Kapitel 10: Klassen in Java

Software Entwicklung 1

Annette Bieniusa, Mathias Weber, Peter Zeller

Wir haben im vorherigen Kapitel bereits erste Erfahrung mit der objekt-orientierten Programmierung in Java gemacht. Dabei stand die Verwendung von Objekten im Mittelpunkt, insbesondere Konstruktor- und Methodenaufrufe. Obgleich für Java vielfältige Bibliotheken existieren, sind Programmierer dennoch oft in der Situation eigene Klassen definieren zu müssen.
In diesem Kapitel werden wir sehen, wie man in Java eigene Klassentypen implementieren kann. Dabei stehen die Definition von Attributen und Methoden im Vordergrund. Um Klassen zu modellieren verwenden wir UML-Klassendiagramme und leiten von diesen Implementierungen ab. Dabei erläutern wir die Komposition von Klassen und Objektgeflechte. Wir vertiefen den Umgang mit Klassen und Objekten im Rahmen einer Fallstudie, einem Code-Tagebuch.

Lernziele dieses Kapitels:

- Klassen in Java (Attribute, Konstruktoren, Methoden) zu deklarieren.
- Einfache Arten von UML-Klassendiagramme zu lesen und zu erstellen.
- Delegation als Programmiermuster anzuwenden.
- Aus einer Problembeschreibung ein objektorientiertes Modell und Implementierung ableiten.

1 Klassendeklarationen in Java

Eine einfache ***Klassendeklaration*** in Java hat folgende Bestandteile:

```
class Person {                          Klassenname
  String name;                          Attribut
  Person(String n) {                    Konstruktor
    this.name = n;
  }
  String getName() {                    Methode
    return this.name;
  }
}
```

Ein Java-Objekt kann genau auf die Nachrichten reagieren, für die Methoden in seiner Klasse deklariert sind oder für die es Methoden geerbt hat (siehe Kapitel “Vererbung”).

- In der Regel speichern wir den Code für die Deklaration einer Klasse in einer gleichnamigen `.java` - Datei ab.
  Im Beispiel: `Person.java`
- Klassennamen beginnen in Java nach Konvention mit einem Großbuchstaben.

Eine Klasse kann durch ein ***Klassendiagramm*** spezifiziert werden. Klassendiagramme dienen hauptsächlich der Datenmodellierung. Sie sind im UML (Unified Modeling Language) Standard definiert. Ein Klassendiagramm enthält den *Name* der Klasse, die *Attribute* mit dem jeweiligen Typ und *Methoden* mit Signaturen.

Beispiel Ein UML-Klassendiagramm für die `Person`-Klasse sieht so aus:

Person
name : String
getName() : String

Die Objekte einer Klasse K nennt man auch ***Instanzen*** oder *Ausprägungen* von K. Sie werden durch Objektdiagramme modelliert.

: Person
name = “Karl”

1.1 Klassenname

Direkt hinter dem Schlüsselwort `class` wird der Name der Klasse angegeben.
Der Klassenname wird gleichzeitig als *Typname* für die Objekte dieser Klasse verwendet (*Klassentyp*). Er kann im Programm dann wie elementare Typen (`int`, `double`, usw.) für die Deklaration von lokalen Variablen, Parametern und Rückgabewerten verwendet werden.

1.2 Attribute

Innerhalb einer Klasse K können Attribute deklariert werden. Für jedes in K deklarierte Attribut vom Typ T besitzen die Objekte der Klasse K eine ***objektlokale*** Variable vom Typ T. Diese objektlokalen Variablen nennt man häufig auch ***Instanzvariablen***. Die Werte dieser Attribute entsprechen dem aktuellen Zustand einer Instanz dieser Klasse.
Die Lebensdauer der Instanzvariablen entspricht der Lebensdauer des Objekts.

Syntax in Java:

```
AttributDeklaration →
      Typ ≪ Bezeichner ≫;
    | Typ ≪ Bezeichner ≫ = Ausdruck;
```

Beispiel Die Klasse `Mensch`, die einen Menschen (vereinfacht) modelliert, umfasst die folgenden Attribute:

```
class Mensch {
  String name;
  Mensch vater;
  Mensch mutter;
  boolean lebt = true;
}
```

1.3 Konstruktoren und Initialisierung

Konstruktoren erzeugen und initialisieren Objekte. Sie haben den gleichen Namen wie die Klasse, in der sie deklariert sind. Beim Start der Ausführung eines Konstruktors ist das zugehörige Objekt bereits erzeugt, seine Attribute jedoch nur mit Standardwerten initialisiert. Im Konstruktor werden die Objektattribute geeignet initialisiert, basierend auf den aktuellen Parametern beim Konstruktoraufruf.
Konstruktoren liefern als Ergebnis das neu erzeugte Objekt zurück, genauer, eine Referenz auf dieses Objekt.

Syntax in Java:

```
KonstruktorDeklaration →
    ≪ Bezeichner ≫ (FormaleParameter) Anweisungsblock
```

Beispiele: In Anlehnung an die `Color`-Klasse aus Kapitel 09 deklarieren wir eine `Farbe` -Klasse. Diese enthält Attribute für die RGB-Werte, beim Konstruktoraufruf mit den entsprechenden Parameterwerten initialisiert werden.

```
class Farbe {
  // Attribute:
  int rot;
  int gruen;
  int blau;
  // Konstruktor:
  Farbe(int rot, int gruen, int blau) {
    this.rot = rot;
    this.gruen = gruen;
    this.blau = blau;
  }
}
```

Zur Erinnerung: Konstruktoren werden über einen Ausdruck der Form `new` ≪ *Bezeichner* ≫ (AktuelleParameterListe) aufgerufen, wie wir im vorherigen Kapitel bereits gesehen haben. Zum Beispiel: `new Farbe(255,200,100)`. Die aktuelle Parameterliste muss dabei zu den formalen Parametern in der Konstruktor-Deklaration passen, das heißt Anzahl und Typen müssen übereinstimmen.

Attribute (wie auch lokale Variablen) können alternativ direkt an ihrer Deklarationsstelle initialisiert werden. Die folgenden drei Definition der Klasse `C` sind daher äquivalent:

Option 1	Option 2	Option 3

`class C {`
`  int a;`
`  C() {`
`  a = 78;`
`  }`
`}`
	`class C {`
	`  int a = 78;`
	`  C() {}`
	`}`
		`class C {`
		`  int a = 78;`
		`}`

Falls kein Konstruktor deklariert ist, erhält die Klasse einen leeren Standard-Konstruktor. Wie in Option 3 kann daher der leere Konstruktor weggelassen werden.
Falls (mind.) ein Konstruktor in der Klasse definiert ist, gibt es keinen Standard-Konstruktor. Zum Beispiel kann die Klasse `Farbe` oben nur mit Angabe der drei Farbwerte initialisiert werden. Der Aufruf `new Farbe()` ohne Parameter ist **nicht** gültig.
Wie oben erwähnt, erfolgt die Initialisierung von Attributen *vor* dem Eintritt in den Konstruktorrumpf.
In Java können Attribute und Variablen durch das Schlüsselwort `final` als ***unveränderlich*** deklariert werden. In diesem Fall *muss* die Initialisierung an der Deklarationsstelle erfolgen oder in geeigneter Weise im Konstruktor.

```
class Mathe {
   ...
   final float PI = 3.141; // Konstante
   ...
}
```

1.4 Methoden

Innerhalb der Klassendeklaration können beliebig viele *Methoden* deklariert werden. Methodendeklarationen bestehen aus einer Signatur und einem Methodenrumpf. Syntaktisch sind sie wie Prozedurdeklarationen aufgebaut.
Außer den deklarierten Parametern besitzt jede Methode `m` einen weiteren, sogenannten ***impliziten Parameter*** vom Typ der Klasse, in der `m` deklariert wurde. Dieser Parameter wird im Methodenrumpf mit `this` bezeichnet.

Syntax in Java:

MethodenDeklaration →
  TypOderVoid ≪ *Bezeichner* ≫ (FormaleParameter) Anweisungsblock

```
// Zugriff auf impliziten Parameter in Methoden:
Ausdruck → this
```

Beispiel Wir ergänzen die Klasse `Mensch` um eine Methode, die den Namen liefert:

```
class Mensch {
  // ...
  String getName() {
    return this.name;
  }
}
```

1.5 Attributzugriff

Syntax in Java:
Auf Instanzvariablen von Objekten kann mit Ausdrücken folgender Form zugegriffen werden:

Ausdruck → Ausdruck . ≪ *Bezeichner* ≫

Zuweisung → Ausdruck . ≪ *Bezeichner* ≫ = Ausdruck;

Semantik:
Werte den Ausdruck aus; dieser muss eine Referenz liefern.
Liefert dieser `null`, löse eine `NullPointerException` aus.
Andernfalls liefert er die Referenz auf ein Objekt X; in dem Fall liefert der gesamte Ausdruck die Instanzvariable von X zum angegebenen Attribut (L-Wert) oder deren Wert (R-Wert).

Beispiel: Kaskadierender Attributzugriffe Man kann den Zugriff auf Attribute direkt hintereinander ausführen. Im folgenden Beispiel wird in Zeile 12 auf das Attribut `name` des Objekts zugriffen, dessen Referenz als Attribut `autor` in der aktuellen Instanz verwendet wird.

```
1  class Autor {
2    String name;
3    int    geburtsjahr;
4  }
5
6  class Buch {
7    Autor  autor;
8    String titel;
9
10    void printInfo() {
11       StdOut.println("Titel:" + this.titel);
12       StdOut.println("Autor:" + this.autor.name);
```

```
13      }
14  }
```

Abkürzende Notation Der implizite Methodenparameter `this` kann beim Zugriff auf ein Attribut `a` weggelassen werden, d.h. `a` ist gleichbedeutend mit `this.a` innerhalb von Klassen, in denen `a` deklariert ist (und in denen es keine andere lokale Variable mit gleichem Namen gibt).
Wie beim Attributzugriff kann auch beim Methodenaufruf *innerhalb der Klassendeklaration* der implizite Methodenparameter `this` weggelassen werden, also `m(...)` statt `this.m(...)`.

1.6 Testen von Klassen

Wir könne das JUnit-Framework auch zum Testen von Klassendeklarationen verwenden. Dazu müssen für den Test Instanzen der zu testenden Klasse abgeleitet werden und geeignet initialisiert werden. Im folgenden Beispiel wird eine Familienhierarchie mit der `Mensch`-Klasse nachgebildet und damit die Methode `getOpa` getestet.

```
1   import org.junit.Test;
2   import static org.junit.Assert.*;
3
4   class Mensch {
5     Mensch vater, mutter;
6     String name;
7     boolean lebt;
8
9     Mensch(String name, Mensch vater, Mensch mutter) {
10      this.name = name;
11      this.vater = vater;
12      this.mutter = mutter;
13    }
14
15    Mensch getOpa (boolean mutterseits) {
16      if (mutterseits) {
17        return mutter.vater;
18      } else {
19        return vater.vater;
20      }
21    }
22  }
23
24  // Beispiel zur Verwendung:
25  public class MenschTest {
26    @Test
27    public void ermittleOpa() {
28      Mensch gundula = new Mensch("Gundula", null, null);
29      Mensch fritz = new Mensch("Fritz", null, null);
30      Mensch erika = new Mensch("Erika", null, null);
31      Mensch detlef = new Mensch("Detlef", null, null);
32      Mensch christine = new Mensch("Christine", fritz, gundula);
33      Mensch bob = new Mensch("Bob", detlef, erika);
34      Mensch alice = new Mensch("Alice", bob, christine);
35
36      Mensch opaV = alice.getOpa(false);
37      assertEquals(detlef, opaV);
38
39      String opaMName = alice.getOpa(true).name;
40      assertEquals("Fritz", opaMName);
41    }
```

```
42 }
```

2 Fallstudie: Code-Tagebuch

In diesem Abschnitt zeigen wir das Erstellen von Klassen in einem größeren Kontext. Hier ist die Problembeschreibung:

> Entwickle ein Programm, mit dem man ein persönliches Code-Tagebuch führen kann.
>
> - Es soll möglich sein, Einträge im Code-Tagebuch hinzuzufügen und zu entfernen, sich alle Einträge anzeigen zu lassen, die Gesamtzahl der geschriebenen Codezeilen zu ermitteln und den Tag mit der höchsten Geschwindigkeit bei der Codegenerierung zu ermitteln.
> - Ein Eintrag besteht aus dem Datum, der Anzahl an geschriebenen Zeilen Code, der Dauer und einem Kommentar. Für einen Eintrag soll man außerdem die durchschnittliche Anzahl an Zeilen Code pro Session ermitteln.
> - Ein Datum besteht aus Tag, Monat und Jahr.

Um ein Programm für diese Aufgabenstellung zu implementieren, gehen wir wie folgt vor:

1. Studiere die Problembeschreibung. Identifiziere die darin beschriebenen Objekte und ihre Attribute und Methoden.
2. Erstelle entsprechende Klassendiagramme. (*Entwurf*)
3. Übersetze die Klassendiagramm in eine Klassendefinition. Füge einen Kommentar hinzu, der den Zweck der Klasse erklärt. (*Implementierung*)
4. Repräsentiere einige Beispiele durch Objekte. Stelle fest, ob sie den Anforderungen entsprechen. (*Test*)

2.1 Analyse der Problembeschreibung

Um zu ermitteln, wie die Klassen (und damit Objekte) aussehen, die in dem Programm verwendet werden sollen, betrachten wir die Problembeschreibung. Dabei sollten zuerst die Substantive und Verben in der Beschreibung analysiert werden.

- Substantive in der Beschreibung liefern Hinweise auf Klassen oder Attribute.
- Verben liefern Hinweise für Methoden.

Für die benötigten Daten müssen außerdem Datentypen entsprechend ihrer Verwendung gewählt werden.

2.2 Klassendiagramm

Aus der Problembeschreibung können wir folgende Klassendiagramme ableiten:

Entry
day : Date loc : int duration : int comment : String
double calculateSpeed()

Date
day : int month : int year : int

CoderDiary
name : String liste : List≤Entry≥
void add(Entry e) void remove(int i) int totalLoc() Date dayWithMaxSpeed()

Um die `Entry`-Objekte eines Code-Tagebuchs zu verwalten, wollen wir in diesem Beispiel – als Alternative zum Arry – eine Liste verwenden (Details dazu im nächsten Abschnitt).

2.3 Implementierung

2.3.1 Die Klassen Entry und Date

Aus dem Klassendiagramm lässt sich nun direkt ein Gerüst für die Implementierung für `Entry` und `Date` ableiten.

```
1  // Repraesentiert einen Eintrag im Lauftagebuch
2  class Entry{
3      Date day;
4      int loc;         // lines of code
5      int duration;    // in min
6      String comment;
7
8
9      // requires duration != 0
10     Entry(Date day, int loc, int duration, String comment) {
11         this.day = day;
12         this.loc = loc;
13         this.duration = duration;
14         this.comment = comment;
15     }
16
17     // Durchschnittsgeschwindigkeit LOC/h bei der Codeerzeugung
18     double calculateSpeed() {
19         double h = duration / 60.0;
20         return loc / h;
21     }
22
23     // String-Repräsentierung
24     public String toString() {
25         return this.day.toString() + " "
26             + this.loc + " LOC in "
27             + this.duration  + " min; "
28             + this.comment;
29     }
30
31 }
```

```
1  // Repräsentiert einen Datumswert bestehend aus Tag, Jahr und Monat
2  class Date {
3    int day;
4    int month;
5    int year;
6
7    Date(int day, int month, int year) {
8      this.day = day;
9      this.month = month;
10     this.year = year;
11   }
12
13   // Alternativer Konstruktor aus String (dd.mm.yyyy)
14   Date(String s){
15     this.day   = Integer.valueOf(s.substring(0,2));
16     this.month = Integer.valueOf(s.substring(3,5));
17     this.year  = Integer.valueOf(s.substring(6,10));
18   }
19
20   // String-Repraesentation; Beispiel: "01.12.2016"
21   public String toString() {
22     return this.day + "." + this.month + "." + this.year;
23   }
24 }
```

Da wir später Datumswerte als Benutzereingabe in Form eines Strings erlauben wollen, haben wir in der `Date`-Klasse zwei Konstruktoren definiert.

- Der erste Konstruktor `Date(int day, int month, int year)` erwartet drei `int`-Parameter für Tag, Monat und Jahr.
- Der zweite Konstruktor `Date(String s)` erwartet eine String-Repräsentierung des Datums im Format `dd.mm.yyyy`, wie z.B. “30.12.2019”. Dieser Konstruktor extrahiert aus dem String `s` die Werte für Tag, Monat und Jahr.

Wir wollen die folgenden Einträge im Lauftagebuch eintragen:

- am 2. November 2017, 756 LOC in 210 Minuten, Initiales Design
- am 8. November 2017, 140 LOC in 20 Minuten, Tests
- am 10. November 2017, 8 LOC in 75 Minuten, Debugging

Dazu erzeugen wir die folgenden Objekte:

```
diary.add(new Entry(new Date(2,11,2017),765,210,"Initiales Design"));
diary.add(new Entry(new Date(8,11,2017),140,20,"Tests"));
diary.add(new Entry(new Date(10,11,2017),8,75,"Debugging"));
```

Diese kompakte Konstuktorverschachtelung kann auch in zwei Schritten mit Hilfsdefinitionen erfolgen:

```
Date d1 = new Date(2,11,2017);
Entry e1 = new Entry(d1,765,210,"Initiales Design"));
```

Alternativ können wir `Date`-Objekte mit dem anderen Konstruktor aus einem String erstellen, z.B.:

```
Date d2 = new Date("02.11.2017");
```

Wir haben beiden Klassen jeweils eine `toString`-Methode hinzugefügt. Alle Referenztypen in Java haben eine Standardmethode `toString()`. In den meisten Fällen ist es aber sinnvoll eine eigene String-Repräsentationen zu definieren. Die `toString()` Methode **muss** als `public` deklariert werden (Näheres dazu im Kapitel "Vererbung"). Die String-Repräsentation enthält üblicherweise Informationen zu den Attributwerten. `toString()` wird (automatisch) aufgerufen, wenn das Objekt in einem Kontext verwendet wird, das einen String erwartet.

```
Date d = new Date (5,6,2015);
StdOut.println(d.toString());
StdOut.println(d); //alternativ
```

Für die Implementierung von `toString()` ***delegieren*** wir die String-Repräsentierung des Datumsattributs an die `Date`-Klasse.

```
class Entry {
  Date d;
  int loc;
  int duration;     // in min
  String comment;
  ...
  public String toString() {
        return d.toString() + ": " + this.loc + " LOC in "
            + this.duration + " min; " + this.comment;
  }
}
```

Dieses Muster ist typisch für Klassen, deren Objekte (u.a.) aus Objekten anderer Klassen zusammengesetzt sind (sogenannte ***Aggregate*** oder ***Kompositionen***). Eine Menge von Objekten, die sich gegenseitig referenzieren, nennen wir ein ***Objektgeflecht***. Objektgeflechte werden zur Laufzeit aufgebaut und verändert, sind also dynamische Entitäten. Klassendiagramme kann man als vereinfachte statische Approximationen von Objektgeflechten verstehen.

2.3.2 Die Klasse CoderDiary

Unsere Implementierung umfasst eine Klasse `Entry` für einzelne Einträge im Tagebuch. Noch offen ist die Problemstellung ein Tagebuch mit **beliebiger Anzahl** von `Entry`-Objekten zu modellieren und implementieren.
Ein Möglichkeit ist die Einträge des Tagebuchs in einem Array mit `Entry`-Objekten zu verwalten. Wir zeigen hier eine alternative Variante, nämlich die Einträge in einer ***Liste*** zu verwalten. Die hier verwendete `ArrayList`-Klasse aus der `java.util.ArrayList` -Bibliothek basiert auf einem Array zur Verwaltung der Einträge, vereinfacht die Verwendung aber durch abstraktere Methoden. [1] Wie beim Array geben wir dabei an, von welchem Typ die Elemente in der Liste sind. Hier ein Auszug aus der API der Klasse `ArrayList`:

[1] Wir diskutieren die Implementierung von `ArrayList` und anderen Listen im Detail in Kapitel 12.

	API von `java.util.ArrayList`	(Auszug)
	`ArrayList<E>()`	Konstruktor für Liste mit Elementen vom Typ `E`
`int`	`size()`	Anzahl der Elemente
`E`	`get(int i)`	Liefert das Element an Position `i`
`E`	`remove(int i)`	Entfernt das Element an Position `i` und gibt dieses zurück

```
1  // Code-Tagebuch
2  import java.util.ArrayList;
3
4  class CoderDiary{
5      String name;
6      ArrayList<Entry> liste;
7
8      CoderDiary(String name) {
9          this.name = name;
10         this.liste = new ArrayList<Entry>();
11     }
12
13     // Fuegt einen Eintrag am Ende der Liste hinzu
14     void add(Entry e) {
15       this.liste.add(e);
16     }
17
18     // Entfernt den Eintrag an Position i
19     void remove(int i) {
20       this.liste.remove(i);
21     }
22
23     // Ermittelt die Gesamtzahl der Codezeilen
24     int totalLoc() {
25         int sum = 0;
26         for(Entry e : this.liste) {
27           sum += e.loc;
28         }
29         return sum;
30     }
31
32     // Ermittelt den Tag des Eintrags mit der höchten
33     // Codeerzeugungsrate
34     Date dayWithMaxSpeed() {
35         double maxSpeed = 0.0;
36         Date   maxSpeedDay = null;
37         for(Entry e : this.liste) {
38             double speed = e.calculateSpeed();
39             if (speed > maxSpeed) {
40                 maxSpeed = speed;
41                 maxSpeedDay = e.day;
42             }
43         }
44         return maxSpeedDay;
45     }
46
47
48     // Darstellung als String
49     public String toString() {
50         String result = "Code-Tagebuch von " + name + "\n";
51
52         for (int i = 0; i < this.liste.size(); i++) {
53             Entry e = this.liste.get(i);
54             result += i + ". " + e.toString() + "\n";
55         }
```

```
56
57          return result;
58      }
59
60  }
```

Um über die Elemente in der Liste zu iterieren, können wir entweder eine Zählschleife verwenden (wie in der Methode `toString()`) oder eine kompakte Variante, die sogenannte **foreach**-Schleife (wie in der Methode `totalLoc`).
Wir können nun Beispiel-Objekte für Lauftagebücher erstellen und mit diesen arbeiten:

```
CoderDiary diary = new CoderDiary("Hugo");
diary.add(new Entry(new Date(2,11,2017),765,210,"Initiales Design"));
diary.add(new Entry(new Date(8,11,2017),140,20,"Tests"));
diary.add(new Entry(new Date(10,11,2017),8,75,"Debugging"));

StdOut.println(diary);

StdOut.println("Gesamtzahl: " + diary.totalLoc());

Entry e = new Entry(new Date(2,12,2017),10,60,"Juhuu es klappt!");
StdOut.println(e + "\nGeschwindigkeit: " + e.calculateSpeed() + "LOC/h");

StdOut.println(diary);
```

2.3.3 Die Anwendung CoderDiaryApp

Wir erstellen schließlich eine Anwendung, die es erlaubt ein Code-Tagebuch zu führen. Die Verwendung der Applikation erfolgt über die Kommandozeile mit Hilfe von verschiedenen Befehlen:

exit Beende die Ausführung der App.

add Füge einen Eintrag hinzu; dazu muss der Nutzer Informationen zu Datum, LOC, Dauer und Kommentar eingeben.

remove i Entferne den Eintrag an Position i.

show Zeige alle Einträge an.

speed Zeige den Tag mit der schnellsten durchschnittlichen Code-Erzeugungsrate.

loc Zeige die Gesamtzahl der Codezeilen für alle Einträge an.

Dabei wird vom Benutzer zunächst der Name abgefragt und dann in einer Interaktionsschleife die vom Benutzer eingegebenen Befehle abgearbeitet, bis der Benutzer `exit` eingibt.

```
1  public class CoderDiaryApp {
2
3      public static void main(String[] args) {
4          // Abfrage des Namens
5          StdOut.print("Name: ");
6          String name = StdIn.readLine();
7
```

```
8           // Erstellen eines CoderDiaries
9           CoderDiary d = new CoderDiary(name);
10
11          // Interaktionsschleife
12          boolean exit = false;
13          do {
14              StdOut.print("Aktion (exit, add, remove, show, loc, speed): ");
15              String action = StdIn.readString();
16              switch (action) {
17               case "exit":    exit = true; break;
18               case "add" :    Entry e = readEntry(); d.add(e); break;
19               case "remove" : int i = StdIn.readInt(); d.remove(i); break;
20               case "show":    StdOut.println(d);  break;
21               case "speed" :  StdOut.println(d.dayWithMaxSpeed()); break;
22               case "loc" :    StdOut.println(d.totalLoc()); break;
23               default:        StdOut.println("Aktion nicht bekannt!");
24              }
25          } while(!exit);
26      }
27
28      // Einlesen der Daten für einen neuen Eintrag
29      public static Entry readEntry() {
30          StdOut.print("Datum (dd.mm.yyyy): ");
31          String date = StdIn.readString();
32          StdOut.print("LOC:                ");
33          int loc = StdIn.readInt();
34          StdOut.print("Zeit:               ");
35          int time = StdIn.readInt();
36          StdOut.print("Kommentar:          ");
37          String comment = StdIn.readString();
38          return new Entry(new Date(date), loc, time, comment);
39      }
40  }
```

Die Anwendung dient uns hier als Fallstudie zum objekt-orientierten Modellieren. Sie hat aber noch einige Schwächen:

- Da die Einträge nicht persistent in eine Datei o.ä. geschrieben werden, stehen die Einträge nach Beenden der Anwendung nicht mehr zur Verfügung.
- Bei Datumswerte wird nicht überprüft, ob sie tatsächlich ein gültiges Datum repräsentieren.

Diesen Herausforderungen werden wir uns in späteren Kapiteln widmen.