *Allgemeine Relativitätstheorie*, BI-Wiss.-Verl., Mannheim, 1990

Fowler M., Scott K.: *UML konzentriert*, Addison Wesley, Bonn, 1998

Friedel D.H.jun., Potts A.: *JAVA-Programmierhandbuch*, tewi Verlag, München, 1996

Grossman N.: *The sheer Joy of Celestial Mechanics*, Birkhäuser, Boston, 1996

Hekimoglu S.: *Generalized iterative solution for geodetic coordinates from cartesian coordinates.* In: Bolletino di Geodesia e Scienze affini, No. 2, 1995

Herrmann J.: *dtv-Atlas zur Astronomie*. Deutscher Taschenbuch Verlag, München, 1990

Holzner S.: *Java 1.1 - Das Trainingsbuch*, Sybex, Düsseldorf, 1997

Kaler J.B.: *Astronomy!*, Harper Collins, New York, 1994

Kaufmann W.J. III, Comins N.F.: *Discovering the Universe*, 4th Ed., Freeman, New York, 1996 (with CD-ROM)

Lemay L., Perkins Ch.L.: *teach yourself JAVA in 21 days*, Sams.net, Indianapolis, 1996

Meeus J.: *Astronomical Formulae for Calculators*, 3rd Ed., Willmann-Bell, Richmond, 1985

Meeus J.: *Astronomische Algorithmen*, Barth, Leipzig, 1992

Middendorf S., Singer R., Strobel S.: *JAVA Programmierhandbuch und Referenz*, dpunkt, Heidelberg, 1996

Moore P., Zimmer H.: *Guinness Buch der Sterne*. Ullstein Verlag, Frankfurt/ Main, 1985

Mueller I.I.: *Introduction to Satellite Geodesy.* Frederick Ungar Publishing Co., New York, 1964

North G.: *Astronomy Explained*, Springer, London, 1997

Oestereich B.: *Objektorientierte Softwareentwicklung,* Oldenbourg Verlag, München Wien, 1998

Petrahn G.: *Grundlagen der Vermessungstechnik*, Cornelsen, Berlin, 1996

Schmutzer E.: *Relativitätstheorie aktuell*, 5. Aufl., Teubner, Stuttgart, 1996

Schneider M.: *Himmelsmechanik*, mehrere Bände, 3. völlig neu bearb. und erw. Auflage, BI-Wiss.-Verl., Mannheim: Band 1, 1992 und Band 2, 1993, Spektrum Akademischer Verlag, Heidelberg: Band 3, 1996 und Band 4, 1999

Schneider M.: *Satellitengeodäsie*, BI-Wiss.-Verl., Mannheim, 1988

Schröder U.E.: *Spezielle Relativitätstheorie*, 3. Aufl., Harri Deutsch, Thun, 1994

Seeber G.: *Satellite Geodesy*, de Gruyter, Berlin, 1993

Seidelmann K. (Ed.): *Explanatory Supplement to the Astronomical Almanac*, University Science books, Mill Valley, CA, 1992

Sigl R.: *Geodätische Astronomie*, 3. Auflage, Herbert Wichmann Verlag, Karlsruhe, 1983

Stardivision: *StarOffice Benutzerhandbuch*, Stardivision GmbH, Hamburg, 1998, jetzt bei Fa. Sun, Kalifornien, www.sun.com

Sun: www.sun.com, Java Development Kit + Documentation, Versionen bis 1.3.1 wurden eingesetzt.

Taff L.G.: *Computational Spherical Astronomy*, Wiley, New York, 1981

The Astronomical Almanac for the Year 1984, Her Majesty's Stationery Office, London, 1983

Vanicek P., Krakiwsky E.: *Geodesy – The Concepts*, 2nd Ed., North-Holland, Amsterdam, 1986

Vogel H.: *Gerthsen Physik*, 18. Aufl., Springer, Berlin, 1995

Wilhelms G., Kopp M.: *Java professionell*, MIPT-Verlag, Bonn, 1999

Zimmermann H., Weigert A.: *ABC-Lexikon Astronomie*, 8. Aufl., Spektrum Akademischer Verlag, Heidelberg, 1995

Index