Ergänzungen zum Buch

Dietmar Ratz, Jens Scheffler, Detlef Seese, Jan Wiesenberger

Grundkurs Programmieren in Java

Inhaltsverzeichnis

Willkommen

Willkommen zum Ergänzungsband zum „Grundkurs Programmieren in Java". Wir gehen davon aus, dass Sie durch unser Buch auf dieses Dokument gestoßen sind. Falls dem so ist, können wir nur zu diesem Durst nach zusätzlichen Informationen gratulieren. Wir hoffen, dass Ihnen die Web-Erweiterungen helfen werden, das erworbene Wissen zu vertiefen und anzuwenden.
Diese Dokument ist in zwei Teile unterteilt: Theorie und Praxis. Im Theorie-Teil werden wir auf Aspekte der Java-Programmierung eingehen, die im Buch leider keinen Platz mehr gefunden haben. Hierzu zählen unter anderem Java Annotations, Unit Tests und Entwurfsmuster.
Im Praxis-Teil werden wir unser Buchwissen anhand praktischer Programmieraufgaben anzuwenden lernen. Wir haben eine Menge über Java gelernt und sind nun theoretisch in der Lage, die ersten komplexeren Programme zu schreiben. Leider kann man eine Sprache anhand der Theorie genauso wenig erlernen wie das Autofahren – wir benötigen *Praxis*. Zu diesem Zweck werden wir verschiedene Aufgabenstellungen zu lösen versuchen und uns hierbei vor allem damit befassen, wie man an ein Problem

- systematisch herangeht,
- eine Lösung sucht und
- diese in Java programmiert.

Teil I

Theorie

Ergänzung 1

Strings für Fortgeschrittene

Wir wissen bereits, wie sich mit Hilfe der Klasse `String` grundlegende Operationen auf Zeichenketten durchführen lassen. In diesem Abschnitt werden wir uns nun mit einigen fortgeschrittenen Konzepten befassen, die seit der Version 1.4 in Java eingeführt wurden.

1.1 Reguläre Ausdrücke

Reguläre Ausruckücke stellen eine formale Sprache zur Beschreibung von Zeichenketten dar. Wer sich darunter etwas vorstellen kann, muss entweder höheres Semester Informatik oder aber ein Definitionsfanatiker sein. Lässt sich das Ganze nicht auch etwas verständlicher erklären?
Stellen wir uns einmal vor, wir kommen nach einem langen Tag nach Hause und finden eine Nachricht auf dem Anrufbeantworter vor: „Hallo, mein Name ist Herrmann Maier. Es ist meine erfreuliche Pflicht, Sie über einen Millionengewinn in der Lotterie in Kenntnis zu setzen. Bitte rufen Sie mich baldmöglichst zurück. Meine Telefonnummer ist ...“ – in diesem Moment explodiert der Anrufbeantworter.
Mit zitternden Fingern greifen wir zum Telefonbuch. Maier war der Name - wie schreibt man das doch gleich? Ist es mit „A“ oder eher mit „e“? Und wie war der Vorname noch einmal? Irgendetwas mit H, aber was?
Da wir uns nicht mehr sicher sind, wollen wir alle Möglichkeiten der Reihe nach durchgehen. Wir wissen:

- Der Nachname war Maier, wir wissen aber nicht, wie man ihn schreibt. Prinzipiell in Frage kommen also
  - Maier,
  - Meyer,
  - Mayer und

- Majer.

  Wenn man es richtig betrachtet, kennen wir eigentlich nur den ersten und die letzten zwei Buchstaben.
- Der Vorname beginnt mit einem H, aber wir können uns nicht mehr erinnern.
- wir wissen nicht, ob die entsprechende Person noch einen zweiten Vornamen hat, der im Telefonbuch geführt wird.

Glücklicherweise hat unser Telefonbuch eine CD-ROM, auf der wir nach gewissen Suchkriterien forschen können. Die Suchkriterien werden dem Computer in einem gewissen Format mitgeteilt – eben einem regulären Ausdruck. Somit ist alles klar – es handelt sich also lediglich um eine weitere Computersprache (wie Java), die wir lediglich erlernen müssen.

1.2 Ein konkretes Beispiel

Dieser Abschnitt soll keine komplette Einführung in die Welt regulärer Ausdrücke darstellen – für mehr Details empfehlen wir einen Blick in die Dokumentation der Klasse `java.util.regex.Pattern`. Wie wir anhand des folgenden Beispiels sehen werden, sind die Grundlagen allerdings nicht sonderlich schwer.
Das folgende Programm stellt eine vereinfachte Version unseres elektronischen Telefonbuchs dar:

```
1  public class Textsuche {
2
3    public static void main(String[] args) {
4      String[] namensliste = {
5        "Fritz Maier",
6        "Karl Hansen",
7        "Fred Mustermann",
8        "Horst Metzger",
9        "Hermann Meyer",
10       "Fritz H. Maurer",
11       "Hoerbi Maier",
12       "Hans Maler",
13       "Harry Mooshammer",
14       "Hurgan Malinkow"
15      };
16     String suchstring = "?????";
17     for(int i = 0; i < namensliste.length; i++) {
18       if (namensliste[i].matches(suchstring))
19         System.out.println("Gefunden: " + namensliste[i]);
20     }
21   }
22
23 }
```

In einem Feld namens `namensliste` haben wir eine Ansammlung von Namen abgespeichert – einer von ihnen ist unser Tor zu Ruhm und Reichtum. Um den

Namen zu finden, iterieren wir mit einer Schleife durch alle Feldelemente. Entspricht ein Feldeintrag unserem Suchkriterium (die Methode `matches` der Klasse `String` vergleicht einen `String` mit einem regulären Ausdruck), so geben wir den Namen auf dem Bildschirm aus.
Es verbleibt also nur noch, das Suchkriterium in der Variablen `suchstring` zu definieren. Im ersten Versuch wollem wir dem Computer lediglich mitteilen, dass im Namen ein großes M (wie Maier) vorkommen muss. Davor und danach können beliebig andere Zeichen stehen.
Wie sagen wir's dem Computer? Hierzu müssen wir wissen, dass der Punkt in einem regulärem Ausdruck für beliebige Zeichen steht. Folgt danach ein Stern, bedeutet dies „beliebig viele". Die Zeile

```
String suchstring = ".*M.*";
```

bedeutet also *beliebig viele beliebige Zeichen, dann ein großes M, danach wieder beliebig viele beliebige Zeichen*. Lassen wir unser Programm laufen, erhalten wir folgendes Suchergebnis:

Konsole

```
Gefunden: Fritz Maier
Gefunden: Fred Mustermann
Gefunden: Horst Metzger
Gefunden: Hermann Meyer
Gefunden: Fritz H. Maurer
Gefunden: Hoerbi Maier
Gefunden: Hans Maler
Gefunden: Harry Mooshammer
Gefunden: Hurgan Malinkow
```

Offensichtlich ist unser Suchmuster noch zu grob. Wir setzen deshalb noch voraus, dass der Vorname mit einem großen H beginnt:

```
String suchstring = "H.*M.*";
```

Wie wir sehen, schränkt dies die Suche schon deutlich ein:

Konsole

```
Gefunden: Horst Metzger
Gefunden: Hermann Meyer
Gefunden: Hoerbi Maier
Gefunden: Hans Maler
Gefunden: Harry Mooshammer
Gefunden: Hurgan Malinkow
```

Gehen wir nun noch davon aus, dass der Nachname (Maier) mit den Buchstaben „er" endet, also das Suchmuster

```
String suchstring = "H.*M.*er";
```

so lässt sich auch Herr Malinkow von der Suche ausschließen:

Konsole

```
Gefunden: Horst Metzger
Gefunden: Hermann Meyer
Gefunden: Hoerbi Maier
Gefunden: Hans Maler
Gefunden: Harry Mooshammer
```

Wie schränken wir unsere Namenssuche noch weiter ein? Nun, egal wie man den Namen Maier schreibt, zwischen dem *M* und dem *er* kommen immer nur zwei Buchstaben vor. In einem regulären Ausdruck können wir dies beschreiben, indem wir den Stern durch die Anzahl der Vorkommnisse (in geschweiften Klammern) ersetzen:

```
String suchstring = "H.*M.{2}er";
```

Mit dieser verfeinerten Suche lässt sich die Anzahl der Resultate so weit einschränken, dass wir sie auch leicht per Hand durchgehen können.

Konsole

```
Gefunden: Hermann Meyer
Gefunden: Hoerbi Maier
Gefunden: Hans Maler
```

1.3 Textersetzung

Wie im vorigen Kapitel bereits gesehen, unterstützt Java reguläre Ausdrücke für die Suche in Texten. Neben der Methode `match` gibt es im Paket `java.util.regex` Hilfsklassen, die auch komplexere Suchen auf Texten mit regulären Ausdrücken ermöglichen.
Doch schon mit dem Grundvorat an Hilfsmethoden in der Klassen `String` lassen sich einige erstaunliche Dinge bewerkstelligen. Die Methode `matches` ist dem Leser bereits ein Begriff. Wir wollen uns nun mit einer weiteren Möglichkeiten der regulären Ausdrücke vertraut machen: dem Ersetzen von Texten innerhalb eines Strings.
Das Ersetzen von Zeichenketten, heutzutage Standard in den einfachsten Texteditoren auf unserem Computer, war lange Zeit im Standard der Java-Sprache nicht enthalten. Ersetzen war nur auf Zeichenebene (Methode `replace`) möglich. Dies ist seit Java Version 1.4 anders geworden:

- Die Methode `replaceFirst` ersetzt *das erste* Auftreten eines regulären Ausdruckes durch einen anderen Wert. So gibt die Zeile

  ```
  System.out.println("Hund, Hemd".replaceFirst("H..d","Hand"));
  ```

  beispielsweise den Text

Konsole

```
Hand, Hemd
```

auf dem Bildschirm aus.

- Die Methode `replaceAll` ersetzt *alle* gefundenen Passagen durch einen anderen String. Die Zeile

```
System.out.println("Hund, Hemd".replaceAll("H..d","Hand"));
```

gibt also somit den Text

Konsole

```
Hand, Hand
```

auf dem Bildschirm aus.

1.4 Zusammenfassung

Reguläre Ausdrücke geben dem erfahrenen Programmierer ein mächtiges Werkzeug zur Verarbeitung von Zeichenketten in Java.[1] Neben dem Paket `java.util.regex` wurden Hilfsmethoden in die Klasse `String` aufgenommen, die Suche und Ersetzung in Texten wesentlich vereinfachen.[2] Reguläre Ausdrücke sind eine kleine Sprache an sich – wer sie verwenden will, muss sie sprechen lernen. Hat man sie jedoch einmal erlernt, lässt sich dieses Wissen auf viele andere Programmiersprachen (und auch Texteditoren wie beispielsweise **Emacs** oder **vi**) übertragen.

[1] Dies gilt auch für andere Sprachen wie Perl oder Tcl, in denen diese Ausdrücke traditionall verankert sind.

[2] Für weitere Hilfsmethoden, etwa zum Aufteilen von Text mittels der Methode `split`, sei auf die API-Dokumentation verwiesen

Ergänzung 2

Annotations in Java

2.1 Standard Annotations im Java SDK

Annotations sind eine der mächtigsten Errungenschaften der neueren Versionen von Java. Ihr Nutzen ist leider aber auch am schwersten im Rahmen eines Lehrtextes zu erklären. Beginnen wir also mit einem motivierenden Beispiel: Gegeben sei eine Klasse:

```
1 public class Dog {
2   public String belle() {
3     return "wuff";
4   }
5 }
```

Hund sei Bestandteil einer größeren Programmbibliothek, die zur Erstellung von Tier-basierten Anwendungen (Pet-Store, Zoo-Verwaltung, mein virtuelles Pony, ...) verwendet wird. Die Bibliothek beinhaltet eine Vielzahl nützlicher Hilfsklassen und –methoden, wie zum Beispiel

```
public static void randomBark(int amount, Dog... dogs) {
  Random rnd = new Random();
  for (int i = 0; i < amount; i++) {
    System.out.println(
    dogs[rnd.nextInt(dogs.length)].belle());
  }
}
```

Ein Anwender hat sich seine eigene Kindklasse `SmallDog` gebaut:

```
1 public class SmallDog extends Dog {
2   public String belle() {
3     return "wiff";
4   }
5 }
```

Er verwendet diese Kindklasse mit den verschidensten Hilfsmethode, wie zum Beispiel in

```
randomBark(10, new Dog(), new SmallDog());
```

Dank objektorientierter Programmierung funktioniert dies wunderbar – der kleine Hund bellt anders als der große:

Konsole

```
wiff
wuff
wuff
wiff
wiff
wiff
wuff
wiff
wuff
wuff
```

Eines Tages wirft der Programmierer der Grundklasse einen Blick auf die Schnittstelle und beschließt, den Mischmasch von deutsch-englischen Begriffen zu verringern. Aus diesem Grund benennt er die Method `belle` in `bark` um[1]:

```
1 public class Dog {
2   public String bark() {
3     return "wuff";
4   }
5 }
```

Was bedeutet dies in der Praxis für unseren Benutzer? Auf den ersten Blick nicht viel, denn die Kindklasse `SmallDog` wird auch weiter vom Java Compiler anstandslos übersetzt. Lassen wir unseren Einzeiler von oben jedoch nochmals laufen, sehen wir zu unserer Überraschung, dass urplötzlich all Hunde auf dieselbe Art und Weise bellen:

Konsole

```
wuff
wuff
wuff
wuff
wuff
wuff
wuff
wuff
wuff
wuff
```

Was ist passiert? `SmallDog` hatte die Methode `belle` überschrieben, um einen anderen Bell-laut zurückzuliefern. Durch die Namensänderung in der Superklasse ist diese Ersetzung nicht mehr gegeben. Für Java sind `bark` und `belle` nun

[1] In der Praxis ist so etwas übrigens eine Todsünde – eine externe Schnittstelle, die von anderen Programmieren verwendet wird, darf sich nicht einfach ändern. Siehe auch [12] für mehr Details.

zwei völlig unterschiedliche Methoden, die nichts mehr miteinander zu tun haben. Derartige Fehler in Programmen sind oftmals schwierig aufzuspüren, da sie der Compiler nicht finden kann. Sie beschränken sich auch nicht auf umbenannte Methoden; eine der Hauptquellen für diese Arten von Problemen sind simple Tippfehler beim Eingeben des Methodennamens. Gibt es denn keinen Weg, dass der Compiler uns dies melden kann?

2.1.1 Die `@Override` Annotation

Im obigen Beispiel möchten wir dem Compiler mitteilen, dass unsere Methode `belle` eine Methode in einer Superklasse überschreibt. Wir möchten Java einen Hinweis geben, dass ein bestimmtes Codestück eine besondere Bedeutung hat. Diese Anmerkung (oder englisch, Annotation) am Code würde in Java wie folgt aussehen:

```
1 public class SmallDog extends Dog {
2   @Override
3   public String belle() {
4     return "wiff";
5   }
6 }
```

Annotations beginnen mit dem „at"-Zeichen `@` und können vor diversen Code-Segmenten stehen, wie zum Beispiel der Deklararion von Methoden oder Konstruktoren, Klassen-Definitionen, oder sogar der Deklaration von lokalen Variablen[2]. Im Falle von `@Override` steht die Anmerkung vor einer deklarierten Methode, wie in Zeile 2 unseres Beispielprogrammes. Wenn wir versuchen, die Klasse `SmallDog` zu übersetzen, erhalten wir nun einen Kompilierfehler:

Konsole

```
SmallDog.java:2: method does not override or implement a method
                 from a supertype

  @Override
  ^
1 error
```

Was ist geschehen? Durch das einfügen der `@Override` Annotation haben wir dem Compiler mitgeteilt, dass die Methode `belle` entweder die Methode einer Superklasse überschreibt oder (ab Java 6) die Methode eines Interfaces implementiert. Der Java Compiler wird nun in allen Super-Klassen nach eine Methode `belle` suchen. Falls diese nicht existiert, bricht der Übersetzungsvorgang mit einer Fehlermeldung ab.
`@Override` ist ein sehr nützliches Werkzeug, um sich gegen Tippfehler und wechselnde Schnittstellen in Klassen zu schützen, auf die man baut. Manche Entwicklungsumgebungen wie Eclipse können heutzutage die Annotation sogar au-

[2] Eine Liste ist in der Klasse `java.lang.annotation.ElementType` gegeben

tomatisch einfügen oder den Entwickler warnen, wenn er anscheinend eine Annotation übersehen hat. Und wo wir gerade von Warnungen sprechen: dies ist die perfekte Überleitung zur nächsten Standard-Annotation aus dem Java SDK.

2.1.2 Die `@Deprecated` Annotation

Werfen wir noch einmal einen Blick auf unsere Basisklasse `Dog`, aber diesmal aus der Sicht ihres Autoren:

```
1 public class Dog {
2   public String belle() {
3     return "wuff";
4   }
5 }
```

Die Klasse hat einen englischen Namen, aber besitzt eine deutsche Methode `belle`. Ein solcher Mix aus deutschen und englischen Namen kann leicht verwirren, und so ist das Ziel des Autoren verständlich, die Sprache im Programm zu vereinheitlichen. Die Methode einfach umzubenennen war allerdings keine gute Idee, da dies die Vererbung in Kindklassen schädigen konnte. Wir können nicht einfach eine Methode herausschmeißen – wir müssen den Benutzern Zeit geben, ihren Code darauf umzustellen. Wie gehen wir also vor?
Der Vorgang, den wir zu diesem Zweck einsetzen, nennt sich **Deprecation** – sprich, wir markieren eine Schnittstelle als veraltet. [14] beschreibt drei häufige Gründe für das Veralten einer Schnittstelle:

- Die Schnittstelle ist unsicher, fehleranfällig oder ineffizient.
- Die Schnittstelle wird in der Zukunft entfernt werden.
- Die Schnittstelle verleitet dazu, schlechten Code zu schreiben.

In unserem Fall treffen die letzten zwei Gründe zu, da die Schnittstelle zu gemischtsprachigem (und daher schlecht lesbarem) Code verleitet und wir sie daher in Zukunft entfernen werden.
Bevor wir unsere Methode veralten, brauchen wir natürlich eine vernünftige Alternative, die alten Code nicht schädigt. Die folgende Klasse erfüllt diese Anforderung:

```
1  public class Dog {
2
3    public String belle() {
4      return "wuff";
5    }
6
7    public String bark() {
8      return belle();
9    }
10 }
```

Unsere Hunde-Klasse hat nun sowohl die alte Methode `belle` als auch die neue Methode `bark`. Die Standard-Implementierung momentan ist, alle Aufrufe auf

die alte Methode weiterzuleiten. Auf diese Art und Weise bleibt die Funktionalität in alten Klassen wie `SmallDog` erhalten. Neue Klassen können stattdessen `bark` überschreiben. Wie sollen wir aber dem Benutzer mitteilen, welche unserer Methoden veraltet ist? Die Antwort ist erneut das Ergänzen des Quelltextes durch eine Annotation – in diesem Falle `@Deprecated`:

```
@Deprecated
public String belle() {
  return "wuff";
}
```

Die `@Deprecated` Annotation läßt den Compiler wissen, dass die markierte Methode (oder Klasse, wenn sie vor einer Klassendefinition steht) veraltet ist. Wenn wir nun unsere Kindklasse übersetzen, erhalten wir die folgende Warnung:

Konsole

```
Note: SmallDog.java uses or overrides a deprecated API.
Note: Recompile with -Xlint:deprecation for details.
```

Wir können uns die Details dieser Warnung mit dem oben genannten Schalter `Xlint` anschauen (die meisten Entwicklungsumgebungen machen dies automatisch):

Konsole

```
C:\javabuch\Code\annotations>javac -Xlint SmallDog.java
SmallDog.java:3: warning: [deprecation] belle()
    in Dog has been deprecated
  public String belle() {
                ^
1 warning
```

Mit dieser einfachen Änderung haben wir es den Benutzern unserer Klasse leicht gemacht, veralteten Code zu erkennen und zu vermeiden. Es sollte an dieser Stelle auch erwähnt werden, dass es guter Stil ist, in der Javadoc Dokumentation zu erklären, wie diese Vermeidung stattfinden kann. Das folgende Programm zeigt die komplette Klasse mit veraltetem Code und inklusive korrekter Dokumentation:

```
1  public class Dog {
2
3    /**
4     * @deprecated verwende stattdessen {@link #bark()}
5     */
6    @Deprecated
7    public String belle() {
8      return "wuff";
9    }
10
11   public String bark() {
12     return belle();
13   }
14 }
```

2.1.3 Die `@SuppressWarnings` Annotation

Im vorherigen Abschnitt wurde gezeigt, wie man durch die Verwendung der `@Deprecated` Annotation dem Java Compiler zusätzliche Hinweise liefern konnte, die in Warnungen während des Übersetzungsvorganges resultierten. Warnungen sind nützlich, denn sie weisen den Programmierer auf Dinge hin, die eventuell ein Problem darstellen könnten. Zu viele (und unnütze) Warnungen jedoch stellen ein Problem dar, da sie wie ein Rauschen in der Telefonleitung sind: sie stören und lenken von den wirklich wichtigen Dingen ab, auf die man achten sollte. Nehmen wir zum Beispiel die folgende Methode:

```
public static <K, V> Map<K, V> of(
    K key1,
    V value1,
    Object... moreKeyVals) {
  Map<K, V> result = new HashMap<K, V>();
  result.put(key1, value1);
  for (int i = 0; i < moreKeyVals.length; i += 2) {
    result.put(
        (K) moreKeyVals[i],
        (V) moreKeyVals[i + 1]);
  }
  return result;
}
```

Die Methode ist ein einfaches Hilfsmittel, um eine typsichere `java.util.Map` zu erstellen, die mit einigen Werten initialisiert ist. Das erste Wertepaar ist explizit gegeben (damit der Compiler die generischen Typen erschließen kann); es können aber beliebig viele zusätzliche Werte dank des varargs Parameters übergeben werden. Auf den ersten Blick erscheint dies logisch; der Java Compiler liefert uns jedoch einige Warnungen:

Konsole

```
C:\...>javac -Xlint SuppressWarningsExample.java
SuppressWarningsExample.java:23:
    warning: [unchecked] unchecked cast
found   : java.lang.Object
required: K
              (K) moreKeyVals[i],
                              ^
SuppressWarningsExample.java:24:
     warning: [unchecked] unchecked cast
found   : java.lang.Object
required: V
                  (V) moreKeyVals[i + 1]);
                                  ^
```

Wo liegt das Problem, und wie lösen wir es? Wie wir in Abschnitt 11.2 des Buches gelernt haben, sind generische Methoden ein wunderbares Mittel, um

Typsicherheit beim Übersetzen zu garantieren. Diese Typsicherheit ist zur Laufzeit allerdings nicht garantiert – die Typinformationen sind nicht Bestandteil der Laufzeitumgebung, welche all unsere Parameter als allgemeine Objekte betrachtet. Aus diesem Grund finden die Typecasts von `moreKeyVals[i]` und `moreKeyVals[i+1]` nicht wirklich statt[3], und der Compiler kann zu Übersetzungszeiten nicht garantieren, dass `moreKeyVals` auch immer vom richtigen Typ sein wird.
Die Lösung des Problems ist nicht offensichtlich: wie sollen wir den Typ der extra Parameter garantieren, wenn wir zwei verschiedene Typen (`K` und `V`) in dasselbe Feld stecken? Eine Möglichkeit wäre es, einen komplett anderen Ansatz zu wählen, in dem sich dieses Problem nicht ergibt (siehe etwa das Builder Entwurfsmuster in [15]). Wir wollen stattdessen die Prüfung des Typs zur Laufzeit einfach selbst implementieren: innerhalb unserer Schleife überprüfen wir die jeweiligen Werte mit `key1` und `value1`. Stimmen die Klassen nicht überein, werfen wir eine Exception[4].

```
public static <K, V> Map<K, V> of(
    K key1,
    V value1,
    Object... moreKeyVals) {
  Map<K, V> result = new HashMap<K, V>();
  result.put(key1, value1);
  for (int i = 0; i < moreKeyVals.length; i += 2) {
    K key2 = (K) moreKeyVals[i];
    V value2 = (V) moreKeyVals[i + 1];
    if (!key1.getClass().equals(key2.getClass())) {
      throw new ClassCastException("key, index "
          + i / 2 + ": " + key2.getClass());
    }
    if (!value1.getClass().equals(value2.getClass())) {
      throw new ClassCastException("value, index "
              + i / 2 + ": " + value2.getClass());
    }
    result.put(key2, value2);
  }
  return result;
}
```

Mit Hilfe dieser Ergänzungen können wir zur Laufzeit sicherstellen, dass die übergebenen Werte vom richtigen Typ sind. Leider beschwert sich jedoch der Compiler noch immer:

Konsole

```
SuppressWarningsExample.java:22:
    warning: [unchecked] unchecked cast
found   : java.lang.Object
required: K
      K key2 = (K) moreKeyVals[i];
```

[3] Man nennt dieses Phänonmen auch **Type Erasure**
[4] Es sei dem Leser als Übung überlassen, diesen Test auch für Kindklassen und Interfaces zu implementieren.

```
                                ^
SuppressWarningsExample.java:23:
    warning: [unchecked] unchecked cast
found   : java.lang.Object
required: V
      V value2 = (V) moreKeyVals[i + 1];
                                   ^
2 warnings
```

Irgendwie ist es ja verständlich: wie soll der Java Compiler wissen, dass wir dieses Problem erkannt und es manuell behoben haben? Die Antwort liegt in der `@SuppressWarnings` Annotation, die wir wie folgt verwenden können:

```
@SuppressWarnings("unchecked")
public static <K, V> Map<K, V> of(
    K key1,
    V value1,
    Object... moreKeyVals) {
  // ...
}
```

Durch das Verwenden der Annotation an der Methode `of` machen wir dem Compiler klar, dass Probleme der Form „unchecked"[5] nicht ausgegeben werden sollen. Wir schalten diese spezielle Warnung also aus, da sie uns bekannt ist und wir entsprechende Vorkehrungen getroffen haben.
Natürlich ist das Ausschalten der Warnung in der kompletten Methode ein ziemlich grober Hammer – wenn wir irgendwo sonst in `of()` einen weiteren, ungeplanten Fehler derselben Art gemacht haben, fällt dieser von jetzt an unter den Tisch. Aus diesem Grund ist es zu bevorzugen, das kleinste annotierbare Codesegment zu wählen – in diesem Fall die Wertzuweisung, in der die Warnung aufgetreten ist:

```
// Die folgenden Casts sind ok, da wir die Klassen-
// zugehoerigkeit manuell pruefen
@SuppressWarnings("unchecked") K key2 = (K) moreKeyVals[i];
@SuppressWarnings("unchecked") V value2 =
    (V) moreKeyVals[i + 1];
```

Auf diese Art und Weise ist der Platz der Warnung so weit wie möglich eingeschränkt auf

- die genaue Art des Problems, dass wir zu ignorieren gedenken, und
- die genaue Stelle, and der wir das Problem ignorieren.

So wie es sich normalerweise gehört, die Verwendung von `@Deprecated` im Javadoc zu erläutern, sollte ein Programmierer auch jeweils genau dokumentieren, warum eine gewisse Compiler-Warnung ausgeschaltet ist. Dies haben wir in diesem Beispiel getan. Die komplette Methode sieht nun wie folgt aus:

[5] Die Art der Warnung stand in eckigen Klammern in der Warnungsmeldung.

```
/**
 * Erzeuge eine Map mit einem oder mehr Werte-Paaren.
 * Eine typische Verwendungsweise waere z.B.
 * Map<String, Integer> m = of("a", 1, "b", 2)
 * @param key1 der erste Schluesselwert
 * @param value1 der zu key1 passende Wert
 * @param moreKeyVals eine Liste von zusaetzlichen
 *    Wertepaaren
 * @exception ClassCastException falls einer der variablen
 *   Parameter vom falschen Typ ist
 * @exception NullPointerException falls einer der Parameter
 *    null ist.
 */
public static <K, V> Map<K, V> of(
    K key1,
    V value1,
    Object... moreKeyVals) {

  // Key1 und Value1 sollen nicht null sein
  if (key1 == null || value1 == null) {
    throw new NullPointerException();
  }

  // Initialisiere Map mit den ersten zwei Werten
  Map<K, V> result = new HashMap<K, V>();
  result.put(key1, value1);

  // Gehe durch die Variablen Parameter
  for (int i = 0; i < moreKeyVals.length; i += 2) {

    // Die folgenden Casts sind ok, da wir die Klassen-
    // zugehoerigkeit manuell pruefen
    @SuppressWarnings("unchecked") K key2 = (K) moreKeyVals[i];
    @SuppressWarnings("unchecked") V value2 =
        (V) moreKeyVals[i + 1];

    // Pruefe Klassenzugehoerigkeit
    if (!key1.getClass().equals(key2.getClass())) {
      throw new ClassCastException("key, index "
          + i / 2 + ": " + key2.getClass());
    }
    if (!value1.getClass().equals(value2.getClass())) {
      throw new ClassCastException("value, index "
        + i / 2 + ": " + value2.getClass());
    }

    // Fuege zur Map hinzu
    result.put(key2, value2);
  }

  // Fertig :-)
  return result;
}
```

2.2 Annotations zum Selberbauen

Wir haben auf den vergangenen Seiten drei Annotations kennengelernt, die uns neue Möglichkeiten erschlossen, den Compiler bei seiner Fehlersuche zu unterstützen. Dennoch sind drei Standard-Annotationen im gesamten JDK nicht gerade viel – gibt's da nicht noch mehr?
Tatsächlich werden Annotations heutzutage in einer Vielzahl von Bibliotheken sehr erfolgreich eingesetzt (siehe zum Beispiel die Annotations für Junit in Abschnitt 3.3). Andere Beispiele wären etwa die Datenbank-nahen Standards JDO und JPA, die ebenfalls Annotations zum Einsatz bringen. Ferner werden in diversen Standardisierungs-Kommittees (siehe [13]) neue Annotations diskutiert, die die Fehlersuche beim Übersetzen noch weiter erleichtern würden[6]. Was all diese Annotationen gemein haben, ist dass sie keine Erweiterungen von Java benötigt haben, um implementiert zu werden. Jeder der will, kann Annotations schreiben. Auch wir.

2.2.1 CoDo: *Co*de *Do*kumentieren durch Annotations

Nehmen wir an, wir haben ein simples Interface `Addierer` geschrieben:

```
1  public interface Addierer {
2
3    public Integer addValues(Integer v1, Integer v2);
4    public Integer addToInt(Integer v1, Number v2);
5
6  }
```

Unsere erste implementierung, `AddiererImpl`, sieht relativ simpel aus:

```
public class DemoProgram {

  public static class AddiererImpl implements Addierer {
    @Override
    public Integer addValues(Integer v1, Integer v2) {
      return v1 + v2;
    }

    @Override
    public Integer addToInt(Integer v1, Number v2) {
      return addValues(v1, v2.intValue());
    }
  }
```

Wie der Leser wahrscheinlich sieht, kann bei der Verwendung diese Klasse so einiges schiefgehen. Beispielsweise:

- Wenn einer der Parameter **`null`** ist, wird die Methode `addValues` eine `NullPointerException` werfen.

[6] Ein gutes Beispiel hierfür ist die Antwort auf die Frage, wann ein Parameter **`null`** sein darf. Wenn man dem Übesetzer irgendwie klarmachen kann, dass ein Parameter niemals null sein darf, kann er dann zusätzliche mögliche Probleme automatisch finden?

- Ferner kann bei der Methode `addToInt` Präzision verloren gehen, z.B. wenn eine der angegebenen Zahlen ein `Double` mit Wert 2.5 ist.

Die Implementierung des Interface beruht also auf gewissen impliziten Annahmen (preconditions), dass etwa die Parameter nicht **`null`** sind oder dass nur bestimmte Unterklassen von `Number` miteinander kombiniert werden. Diese Dinge werden normalerweise im Javadoc beschrieben, doch wir wollen heute einen anderen Weg beschreiben: statt im Javaddoc, wollen wir die entsprechenden Parameter mit Annotations versehen:

```
1  import codo.ParameterIs.NotNull;
2  import codo.ParameterIs.RestrictedType;
3
4  public interface Addierer {
5
6    public Integer addValues(@NotNull Integer v1, @NotNull Integer v2);
7    public Integer addToInt(
8        @NotNull Integer v1,
9        @RestrictedType(of = {
10           Integer.class,
11           Short.class,
12           Byte.class }) Number v2);
13
14 }
```

In dieser Version des Addierers ist klar, dass **`null`** kein erlaubter Wert für einen Parameter ist. Ferner wird auch dokumentiert, dass nicht alle Unterklassen von `Number` ein gültiger Parameter sind. Diese Annotationen lesen sich nicht nur recht einfach, sie haben auch den Vorteil dass sie durch das Programm hinweg konsistent eingesetzt und maschniell ausgewertet werden können. Unser einziges Problem: die Annotationen `NotNull` und `RestrictedType` existieren bislang noch nicht!

2.2.2 Wir schreiben neue Annotations

Die Definition einer neuen Annotation ist nicht viel anders als die Definition einer Klasse oder eines Interfaces – auch wenn der Code auf den ersten Blick ein wenig seltsam ausschaut. Um dies zu zeigen, packen wir `NotNull` und `RestrictedType` in einer äußeren Klasse `ParameterIs` ein, und definieren sie quasi wie innere Klassen:

```
package codo;

import java.lang.annotation.ElementType;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;

/**
 * Eine Sammlung von Annotations die zusaetzliche
 * Aussagen ueber einen Parameter machen.
```

```
 */
public class ParameterIs {

  /**
   * Die Klasse selbst ist nicht instantiierbar.
   */
  private ParameterIs(){}
```

Beginnen wir mit `NotNull`. Unsere Definition wird aus drei Teilen bestehen:

- `@Retention`: Wir müssen dem Compiler mitteilen, wie „tief" eine Annotation in den übersetzten Code eingebaut werden soll. Ist die Annotation nur zur Zeit des Übersetzens notwendig (`RetentionPolicy.SOURCE`), oder soll sie der virtuellen Maschine zur Laufzeit zur Verfügung stehen (`RetentionPolicy.RUNTIME`)? Wird die `Retention` nicht angegeben, wählt der Übersetzer als Standard einen Mittelweg (`RetentionPolicy.CLASS`): die Annotation ist zur Laufzeit nicht zugänglich, wird aber in der generierten Binärdatei aufgeführt. Für unserer Annotationen wählen wir `RetentionPolicy.RUNTIME`.
- `@Target`: An welchen Teilen des Codes soll unsere Annotation einsetzbar sein? Klassen-Definitionen? Methoden? Package-Deklarationen? Für unsere Annotation wählen wir `ElementType.PARAMETER`, also die Deklaration von Parametern in Methoden.
- `@interface`: Die Definition der Annotation selber sieht ähnlich wie die einer Klasse (**`class`**) oder eines Interfaces (**`interface`**) aus; jedoch kommt in der Deklaration ein neues Schlüsselwort (`@interface`) zum Einsatz.

Betrachten wir also die fertige Annotation:

```
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.PARAMETER)
public static @interface NotNull {}
```

Unser Code liest sich wie folgt: wir haben eine Annotation (`@interface`) `NotNull` definiert, die auf Methoden-Parameter (`PARAMETER`) abzielt (`@Target`), und die zur Laufzeit (`RUNTIME`) erhalten bleibt (`@Retention`). So weit so gut – wie schaut es aber aus, wenn unsere Annotation noch zusätzliche Parameter braucht, wie im Falle von `@RestrictedType`?

```
public Integer addToInt(
    @NotNull Integer v1,
    @RestrictedType(of = {
        Integer.class,
        Short.class,
        Byte.class }) Number v2);
```

Wir wollen unserer Annotation zwei Parameter verpassen: einen Parameter `of`, der einen Liste von Klassen akzeptiert, und einen booleschen Parameter `isNullable`, der besagt, dass das annotierte Element **`null`** sein darf. Werfen wir einen Blick auf den resultierenden Code:

```
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.PARAMETER)
public static @interface RestrictedType {
  Class<?>[] of();
  boolean isNullable() default false;
}
```

Die Definition der Parameter sieht beinahe aus wie das Deklarieren von Methoden in Interfaces, aber es gibt einige Unterschiede:

- Die „Methoden" sind parameterlos.
- Die Methoden dürfen nicht beliebige Rückgabewerte haben; nur primitive Typen, `String`, `Class`, Enums, Annotations und Felder dieser Typen sind erlaubt.
- Die Methoden dürfen keine **`throws`**-Klausel haben.
- Für optionale Parameter wird ein default-Wert über das Schlüsselwort **`default`** angegeben.

Betrachten wir unsere fertige Klasse noch einmal in ihrer Gesamtheit:

```
1  package codo;
2
3  import java.lang.annotation.ElementType;
4  import java.lang.annotation.Retention;
5  import java.lang.annotation.RetentionPolicy;
6  import java.lang.annotation.Target;
7
8  /**
9   * Eine Sammlung von Annotations die zusaetzliche
10  * Aussagen ueber einen Parameter machen.
11  */
12 public class ParameterIs {
13
14   /**
15    * Die Klasse selbst ist nicht instantiierbar.
16    */
17   private ParameterIs(){}
18
19   /**
20    * Ein hiermit markierter parameter darf niemals
21    * Null sein.
22    */
23   @Retention(RetentionPolicy.RUNTIME)
24   @Target(ElementType.PARAMETER)
25   public static @interface NotNull {}
26
27   /**
28    * Beschraenkt einen Typ auf eine gewisse
29    * Menge von Unterklassen.
30    */
31   @Retention(RetentionPolicy.RUNTIME)
32   @Target(ElementType.PARAMETER)
33   public static @interface RestrictedType {
```

```
34      Class<?>[] of();
35      boolean isNullable() default false;
36    }
37  }
```

Das Verfassen neuer Annotationen fühlt sich zugegebenermaßen ein wenig merkwürdig an (der Autor muss die genaue Syntax auch jedes Mal nachschlagen, wenn er eine neue Annotation schreibt). Die gute Nachricht ist, dass man in der Praxis viel weniger Zeit mit dem Verfassen, und viel mehr Zeit mit der Verwendung der fertigen Annotation verbringt – und das wollen wir nun auch tun…

2.2.3 Auswerten von Annotations

Annotations werden in zwei verschiedenen Formen verwendet: ein Programmierer wird sie entweder zur Laufzeit auswerten wollen, oder aber während der Übersetzung eines Programmes. Letzteres ist recht komplex, und es sei deshalb auf [17] verwiesen. Zum Abschluss dieses Kapitels wollen wir eine Klasse `CodoProxy` schreiben, die für die Implementierung eines Interface automatisch die Überprfung der Preconditions übernimmt. Im Falle unseres `Addierer` Interface soll die Verwendung wir folgt aussehen:

```
1  import codo.CodoProxy;
2
3  public class DemoProgram {
4
5    public static class AddiererImpl implements Addierer {
6      @Override
7      public Integer addValues(Integer v1, Integer v2) {
8        return v1 + v2;
9      }
10
11     @Override
12     public Integer addToInt(Integer v1, Number v2) {
13       return addValues(v1, v2.intValue());
14     }
15   }
16
17   public static void main(String[] args) {
18     Addierer proxied = CodoProxy.makeProxy(Addierer.class,
19         new AddiererImpl());
20
21     // Normalfall
22     System.out.println(proxied.addValues(1, 2));
23     System.out.println(proxied.addToInt(1, (byte) 2));
24
25     // Parameter ist null
26     try {
27       System.out.println(proxied.addValues(1, null));
28     } catch (NullPointerException expected) {
29       expected.printStackTrace();
30     }
```

```
31
32      // Falsche Subklasse
33      System.out.println(proxied.addToInt(1, 2.5));
34    }
35
36  }
```

Durch den Aufruf

```
Addierer proxied = CodoProxy.makeProxy(Addierer.class,
    new AddiererImpl());
```

haben wir eine sogenannte Proxy-Klasse erzeugt[7]. Der Proxy stülpt sich über eine konkrete `Addierer` Implementierung und leitet den Aufruf der jeweiligen Methode an die unterliegende Klasse weiter – vorrausgesetzt, er findet in den Annotations keine verletzte Vorbedingung! Der Aufruf unseres Beispielprogrammes würde also folgende Ausgabe erzeugen:

Konsole

```
3
3
java.lang.NullPointerException: Parameter 1
        at codo.CodoProxy.checkNullness(CodoProxy.java:21)
        at codo.CodoProxy.access$000(CodoProxy.java:12)
        at codo.CodoProxy$1.invoke(CodoProxy.java:61)
        at $Proxy0.addValues(Unknown Source)
        at DemoProgram.main(DemoProgram.java:26)
Exception in thread "main" java.lang.ClassCastException:
    Parameter 1: class java.lang.Double
        at codo.CodoProxy.checkRestrictedType(CodoProxy.java:43)
        at codo.CodoProxy.access$100(CodoProxy.java:12)
        at codo.CodoProxy$1.invoke(CodoProxy.java:62)
        at $Proxy0.addToInt(Unknown Source)
        at DemoProgram.main(DemoProgram.java:32)
```

Um den `CodoProxy` zu implementieren, müssen wir ein paar Hilfsklassen kennen, die uns Java zur Verfügung stellt:

- Die Klasse `java.lang.annotation.Annotation` repräsentiert eine Annotation, mit der ein Stück Code in Java verziert wurde. Sie kann auf eine konkrete Subklasse (in unserem Fall `NotNull` und `RestrictedType`) gecastet werden, um so an die gesetzten Werte zu kommen.
- Die Klasse `java.lang.reflect.Method` repräsentiert die Methode einer Klasse. `Method` hat eine Methode `invoke`, mit der die Methode auf einem Objekt aufgerufen werden kann.
- `java.lang.reflect.Method` hat ferner eine Methode `getParameterAnnotations`, mit der man an alle Annotations kommen

[7] Für diejenigen, die Kapitel 4 schon kennen: `Proxy` ist ein weiteres bekanntes Entwurfsmuster.

kann, mit denen die Parameter der Methode verziert sind. Der Rückgabewert ist ein zweidimensionales Feld. Die erste Dimension entspricht den Parametern[8].

Nehmen wir also einmal an, wir haben eine Methode `m` und ein Feld von Parametern `parameters`, mit der sie aufgerufen werden soll, und wir wollen die `NotNull` Annotation überprüfen. Im ersten Schritt müssen wir dazu an die Liste aller Annotations herankommen:

```
Annotation[][] all = m.getParameterAnnotations();
```

Wir iterieren nun durch die erste Dimension des Feldes, die den Parametern `parameters` entspricht, und werfen einen Blick auf jede Annotation `a`, die für den entsprechenden Parameter definiert ist:

```
for (int i = 0; i < all.length; i++) {
  for (Annotation a : all[i]) {
```

Wir wollen eine `NullPointerException` werfen, wenn der Parameter **`null`** ist und es sich tatsächlich um eine `NotNull` Annotation handelt. Letzterer Check kann durch ein simples **`instanceof`** bewerkstelligt werden:

```
if (parameters[i] == null &&
    a instanceof NotNull) {
  throw new NullPointerException("Parameter " + i);
}
```

Fassen wir die Implementierung noch einmal zusammen:

```
private static void checkNullness(
    Method m, Object[] parameters) {
  Annotation[][] all = m.getParameterAnnotations();
  for (int i = 0; i < all.length; i++) {
    for (Annotation a : all[i]) {
      if (parameters[i] == null &&
          a instanceof NotNull) {
        throw new NullPointerException("Parameter " + i);
      }
    }
  }
}
```

Wir haben Parameter für Parameter überprüft, ob eine `NotNull` Annotation besteht, und ob der entsprechende Wert auf `Null` gesetzt ist. War das nicht der Fall, haben wir eine `Exception` geworfen. Die Überprüfung von `RestrictedType` funktioniert genauso; lediglich die zu überprüfende Bedingung ist ein wenig komplexer. Für den konkreten Ablauf (und die quasi-magische Erzeugung der Proxy-Klasse in `makeProxy` sei auf den folgenden Code verwiesen:

```
1 package codo;
2
3 import java.lang.annotation.Annotation;
4 import java.lang.reflect.InvocationHandler;
```

[8] Parameter 0 ist der erste Parameter der Methode, und so weiter.

```
5  import java.lang.reflect.Method;
6  import java.lang.reflect.Proxy;
7  import java.util.Arrays;
8
9  import codo.ParameterIs.NotNull;
10 import codo.ParameterIs.RestrictedType;
11
12 public class CodoProxy {
13
14   private static void checkNullness(
15       Method m, Object[] parameters) {
16     Annotation[][] all = m.getParameterAnnotations();
17     for (int i = 0; i < all.length; i++) {
18       for (Annotation a : all[i]) {
19         if (parameters[i] == null &&
20             a instanceof NotNull) {
21           throw new NullPointerException("Parameter " + i);
22         }
23       }
24     }
25   }
26
27   private static void checkRestrictedType(
28       Method m, Object[] parameters) {
29     Annotation[][] all = m.getParameterAnnotations();
30     for (int i = 0; i < all.length; i++) {
31       for (Annotation a : all[i]) {
32         if (a instanceof RestrictedType) {
33           RestrictedType restriction = (RestrictedType) a;
34           if (parameters[i] == null) {
35             if (restriction.isNullable()) {
36               continue;
37             } else {
38               throw new NullPointerException("Parameter " + i);
39             }
40           }
41           if (!Arrays.asList(restriction.of()).contains(
42               parameters[i].getClass())) {
43             throw new ClassCastException(
44                 "Parameter " + i + ": " + parameters[i].getClass());
45           }
46         }
47       }
48     }
49   }
50
51   @SuppressWarnings("unchecked")
52   public static <S, T extends S> S makeProxy(
53       final Class<S> interfaceClass, final T innerObject) {
54     return (T) Proxy.newProxyInstance(
55         innerObject.getClass().getClassLoader(),
56         new Class[]{interfaceClass},
57         new InvocationHandler(){
58           @Override
59           public Object invoke(Object o, Method m, Object[] parameters)
```

```
60                  throws Throwable {
61                checkNullness(m, parameters);
62                checkRestrictedType(m, parameters);
63                return m.invoke(innerObject, parameters);
64              }});
65      }
66
67  }
```

2.3 Zusammenfassung

Durch die Verwendung von Annotations hat der Programmierer die Möglichkeit, seinen Code mit zusätzlicher Meta-Information zu versehen. Diese Information kann vom Compiler benutzt werden, um zusätzliche Prüfungen durchzuführen; sie kann aber auch zur Laufzeit zur Verfügung stehen. Java liefert einige nützliche Annotations im Standard JDK. Darüber hinaus gibt es viele interessante Open Source Bibliotheken, die sich dieses machtvolle Werkzeug zunutze machen.

Ergänzung 3

JUnit oder Die Kunst, fehlerfreien Code zu schreiben

Stellen Sie sich einen Moment vor, Sie sind Teil eines Teams von Programmierern, die ein komplexes kommerzielles System erstellen. Gegen Ende des Projekts wird der verfasste Code viele Millionen Zeilen umfassen, von denen Sie selbst nicht wenige erstellt haben. Jeder von Ihnen ist für seinen Code verantwortlich und muss bei Bedarf in der Lage sein, vom Kunden gemeldete Fehler zu finden und zu eliminieren. Wie gehen Sie voran?
Um es gleich vorwegzunehmen: Dieses Kapitel wird (und kann) niemandem beibringen, keine Fehler zu machen. Softwareentwickler sind aus Fleisch und Blut und sind somit nicht vor Bugs gefeit. Allerdings gibt es gewisse Techniken, mit denen ein Programmierer dafür sorgen kann,

- dass zumindest keine offensichtlichen Programmierfehler in seinem Code stecken,
- dass er denselben Fehler nicht zweimal macht und
- dass er beim Versuch, den einen Fehler zu beheben, keine neuen Bugs in den Code einbaut[1].

Im weiteren Verlauf des Kapitels werden wir eine Bibliothek namens *JUnit* verwenden. Diese Bibliothek ist nicht Bestandteil des Standard JDK; sie kann aber kostenlos unter [9] heruntergeladen werden. Dieses Kapitel konzentriert sich auf die ,,klassische" Version (JUnit 3), aber wir werden auch kurz einige der Erweiterungen von Junit 4.5 behandeln (siehe Seite 45).

[1] Letzteres kann man natürlich nicht garantieren, aber wir können die Wahrscheinlichkeit zumindest verringern.

3.1 Assertions für Fortgeschrittene

Im Kapitel über Exceptions haben wir das seit Java 1.4 bestehende Schlüsselwort **assert** kennengelernt. Mit Hilfe dieses Konstrukts konnten wir innerhalb des Programmcodes Bedingungen überprüfen, die für das Funktionieren eines Codestückes erfüllt sein mussten.
Auch das JUnit-Framework unterstützt Zusicherungen. Im Gegensatz zum **assert**-Schlüsselwort tut es dies allerdings durch statische Methoden in der Klasse `JUnit.framework.Assert`. Das folgende Beispielprogramm implementiert den „Kehrwehrt–Test" aus dem Buch auf Basis von JUnit:

```
1  import junit.framework.Assert;
2  import junit.framework.AssertionFailedError;
3  import Prog1Tools.IOTools;
4
5  public class AssertionTest {
6
7    public static double kehrwert (double x) {
8      Assert.assertTrue("/ by 0", x != 0);
9      return 1 / x;
10   }
11
12   public static void main(String[] summand) {
13     double x = IOTools.readDouble("x = ");
14     try {
15       System.out.println (kehrwert(x));
16     }
17     catch(AssertionFailedError e) {
18       System.err.println(e.getMessage());
19     }
20   }
21 }
```

Beachten Sie die Änderungen:

- Anstelle des **assert**-Schlüsselwortes verwenden wir die Methode `Assert.assert`. Wir übergeben dieser Methode die zu überprüfende Bedingung plus einen auszugebenden Fehlertext:

  ```
  Assert.assertTrue("/ by 0", x != 0);
  ```

- Anstelle von `java.lang.AssertionError` fangen wir nun `JUnit.framework.AssertionFailedError` ab:

  ```
  catch(AssertionFailedError e) {
    System.err.println(e.getMessage());
  }
  ```

An dieser Stelle werden Sie sich wahrscheinlich zu Recht fragen: „Na und? Worin liegt der große Unterschied?"
Tatsächlich hat sich für den Programmierer bislang nicht viel geändert (außer dem Syntax). Wir haben ein von der Sprache selbst unterztütztes Schlüsselwort

aus dem Programm genommen und durch einen Methodenaufruf ersetzt. Dieses bringt sowohl Vor- als auch Nachteile:

- Der Nachteil ist, dass sich die JUnit-Assertions nicht einfach abschalten lassen. Java-Assertions lassen sich schlicht und ergreifend in der virtuellen Machine aktivieren oder deaktivieren[2]. Insbesondere in extrem rechenintensivem Code kann dies die Ausführung verlangsamen.
- Der Vorteil ist, dass JUnit-Assertions auch mit älteren Java-Versionen (also vor 1.3) funktionieren. Diese kommen selbst heutzutage noch in Projekten vor[3] – wir bleiben also abwärtskompatibel.

Diese Argumente allein rechtfertigen natürlich noch nicht den Wechsel zur Klasse `Assert`. Der Hauptvorteil dieser Klasse ist bislang allerdings noch nicht erwähnt worden: sie liefert *zusätzliche* Assertion-Methoden:

```
1  import junit.framework.Assert;
2
3  public class AssertionTest2 {
4
5    public static void main(String[] args) {
6      Assert.assertNotNull("Parameter-Arrary ist null",args);
7      // assert args != null : "Parameter-Array ist null"
8      for(int i = 0; i < args.length; i++) {
9        Assert.assertNotNull("Parameter " + i + " ist null",args[i]);
10       // assert args[i] != null : ("Parameter " + i + " ist null")
11       System.out.println(args[i]);
12     }
13   }
14 }
```

In diesem Programm geben wir den Inhalt eines übergebenen Parameter-Arrays auf dem Bildschirm aus. Damit das Programm funktioniert, nehmen wir als gegeben hin, dass das Feld und die Elemente des Feldes nicht **`null`** sind.[4] Würden wir Java-Assertions verwenden, würden wir diese Prüfung in einen boolschen Ausdruck umwandeln:

```
// assert args[i] != null : ("Parameter " + i + " ist null")
```

Wie man sieht, verringert sich die Lesbarkeit mit komplexeren Ausdrücken beträchtlich. Betrachten wir nun die Zusicherung in JUnit-Form:

[2] Manche Programmierer würden übrigens argumentieren, dass es sich hierbei eher um einen Vorteil handelt. Assertions überprüfen Bedingungen im Code, die immer erfüllt sein sollten. Aus diesem Grund kann es zur Stabilität des Codes nur beitragen, sie immer angeschaltet zu lassen. Die Philosophie ist: wenn das Programm beim ersten Fehler abstürzt, kann sich der Fehler nicht weiter ins Programm fortziehen. Er kann leichter erkannt und behoben werden.

[3] Insbesondere in älteren Projekten, die hauptsächlich in Wartung sind.

[4] Wenn Sie an dieser Stelle empört aufschreien, dies könne in einer `main`-Methode nicht vorkommen, so sind sie genau in unsere Falle getappt. Es ist korrekt, dass beim Start eines Java-Programmes ohne Argumente stets ein Feld der Länge 0 übergeben wird. Vergessen Sie aber bitte nicht, dass `main`, wie jede andere statische Methode auch, von anderen Methoden im Code direkt aufgerufen werden kann. Und in diesem Fall kann **`null`** durchaus ein übergebener Parameter sein. Für weitere interessante Beispiele zum Thema Annahme und Realität sei auf das Buch [4] bzw. [5] aus dem beigefügten Literaturverzeichnis verwiesen.

```
Assert.assertNotNull("Parameter " + i + " ist null",args[i]);
```

Anhand des Methodennamens `assertNotNull` erkennen wir sofort, dass wir hier auf eine „ist nicht **`null`**“-Bedingung abprüfen. Dies erhöht die Lesbarkeit des Codes beträchtlich[5].
Die Klasse `Assert` beinhaltet folgende überprüfbare Zusicherungen:

- `assertTrue` prüft, ob ein boolscher Ausdruck wahr ist.
- `assertFalse` prüft, ob ein boolscher Ausdruck unwahr ist.
- `assertEquals` überprüft zwei Objekte (oder primitive Datentypen) auf Gleichheit.
- `assertSame` überprüft, ob zwei Variablen dasselbe Objekt referenzieren (oder beide **`null`**) sind.
- `assertNotSame` überprüft, ob zwei Variablen unterschiedliche Objekte referenzieren.
- `assertNull` überprüft, ob eine Variable **`null`** referenziert.
- `assertNotNull` überprüft, ob eine Variable *nicht* **`null`** referenziert.

Jede der assert-Methoden in JUnit ist überladen, sodass sie sich mit oder ohne Fehlermeldung einsetzen lässt. Die Methode `assertEquals` ist ferner so überladen, dass sie sowohl primitive Datentypen als auch Objekte verarbeiten kann.

3.2 Unit-Tests

Nehmen wir einmal an, wir haben folgende Klasse geschrieben:

```
1  public class Datenklasse {
2
3    private String name;
4
5    public String getName() {
6      return name;
7    }
8
9    public void setName(String name) {
10     this.name = name;
11   }
12
13   public String toString() {
14     return name.toString();
15   }
16
17 }
```

Die Klasse besitzt eine Instanzvariable `name`, und get/set-Methoden für den Zugriff. Ferner haben wir eine `toString`-Methode verfasst, um ein Objekt beispielsweise auf der Konsole ausgeben zu können.

[5] Ab Java 5.0 können wir dank statischer Importe sogar das `Assert` am Anfang weglassen.

Wie können wir nun sichergehen, dass der von uns verfasste Code auch funktioniert? Traditionelles Vorgehen vieler Programmierer ist es, eine `main`-Methode in den Code einzubauen:

```
public static void main(String[] args) {
  Datenklasse d = new Datenklasse();
  Assert.assertNull(d.getName());
  d.setName("test");
  Assert.assertNotNull(d.getName());
  Assert.assertEquals("test",d.getName());
}
```

Mit anderen Worten, wir erweitern unsere Klassen um eine Testmethode. Dieses Vorgehen hat allerdings verschiedene Nachteile:

- Je komplexer eine Klasse wird, desto mehr Tests müssen durchgeführt werden. Die `main`-Methode wird äußerst lang und unübersichtlich werden.
- Die Testlogik ist in die Klasse selbst verdrahtet und somit jedermann zugänglich. Je mehr Klassen auf diese Art und Weise programmiert werden, desto mehr `main`-Methoden werden in der Anwendung herumschwirren.
- Falls die Klasse in einem größeren Kontext benutzt werden muss, z.B. in Verbindung mit einer Datenbank, kann das Vorbereiten des Tests die Klasse sogar noch weiter aufblähen.
- Das Aufrufen einer `main`-Methode ist gut und schön für einen einzelnen Test – im Rahmen eines großen Programmierprojektes ist es allerdings sehr schwer, derartige Tests zu automatisieren und auf regelmäßiger Ebene ablaufen zu lassen.

Wir wollen aus diesem Grund Abstand von Tests in einer Main-Methode nehmen und stattdessen sogenannte Unit-Tests verwenden. Unit-Tests sind Klassen, die sich von der Klasse `JUnit.framework.TestCase` ableiten. Wir betten unseren Test nun in eine solche Klasse ein:

```
1  import junit.framework.*;
2
3  public class DatenTest extends TestCase {
4
5    public DatenTest(String name) {
6      super(name);
7    }
8
9    public void testGetUndSetName() {
10     Datenklasse d = new Datenklasse();
11     Assert.assertNull(d.getName());
12     d.setName("test");
13     Assert.assertNotNull(d.getName());
14     Assert.assertEquals("test",d.getName());
15   }
16
17 }
```

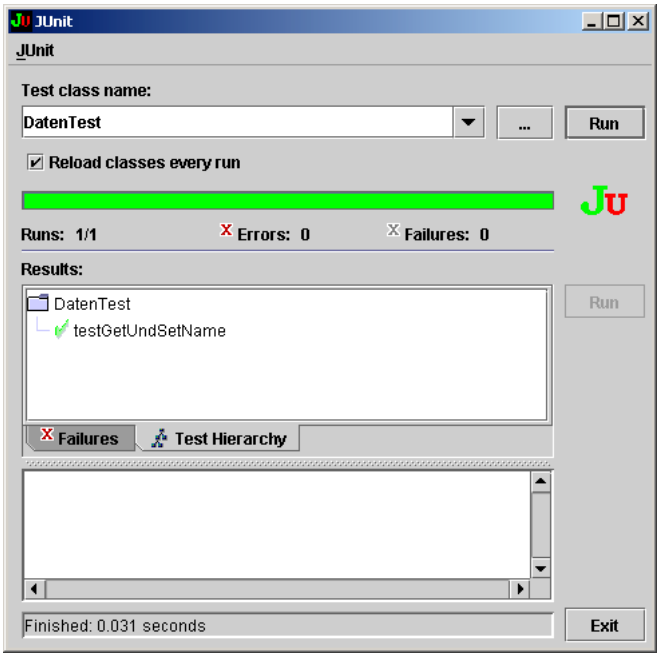


Abbildung 3.1: Der TestRunner im Einsatz

Beachten Sie, dass die Klasse `DatenTest` einen Konstruktor mit einem `String` besitzt. Dies sollte für alle Testklassen der Fall sein.
Wir stellen nun sicher, dass sich die JUnit-Bibliothek in unserem Klassenpfad befindet und rufen

Konsole

```
java JUnit.swingui.TestRunner DatenTest
```

auf. Abbildung 3.1 stellt das Ergebnis dieses Programmaufrufs dar. Wir haben den JUnit Testrunner aufgerufen, eine von mehreren Benutzeroberflächen der Bibliothek. Der Name unserer Testklasse wurde als Parameter übergeben. Nachdem die Anwendung gestartet wurde, durchsuchte diese die Klassendefinition unseres Tests nach Testmethoden (alle Methoden, die mit dem Wort „test“ starten). Anschließend wurden diese Tests automatisch durchgeführt – das grüne Licht zeigt an, dass sie erfolgreich durchlaufen wurden.
Nehmen wir nun an, wir haben eine Beschwerde von einem unserer Kollegen erhalten. Er hat die Datenklasse verwendet, und diese hat eine `NullPointerException` in der Methode `toString` produziert. Ungläubig ewrweitern wir unsere Testklasse um eine weitere Methode:

```
public void testToString() {
  Datenklasse d = new Datenklasse();
```

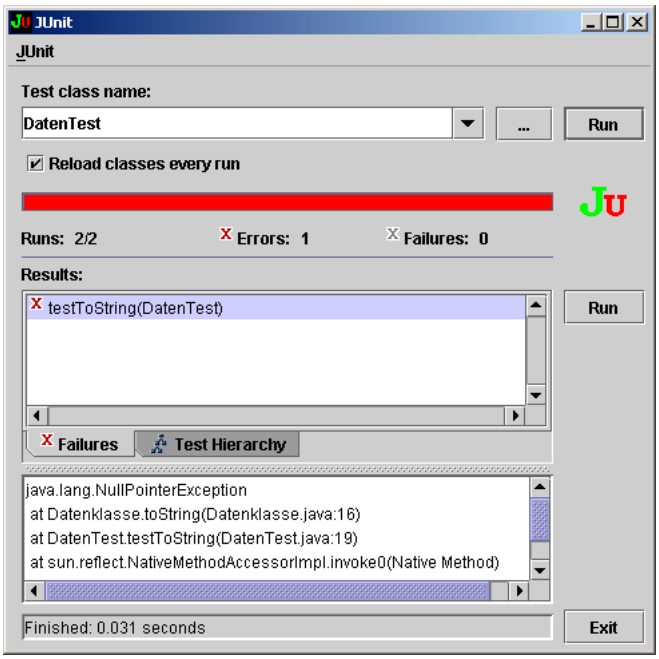


Abbildung 3.2: Der Test schlägt fehl

```
    d.toString();
    d.setName("test");
    d.toString();
  }
```

Wir rufen den Testrunner auf und tatsächlich – unsere neue Testmethode schlägt fehl (siehe Abbildung 3.2). Nun, dies sollte doch einfach zu reparieren sein: wir initialisierten einfach unser Feld mit irgendeinem Wert:

```
1  import junit.framework.Assert;
2
3  public class Datenklasse {
4
5    private String name = "???"; // NEU!!!
6
7    public String getName() {
8      return name;
9    }
10
11   public void setName(String name) {
12     this.name = name;
13   }
14
15   public String toString() {
16     return name.toString();
```

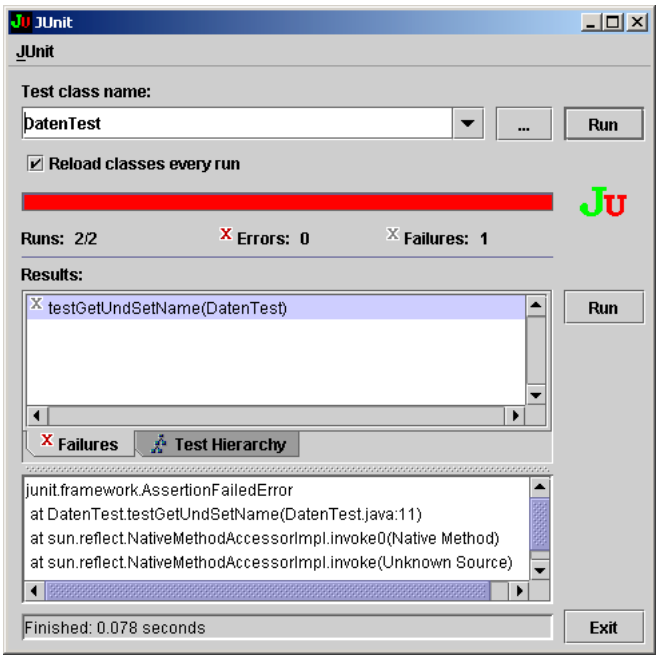


Abbildung 3.3: Ein unerwünschter Nebeneffekt

```
17      }
18
19  }
```

Kaum lassen wir den Test ein weiteres mal laufen (siehe Abbildung 3.3), trifft es uns wie ein Blitz: zwar funktioniert unsere Methode `testToString` jetzt reibungslos, aber dafür liefert die (bislang tadellose) Methode `testGetUndSet` auf einmal einen Assertion-Fehler!
Auch wenn dieses Beispiel stark vereinfacht erscheint, so macht es dennoch den Vorteil von Unit-Tests und Testbibliotheken klar: während wir versucht haben, ein aufgetretenes Problem zu beheben, haben wir einen neuen Bug in unsere Klasse eingebaut. Da wir jedoch bereits einen Test für eben diese Funktionalität hatten (nämlich die Methode `testGetUndSet`, wurden wir vom Computer auf dieses Problem automatisch aufmerksam gemacht). Wir wissen, dass unser Lösungsansatz offensichtlich sch;echt durchdacht war, und können nach einem besseren Lösungsweg suchen[6].
Wie sich denken lässt, ist das JUnit Framework noch weitaus vielschichtiger, als in diesem Beispiel gezeigt. So besitzt die Klasse `TestCase` noch zwei Methoden `setUp` und `tearDown`, um vor Beginn und Ende eines Tests gewisse Vorbereitun-

[6] Machen Sie sich dies am besten zur Übungsaufgabe. Wie muss man die Klasse verändern, damit beide Tests funktionieren?

gen zu treffen (z.B. die Verbindung zu einer Datenbank). Auch besitzt das Framework die Möglichkeit, meherere Testklassen zu einer `TestSuite` zusammenzubinden oder einfach automatisch nach allen Unit-Tests innerhalb des Klassenpfades zu suchen. Für dies und mehr sei hier auf die hervorragende Dokumentation auf `http://www.JUnit.org/` verwiesen.

3.3 Annotations und JUnit

JUnit ist ein Werkzeug, das die Erstellung und Ausführung von Tests erleichtern und automatisieren soll. Um dies zu ermöglichen, haben wir uns bislang an gewisse Konventionen halten müssen. So musste beispielsweise ein Unit Test immer in einer Methode sein, die mit dem Wort `test` beginnt. Code, der einen Test initialisieren sollte, kam in die `setUp` Methode; Code zum Herunterfahren des Tests kam in `tearDown`. Dies war ein wenig fehleranfällig: ein Tippfehler im Methodenname `trstMeinProgramm` konnte verhindern, dass ein Test ausgeführt wurde. Aus diesem Grund verwendet das neuere JUnit 4.5 Annotations anstelle von Namenskonventionen. Werfen wir einen Blick auf eine JUnit 4.5 kompatible Testklasse:

```
1  import org.junit.*;
2
3  public class DatenklasseTest {
4
5    private Datenklasse m_target;
6
7    @Before
8    public void setUpTest () {
9      m_target = new Datenklasse();
10   }
11
12   @After
13   public void tearDownTest () {
14     m_target = null;
15   }
16
17   @Test
18   public void getUndSetNameTestRight () {
19     Assert.assertNotNull(m_target.getName());
20     Assert.assertEquals("",m_target.getName());
21     m_target.setName("test");
22     Assert.assertNotNull(m_target.getName());
23     Assert.assertEquals("test",m_target.getName());
24   }
25
26   @Test
27   public void toStringTestRight () {
28     Assert.assertNotNull(m_target.toString());
29     m_target.setName("test");
30     Assert.assertEquals("test", m_target.toString());
31   }
32 }
```

Wie man sieht, können Testmethoden nun beliebig genannt werden, solange sie mit der `@Test` Annotation versehen sind. Wichtige Annotationen, die man kennen sollte, sind:

- `@BeforeClass` : Eine mit dieser Annoation markierte Methode wird bei der Instanziierung der Test-Klasse (und damit des gesamten Testfalls) ausgeführt und erlaubt die – meist aufwändige – Herstellung einer definierten Testumgebung für alle Tests des Testfalls, z.B. durch den Aufbau einer Reihe von Datenbankverbindungen.
- `@AfterClass` : Eine mit dieser Annoation markierte Methode wird kurz vor der Destruktion der Test-Klasse (und damit ganz am Ende, nachdem alle Tests durchlaufen wurden) ausgeführt und erlaubt den – meist ebenfalls aufwändige – Abbau der anfangs erzeugten, definierten Testumgebung, z.B. das Beenden der anfangs aufgebauten Datenbankverbindungen, so dass die Datenbank entlastet wird.
- `@Before` : Eine mit dieser Annoation markierte Methode wird direkt *vor* jedem Test aufgerufen und stellt so einheitliche – meist testklassen-interne – Testbedingungen sicher.
- `@After` : Eine mit dieser Annoation markierte Methode wird direkt *nach* jedem Test aufgerufen und dient meist dem Aufräumen von Hinterlassenschaften des letzten Testdurchlaufs.
- `@Test` : Eine mit dieser Annoation markierte Methode stellt einen Test dar und wird automatisch vom JUnit-Framework als Test erkannt und – gemäß oben beschriebener Aufrufreihenfolge der vor- und nachbereitenden Methoden und ihrer Position in der Test-Klasse – ausgeführt.

3.4 Best practices

Die folgenden „goldenen Regeln“ beschreiben Vorgehensweisen, die die Autoren im täglichen Umgang mit Softwareentwicklung als wertvoll empfunden haben. Wir erheben natürlich keinen Anspruch auf Vollständigkeit – es gibt verschiedene „Schulen“ der Programmierkunst (siehe beispielsweise [10]), die unterschiedliche Herangehensweisen an Unit-Tests (falls überhaupt) empfehlen. Wahrscheinlich muss ein jeder seinen eigenen Weg finden...

Regel 1: *Write a little, test a little.*

Niemand schreibt wirklich gerne Tests. Andererseits gibt es kaum etwas deprimierenderes, als tagelang vor einem Stück Code zu sitzen und den Fehler zu suchen. Oftmals schreibt man ein Programm Klasse für Klasse und diese wiederum Methode für Methode. Warum nicht für jede neue Klasse eine neue Testklasse anlegen, für jede neue Methode (oder jedes neue zusammenhängende Stück Funktionalität) gleich den zugehörigen Test schreiben? Auf diese Weise kommen Fehler

schnelllstmöglich ans Licht, und die Testbibliothek wächst quasi von allein. Last but not least gibt es kaum ein besseres Gefühl als die Gewissheit, dass der Code am Ende des Tages auch lauffähig sein wird.

Regel 2: *Better some test than no test.*

Viele Leute sind der Meinung, sie hätten keine Zeit für Unit-Tests. Das Projekt ist in Bälde fällig, und jede zusätzliche Zeile Code hält davon ab, Funktionalität zu implementieren.
Unglücklicherweise sind es oftmals diese Leute, die dann Nachtschicht über Nachtschicht lang nach Fehlern in ihren Sourcen suchen, und sie nicht finden. Die Zeit, die sie am Anfang gespart haben, zahlen sie am Ende doppelt und dreifach in der Testphase drauf.
Falls sich der Leser einmal in einer ähnlichen Situation befinden sollte, so sei bitte auf oben stehende Regel verwiesen. Es ist immer noch besser, zumindest die kritischen Teile (jener Code, von dem wir sowieso befürchten, dass sich Fehler einschleichen könnten) mit Tests zu versehen, als auf die eigene Unfehlbarkeit zu vertrauen.

Regel 3: *Test first, then fix.*

Was ist der erste Schritt, wenn beim Entwickler ein Bug-Report eintrifft? Den Fehler suchen und beheben? Falsch! Der allererste Schritt sollte es sein, das Problem zu reproduzieren - und zwar automatisch. Wenn man einen JUnit-Test dafür schreiben kann, weiß man eines mit Gewissheit: wenn der Fehler erst behoben ist, tritt er nie wieder auf!

Regel 4: *Nightly builds.*

Diese Regel klingt vielleicht ein wenig extrem, sie wird aber in vielen grossen Programmierprojekten (von Open Source bei Apache bis hin zum Softwaregiganten Microsoft) mit großem Erfolg eingesetzt.
In vielen großen Programmierprojekten mit vielen Entwicklern wird ein Zyklus für Komplettübersetzungen des Projekts durchgeführt. Dies beinhaltet

- das Kompilieren aller Klassen (gibt es irgendwelche Übersetzungsfehler),
- das Durchführen aller existierenden Unit-Tests gegen den frisch übersetzten Code und
- das Überprüfen der Fehlermeldungen und die Benachrichtigung[7] der Verantwortlichen.

[7] Dies kann in verschiedenen Formen und mit unterschiedlichen Konsequenzen stattfinden. Ein großes Softwarehaus beispielsweise hat es sich zu Eigen gemacht, den Verursacher mit der zukünftigen Auswertung der nächtlichen Builds zu beauftragen. Mit anderen Worten: Er muss jeden Morgen zu seiner zusätzlichen Arbeit die Auswertungen durchführen, bis ein anderer den allnächtlichen Test verfehlt und das Ruder übernimmt.

Im Idealfall findet dieser „Build“ jede Nacht statt, nachdem die Entwickler nach Hause gegangen sind. Der nächtliche Build ist eine hervorragende Methode, um frisch aufgetretene Unverträglichkeiten zwischen Modulen so schnell wie möglich zu erkennen. Ferner gibt es dem Team ein erhöhtes Maß an Zuversicht, wenn die nächste Deadline unaufhaltbar näher rückt.
Die Idee eines Nightly Builds mag auf den ersten Blick ein wenig radikal klingen; es handelt sich hierbei jedoch nicht einmal mehr um den Extremfall. Mehr und mehr Firmen stellen auf noch kleinere Zyklen um, damit Probleme in dem Moment erkannt werden, in dem sie in die Codebasis eingeführt werden. In diesem Modell, dem sogenannten Continuous Build, wird schon wenige Minuten nach dem Vorfall der Programmierer eine automatische Email erhalten und kann dann den Fehler sofort beheben. Für ein beliebtes Open Source System, das dies bewerkstelligt, sei auf [11] verwiesen.

3.5 Zusammenfassung

JUnit is eine Klassenbibliothek für das Verfassen und kontrollierte Durchführen automatischer Software-Tests. Auch wenn nicht Bestandteil des Standard-JDK, handelt es sich doch um eine der verbreitesten und beliebtesten Bibliotheken in der Java Software-Entwicklung. JUnit beinhaltet das Test-Framework plus verschiedene Umgebungen (sowohl Konsole als auch graphisch.[8] für die Ausführung der Tests.
Dieses Kapitel hat einen kurzen Einblick in die Möglichkeiten automatisierter Tests gegeben. Das Konzept derartiger Tests ist heutzutage weit verbreitet und sollte dem angehenden Entwickler kein Fremdbegriff sein.

[8] Die Entwicklungsumgebung Eclipse (wie auch diverse andere Umgebungen) besitzt übrigens eine hervorragende eingebaute Unterstützung für JUnit.

Ergänzung 4

Entwurfsmuster

4.1 Was sind Entwurfsmuster?

Wir werden uns in diesem Kapitel mit den so genannten Entwurfsmustern (englisch: design patterns) befassen. Es handelts sich bei Patterns quasi um das Vokabular, das der Programmierer bzw. die Programmiererin in der Entwicklungsphase verwendet, um die zu lösende Aufgabe zu strukturieren. Mit Hilfe von Entwurfsmustern entwickelt man den Design-Ansatz für ein Programm, das die gestellten Anforderungen erfüllt.
Wie soll man sich aber den konkreten Umgang mit Entwurfsmustern vorstellen? Nehmen Sie einmal an, Sie und ein weiterer Softwareentwickler müssten ein Computerspiel entwickeln. Hierbei soll es sich beispielsweise um ein Pacman-artiges[1] Spiel (oder eine vergleichbare komplexe Aufgabe) handeln. Ihr Gegenüber ist der Grafikexperte, während Sie sich mit all jenen Dingen auskennen, die man sonst in einem Spiel benötigt (etwa die Intelligenz der Monster). Wie teilen Sie die Arbeit am effizientesten auf?
An dieser Stelle erinnern Sie sich vielleicht an unsere ersten „Gehversuche" mit grafischen Oberflächen – insbesondere, falls Sie schon einen Blick in den Praxis–Teil dieser Erweiterung geworfen haben. Wenn wir beispielsweise ein Spiel wie das Game of Life (Seite 115) entwickelten, haben wir den Entwurf aufgeteilt in einen Grafikteil (die `GameEngine`) und ein dazugehöriges Modell (das `GameModel`). Hierbei gehorchte das Modell einer vorgegebenen Schnittstelle, auf die die `GameEngine` Zugriff hatte. Die `GameEngine` benötigte jedoch keinerlei Informationen darüber, welche Vorgänge konkret in dem Modell abliefen. Grafik und Spielsteuerung waren also völlig voneinander entkoppelt.
Zurück zu Pacman. Sie erinnern sich also an diese Vorgehensweise und wollen es in Ihren Projekten ebenso versuchen. Das Spiel wird aufgeteilt in ein **Modell** und eine Grafikoberfläche (der so genannte **View**), die auf das Modell zugreift. Even-

[1] Ein altbekanntes Computerspiel, bei dem eine Spielfigur durch ein Labyrinth mit Monstern gesteuert werden muss, während sie durch „Fressen" Punkte sammelt.

tuell werden Sie sogar beschließen, das Spielfeld-Modell und die Ablaufsteuerung des Spiels (Zählen der Punkte, Intelligenz der Monster) aus dem View auszulagern und in einer separaten Steuerungsklasse, dem so genannten **Controller** zu realisieren. Sie würden hierbei das Modell und den Controller programmieren, während Ihr Partner für die Entwicklung der Grafikoberfläche (des Views) zuständig ist. Die Frage ist jedoch: *Wie erklären Sie Ihrem Partner diese Idee?*
Sie können natürlich aus Ihrer Fachliteratur diesen Kurs herauskramen und ihm die entsprechenden Kapitel zu lesen geben. Auch können Sie ihm ausführlichst erklären, wie Sie sich die Aufteilung des Spiels in Grafik-Engine und Modell vorstellen – und *warum*. In beiden Fällen geht es also darum, Ihrem Partner die Idee verständlich zu machen, die hinter Ihrem Entwurfsansatz steht. Sie versuchen, ihm eine Idee für einen Lösungsansatz zu vermitteln. Warum diesem Ansatz nicht einfach einen Namen geben?
Bei Entwurfsmustern handelt es sich eben um genau diese Vorgehensweise: Erfahrene Softwareentwickler erkennen grundlegende Ideen, die sich im Laufe ihrer Arbeit in vielen Entwürfen wiedergefunden haben. Sie isolieren die dahinterliegende Idee und geben ihr einen Namen. In unserem Fall heißt dieser Name **Model-View-Controller-Pattern**.
Wenn Sie also mit Ihrem Partner kommunizieren wollen, können Sie Ihren gesamten Erfahrungsschatz in einem Satz zusammenfassen: „Ich möchte an dieser Stelle das Model-View-Controller-Pattern einsetzen." Ist Ihr Gegenüber ein erfahrener Programmierer, wird er wahrscheinlich wissen, worauf es Ihnen ankommt. Kennt er das entsprechende Muster noch nicht, können Sie es jetzt sofort anhand eines Beispieles erklären. Er wird in Zukunft wissen, was Sie mit diesem Pattern bezwecken, und es irgendwann sicher selbst in seinen Entwürfen verwenden.
In diesem Abschnitt haben Sie also zwei Dinge gelernt:

- Entwurfsmuster sind heutzutage ein beliebtes Schlagwort, das viele Softwareentwickler gerne verwenden. Wenn man sich von den hochtrabenden Namen jedoch nicht abschrecken lässt, so findet man hier ein nützliches Vokabular und eine Sammlung von Ideen vor, die die Kommunikation mit anderen Entwicklern erheblich vereinfacht. Außerdem lernt man selbst mit jedem neuen Muster etwas dazu.
- Je länger Sie ohne Muster programmieren, desto öfter werden Sie das Rad neu erfinden. Wenn man Sie später auf ein entsprechendes Muster hinweist, werden Sie oft feststellen, dass Sie das gleiche im Sinn hatten. Es kostete Sie nur erheblich mehr Arbeit!

In den folgenden Abschnitten werden wir zwei der wohl bekanntesten Entwurfsmuster behandeln, die sich auch in vielen Standardklassen von Java (etwa in der Grafikprogrammierung) widerspiegeln. Zusammen mit dem bereits bekannten Model-View-Controller Muster ist somit der Grundstock für einen reichhaltigen Erfahrungsschatz für Lösungsansätze in der Objektorientierung gelegt. Sie können dieses Repertoire an Standardlösungen jederzeit mit weiterer Fachliteratur (etwa [3] oder vergleichbare Bücher) ausbauen und vervollkommnen. Mit zu-

nehmender Erfahrung werden Sie lernen, Muster in Ihren Aufgabenstellungen zu erkennen und Ihr gesammeltes Wissen einzusetzen. Sie werden die Muster miteinander kombinieren, anpassen und gegebenenfalls erweitern.

4.2 Das Observer-Pattern

4.2.1 Zugrunde liegende Idee

Erinnern Sie sich an das zuvor erwähnte Game of Life (Seite 115)? Falls Sie diesen Teil noch nicht gelesen haben, hier eine Zusammenfassung: Das zugrunde liegende Spielmodell enthielt eine Ansammlung von Zellen, deren Zustand (lebend oder tot) durch Mausklicks auf der grafischen Oberfläche beeinflusst werden konnte. Jedes Mal, wenn ein solcher Mausklick getätigt wurde, musste unser Modell *benachrichtigt* werden, das heißt, es wartete quasi auf derartige Aktionen, um danach Berechnungen auszuführen und mit den entsprechenden Ergebnissen den Inhalt unseres Zellgewebes zu beeinflussen.
Es handelt sich bei unserem Spiel des Lebens natürlich um eine relativ einfache Situation, aber das zugrundeliegende Konzept ist generell einsetzbar (und ein sehr beliebtes und mächtiges Entwurfsmuster). Ein Objekt (das `GameModel`) wartet auf Aktionen, die von der Grafikoberfläche (der `GameEngine`) ausgelöst werden. Es *überwacht* also quasi den Zustand der Grafik und reagiert auf Veränderungen in ihr (Mausklicks).
Die Idee, dass ein Objekt den Zustand eines anderen Objektes überwacht, haben erfahrene Entwickler als das so genannte **Observer-Pattern** bezeichnet. Wir werden in diesem Abschnitt seine allgemeine Form kennen lernen – und in welchen Variationen sie sich in den verschiedensten Klassen von Java widerspiegelt.

4.2.2 Das Objektmodell

Werfen wir einen Blick auf das in Abbildung 4.1 dargestellte Grundmodell eines Observer-Entwurfsmusters.
Auf der einen Seite haben wir eine Klasse, deren Zustand es zu überwachen gilt. Wir bezeichnen diese zu überwachende Klasse als das **Observable**. Unser Observable kann von einer oder mehreren Klassen, den so genannten Observern, überwacht werden. Ein `Observer` macht sich dem System als solcher bekannt, indem er sich bei dem zu überwachenden Objekt registriert. Dies geschieht durch Aufruf der Methode `addObserver`.
Wie erfährt nun unser `Observer`, dass sich etwas an dem `Observable` geändert hat? Zu diesem Zweck verfügt jeder Observer über eine Methode namens `update`. Hat sich das `Observable` verändert, ruft es diese Methode in jedem registrierten `Observer` auf. Hierbei übergibt es sich selbst als Argument (**`this`**), damit der Observer weiß, welches überwachte Objekt sich verändert hat. Ferner übergibt es ein Objekt als Argument, aus dem der `Observer` schließen kann, *was* sich an dem Objekt geändert hat.

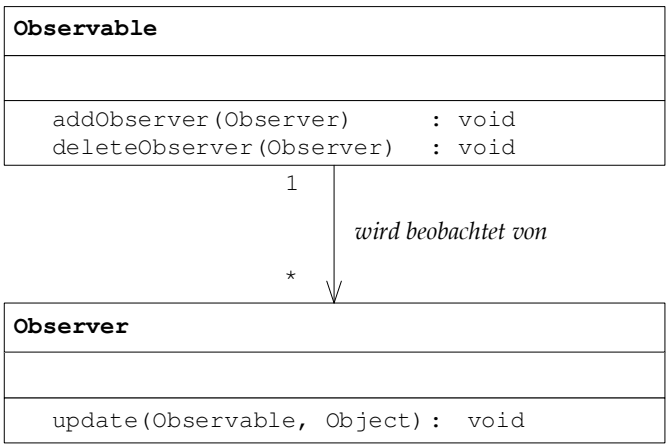


Abbildung 4.1: Das Observer-Pattern

Außerdem besitzt unser `Observable` noch eine Methode `deleteObserver`. Mit Hilfe dieser Methode kann die Registrierung eines einmal bekannt gemachten `Observers` wieder rückgängig gemacht werden. Ein `Observer` ist somit nicht auf ewig an ein bestimmtes `Observable` gebunden.

4.2.3 Beispiel-Realisierung

4.2.3.1 Das Arbeiten mit nur einem Observer

Wir beginnen mit einem einfachen Beispiel. Wir definieren eine Klasse `Name`, die einen Namen (als `String` gespeichert) repräsentiert:

```
1  /** Diese Klasse symbolisiert einen Namen */
2  public class Name {
3    /** Hier wird der Name gespeichert */
4    private String name;
5
6    /** Hiermit laesst sich der Name setzen */
7    public void setName(String name) {
8      this.name = name;
9    }
10   /** Hiermit laesst sich der Name lesen */
11   public String getName() {
12     return name;
13   }
14 }
```

Wir wollen nun eine Ansammlung von Namen (insgesamt 10 Stück) verwalten. Diese Namen sollen vom Benutzer zufällig geändert werden. Wird ein Name geändert, soll eine entsprechende Meldung auf dem Bildschirm erscheinen.

Zu diesem Zweck wollen wir das Observer-Pattern einsetzen und unsere Namensklasse um die notwendigen Methoden erweitern:

```
1  /** Diese Klasse symbolisiert einen Namen */
2  public class Name {
3    /** Hier wird der Name gespeichert */
4    private String name;
5
6    /** Hier speichern wir unseren Observer ab */
7    private Observer observer;
8
9    /** Fuegt einen Observer hinzu */
10   public void addObserver(Observer observer) {
11     this.observer = observer;
12   }
13   /** Loescht einen Observer */
14   public void deleteObserver(Observer observer) {
15     this.observer = null;
16   }
17   /** Hiermit laesst sich der Name setzen */
18   public void setName(String name) {
19     this.name = name;
20     if (observer != null)
21       observer.update(this,name);
22   }
23   /** Hiermit laesst sich der Name lesen */
24   public String getName() {
25     return name;
26   }
27 }
```

Unsere Namensklasse nimmt die Rolle des Observables aus unserem Muster ein. Der Einfachheit halber verwenden wir nur einen Observer, den wir in einer privaten Instanzvariablen abspeichern (Zeile 7). Mit Hilfe der Methoden `addObserver` und `deleteObserver` lässt sich die Instanzvariable modifizieren.
Nun müssen wir unseren `Observer` natürlich auch benachrichtigen, wenn sich unser Namensobjekt geändert hat. Wir modifizieren die set-Methode `setName` unseres Namensattributes deshalb so, dass sie die `update`-Methode des Observers aufruft (Zeile 20 und 21):

```
if (observer != null)
  observer.update(this,name);
```

Kommen wir nun zur Definition unseres Observers. In einem Feld von Namensobjekten speichern wir die verwalteten Objekte ab. Im Konstruktor erzeugen wir dieses Feld und hängen unseren Observer an jedes der zu überwachenden Namensobjekte:

```
/** Speichert zehn Namen achtet auf Aenderungen */
public class Observer {

  /** Die zehn gespeicherten Namen */
  private Name[] namen;
```

```
/** Konstruktor */
public Observer() {
  namen = new Name[10];
  for (int i = 0; i < namen.length; i++) {
    namen[i] = new Name();
    namen[i].addObserver(this);
  }
}
```

Um nun auf die Veränderung eines Namens reagieren zu können, müssen wir lediglich noch die Update-Methode ausformulieren. In dieser Methode geben wir eine wie auch immer geartete Meldung auf dem Bildschirm aus:

```
/** Achte auf die Veraenderung eines der Namen */
public void update(Name n, Object o) {
  // Finde den Index, der zu dem Namen passt
  int index = 0;
  while (n != namen[index])
    index++;
  // Gib die Aenderung auf dem Bildschirm aus
  System.out.println("Name Nr. " + index +
                     " wurde geaendert.");
  System.out.println("Neuer Name: " + o);
  System.out.println();
}
```

Jetzt formulieren wir noch eine get-Methode, mit der wir unser Namensfeld auslesen können:

```
/** Gib einen der zehn Namen zurueck */
public Name getName(int i) {
  return namen[i];
}
```

Unsere Observer-Klasse ist nun komplett. Wir fügen noch ein Hauptprogramm hinzu, das ein `Observer`-Objekt erzeugt und einzelne Namensobjekte modifiziert:

```
1  import Prog1Tools.IOTools;
2  /** Aendert die im Observer gespeicherten Namen zufaellig */
3  public class Hauptprogramm {
4    /** Die main-Routine */
5    public static void main(String[] args) {
6      // Initialisiere den Observer
7      Observer obs = new Observer();
8      // Fuehre fuenf Namensaenderungen durch
9      for (int i = 0; i < 5; i++) {
10       int index = (int)(Math.random() * 10);
11       obs.getName(index).setName(IOTools.readLine("Name:"));
12     }
13   }
14 }
```

Starten wir nun unser Programm. Wir erhalten eine Ausgabe, die mit der folgenden vergleichbar ist:

Konsole

```
Name:Lieschen Mueller
Name Nr. 4 wurde geaendert.
Neuer Name: Lieschen Mueller

Name:Kalle Karlsson
Name Nr. 8 wurde geaendert.
Neuer Name: Kalle Karlsson

Name:Mark Mustermann
Name Nr. 5 wurde geaendert.
Neuer Name: Mark Mustermann

Name:Jan Jannick
Name Nr. 0 wurde geaendert.
Neuer Name: Jan Jannick

Name:Wolf Wolfram
Name Nr. 7 wurde geaendert.
Neuer Name: Wolf Wolfram
```

Unsere Observer-Klasse hat also automatisch alle Änderungen registriert. Wir haben uns um diesen Vorgang nicht mehr ausdrücklich kümmern müssen.

4.2.3.2 Das Arbeiten mit mehreren Observern

Wir wollen unser Programm nun um eine kleine Statistik erweitern. Gegen Ende des Programms soll eine Nachricht auf dem Bildschirm erscheinen, die angibt, welches Namensobjekt wie oft geändert wurde. Zu diesem Zweck wollen wir einfach einen zweiten Observer an unser Namensobjekt anhängen.
Natürlich ist unser bisheriges Observable (die Klasse `Name`) noch nicht in der Lage, mehr als einen Observer zu verarbeiten. Glücklicherweise ist dieses Problem jedoch schnell behoben, denn die Entwickler von Java haben bereits eine Möglichkeit vorgesehen, mit der man sich schnell und einfach eine derartige Observer-Verwaltung aufbauen kann:

- Die Klasse `java.util.Observable` stellt die Verwaltung beliebig vieler Observer, die über die Methode `addObserver` registriert werden können, zur Verfügung. Klassen, die diesen Mechanismus nutzen wollen, müssen lediglich das Interface `java.util.Observer` implementieren. Das Interface beinhaltet die bereits bekannte `update`-Methode.
- Um die `update`-Methode aller registrierten Observer automatisch aufzurufen, verfügt das Observable über eine Methode namens `notifyObservers`. Dieser Methode wird das Objektargument übergeben. Die Benachrichtigung der Observer erfolgt dann automatisch. Zuvor muss der Programmierer das `Observable` jedoch mit Hilfe der Methode `setChanged` als „verändert“

markieren. Diese Methode ist allerdings `protected`, das heißt, sie kann nur von Subklassen der Klasse `Observable` aufgerufen werden.

Wir wollen nun versuchen, unsere Klasse `Name` mit Hilfe dieses vordefinierten Mechanismus zu verbessern. Zu diesem Zweck leiten wir sie von der Klasse `Observable` ab:

```
1  import java.util.Observable;
2
3  /** Diese Klasse symbolisiert einen Namen */
4  public class Name extends Observable {
5
6    /** Hier wird der Name gespeichert */
7    private String name;
8
9    /** Hiermit laesst sich der Name setzen */
10   public void setName(String name) {
11     this.name = name;
12     setChanged();
13     notifyObservers(name);
14   }
15   /** Hiermit laesst sich der Name lesen */
16   public String getName() {
17     return name;
18   }
19 }
```

Wie wir sehen, hat sich unsere Klasse durch die Verwendung der Klasse `java.lang.Observable` deutlich verschlankt. Die Verwaltung der verschiedenen `Observer` wird uns von `Observable` abgenommen; wir erben die Methoden `addObserver` und `deleteObserver` also von der Superklasse. Von unseren eigenen Methoden haben wir lediglich die set-Methode unseres Attributes `name` verändert:

```
/** Hiermit laesst sich der Name setzen */
public void setName(String name) {
  this.name = name;
  setChanged();
  notifyObservers(name);
}
```

Anstatt die Methode `update` der `Observer` wie bisher explizit aufzurufen, verwenden wir die vorgegebenen Methoden `setChanged` und `notifyObservers`. Der Rest geht, wie gesagt, automatisch vonstatten, wir müssen uns um die Verwaltung der Observer nicht weiter sorgen.
Wie müssen wir nun unsere bisherige Klasse `Observer` anpassen? Da die Klasse `java.util.Observable` nur mit Objekten funktioniert, die das Interface `java.util.Observer` implementieren, müssen wir unser Objekt an den folgenden Stellen modifizieren:

1. In der Definition unserer Klasse müssen wir verdeutlichen, dass wir das Interface `java.util.Observer` implementieren, die Klasse also ein `Observer` im Sinne des Entwurfsmusters ist:

```
public class Observer implements java.util.Observer {
```

Achten Sie hierbei darauf, dass sowohl die Klasse als auch das Interface den Namen `Observer` besitzen. Dabei haben wir kein Problem, die beiden zu unterscheiden: wir verwenden einfach den Paketnamen unseres Interface (`java.util`), um Eindeutigkeit herzustellen.

2. Die einzige Methode, die wir für das Interface realisieren müssen, ist die Methode `update`. Diese unterscheidet sich von unserer alten Methode nur in einem Punkt: unser `Observable` ist jetzt nicht mehr die spezielle Klasse `Name`, sondern deren allgemeine Superklasse `java.util.Observable`:

   ```
   public void update(java.util.Observable n,Object o) {
   ```

   Der Rest unserer Methode bleibt vollkommen unverändert.

Kommen wir nun aber zu unserer neuen Klasse: dem `StatisticObserver`:

```
/** Dieser Observer fuehrt eine kleine Statistik auf den Namen aus */
public class StatisticObserver implements java.util.Observer {
```

Da unsere Klasse als `Observer` für unsere Namensobjekte registriert werden soll, muss sie sich von dem allgemeinen Interface `java.util.Observer` ableiten – das heißt, wir werden später eine entsprechende `update`-Methode formulieren müssen.
Wir wollen uns aber zuerst um die interne Datenstruktur kümmern. Da wir auch in diesem Objekt mit unseren zehn Namen arbeiten, wollen wir diese ebenfalls in einem Feld abspeichern. Ferner definieren wir ein Feld von ganzzahligen Zählern, in denen wir speichern, wie oft ein spezielles Namensobjekt geändert wurde:

```
/** Dieses Feld enthaelt die Namensobjekte */
private Name[] namen;

/** Dieses Feld zaehlt, wie oft ein Objekt geaendert wurde. */
private int[] zaehler;
```

Wir benutzen einen Konstruktor, um die Felder zu initialisieren. Hierzu lesen wir die Namensobjekte aus einem vorgegebenen `Observer`-Objekt aus und speichern sie in unserem Feld `namen`. Anschließend registrieren wir den `StatisticObserver` bei dem entsprechenden Namensobjekt. Zu guter Letzt erzeugen wir unser Zählerfeld und füllen es komplett mit 0 auf:

```
/** Konstruktor. Verwendet unsere Observer-Klasse, um aus
    ihr die Namensobjekte zu erhalten. */
public StatisticObserver(Observer obs) {
  // Initialisiere das Namens-Feld
  namen = new Name[10];
  for (int i = 0; i < 10; i++) {
    namen[i] = obs.getName(i);
    namen[i].addObserver(this);
  }
  // Initialisiere das Zaehler-Feld
  zaehler = new int[10];
```

```
    java.util.Arrays.fill(zaehler,0);
  }
```

Kommen wir nun zu unserer Methode `update`. Diese unterscheidet sich nur geringfügig von der bereits definierten Methode in unserer anderen `Observer`-Klasse. Zuerst ermitteln wir den Index des Objektes, das sich geändert hat. Anstatt nun jedoch eine Nachricht auf dem Bildschirm auszugeben, erhöhen wir lediglich den entsprechenden Zähler:

```
  /** Wenn ein Name geaendert wird, wird diese Methode aufgerufen */
  public void update(java.util.Observable n,Object o) {
    // Finde den Index, der zu dem Namen passt
    int index = 0;
    while (n != namen[index])
      index++;
    // Erhoehe den Zaehler an der entsprechenden Stelle
    zaehler[index]++;
  }
```

Anschließend definieren wir noch eine Methode namens `getStatistic`. Diese wertet unser Feld von Zählern aus und gibt die Auswertung in Form einer textuellen Beschreibung (eines `String`) zurück:

```
  /** Erzeuge eine Statistik-Meldung aus den Zaehler-Daten */
  public String getStatistic() {
    String res = "";
    for (int i = 0; i < 10; i++)
      res += "Name Nr. " + i + " wurde " + zaehler[i] +
             "-mal geaendert.\n";
    return res;
  }
```

Unsere Klasse `StatisticObserver` ist somit komplett. Kommen wir nun zu unserem eigentlichen `Hauptprogramm`:

```
1  import Prog1Tools.IOTools;
2  /** Aendert die im Observer gespeicherten Namen zufaellig */
3  public class Hauptprogramm {
4    /** Die main-Routine */
5    public static void main(String[] args) {
6      // Initialisiere den Observer
7      Observer obs = new Observer();
8      // Initialisiere die Statistik
9      StatisticObserver statistic = new StatisticObserver(obs);
10     // Fuehre fuenf Namensaenderungen durch
11     for (int i = 0; i < 5; i++) {
12       int index = (int)(Math.random() * 10);
13       obs.getName(index).setName(IOTools.readLine("Name:"));
14     }
15     // Gib die Statistik aus
16     System.out.println("Zum Schluss noch die Statistik:");
17     System.out.println(statistic.getStatistic());
18   }
19 }
```

Wie Sie sehen, haben wir lediglich zwei winzige Modifikationen vorgenommen:

1. Zu Beginn unserer `main`-Methode haben wir eine Instanz unserer neuen Klasse `StatisticObserver` erzeugt (Zeile 9) und
2. gegen Ende das Ergebnis dieser Statistik auf dem Bildschirm ausgegeben (Zeile 16 und 17).

Weitere Änderungen, etwa bei der Belegung der Namensfelder, waren nicht notwendig. *Unsere Statistik hat sich sämtliche Daten, die sie benötigt, selbstständig aus den Namensobjekten geholt!*
Spätestens mit diesem Hauptprogramm sollte der Nutzen des Observer-Patterns langsam, aber sicher deutlich werden. Wie in einem Baukastensystem können wir mit wenig Aufwand eine komplette Statistik an unsere gegebenen Namensobjekte hängen. Dabei müssen wir an keiner Stelle in unserem Hauptprogramm besondere Rücksicht darauf nehmen, ob wir nun *mit* oder *ohne* Statistik arbeiten. Wir können weiterhin unsere Namenswerte mit `setName` setzen und darauf vertrauen, dass unsere Statistik automatisch benachrichtigt wird.
Der Nutzen dieses Baukastensystems wird noch deutlicher, wenn man bedenkt, dass wir uns nicht auf diese beiden Observer beschränken müssen. Vielleicht möchte ein anderer Programmierer zusätzlich sämtliche Namensänderungen protokollieren und eine Historie der Namensentwicklungen einer bestimmten Person erstellen. Eine Programmiererin möchte vielleicht dafür sorgen, dass bei einer bestimmten Namensänderung ein „Alarm“ ausgelöst wird (Beispiel: eine prominente Frau nimmt wieder ihren Mädchennamen an – soll die Boulevardpresse verständigt werden?). All dies kann völlig problemlos und unabhängig von den anderen Komponenten geschehen, indem diese Entwickler ihre eigenen Observer definieren.
Nach so viel „Philosophie“ wollen wir aber wieder zur Praxis kommen und am Beispiel zeigen, dass unsere Änderung auch tatsächlich funktioniert. Es folgt eine mögliche Bildschirmausgabe bei der Ausführung unseres Programms:

Konsole

```
Name:Lieschen Mueller
Name Nr. 9 wurde geaendert.
Neuer Name: Lieschen Mueller

Name:Kalle Karlsson
Name Nr. 9 wurde geaendert.
Neuer Name: Kalle Karlsson

Name:Mark Mustermann
Name Nr. 6 wurde geaendert.
Neuer Name: Mark Mustermann

Name:Jan Jannick
Name Nr. 4 wurde geaendert.
Neuer Name: Jan Jannick
```

```
Name:Wolf Wolfram
Name Nr. 9 wurde geaendert.
Neuer Name: Wolf Wolfram

Zum Schluss noch die Statistik:
Name Nr. 0 wurde 0-mal geaendert.
Name Nr. 1 wurde 0-mal geaendert.
Name Nr. 2 wurde 0-mal geaendert.
Name Nr. 3 wurde 0-mal geaendert.
Name Nr. 4 wurde 1-mal geaendert.
Name Nr. 5 wurde 0-mal geaendert.
Name Nr. 6 wurde 1-mal geaendert.
Name Nr. 7 wurde 0-mal geaendert.
Name Nr. 8 wurde 0-mal geaendert.
Name Nr. 9 wurde 3-mal geaendert.
```

4.2.4 Variationen des Pattern

Wie bereits in der Einleitung erwähnt, handelt es sich bei Entwurfsmustern lediglich um eine bestimmte Idee. Daher wundern wir uns nicht darüber, dass es bei der Vielzahl von Programmierern auf dieser Welt viele unterschiedliche Ansätze gibt, das Pattern zu realisieren.
Die wohl einfachste Modifikation ist eine Veränderung der Namen. Nicht jeder lernt das Konzept des Observers unter diesem Namen kennen. Einige Programmierer lernen Java beispielsweise über dessen Grafikklassen kennen.[2] Sie werden dann den Observer vielleicht unter einem anderen Namen, etwa **Listener**, kennen. Andere kennen zwar das Wort `Observer`, nennen aber die `update`-Methode vielleicht `notify` oder `objectHasChanged`. Derartige Abwandlungen sind im Allgemeinen leicht zu verstehen.
Eine andere oft auftretende Modifikation ist es, die Argumente der `update`-Methode anders zu formulieren. Erinnern Sie sich beispielsweise an unser `GameModel`. Die Methode `buttonPressed` besaß statt des einen Objektarguments zwei ganzzahlige Parameter vom Typ **`int`**, die die Zeile und Spalte des Knopfes angaben, der gedrückt wurde. Der entsprechende Observer wurde gar nicht erst übergeben, da unser Modell nur an eine einzige `GameEngine` gebunden war. Unsere Methode `firePressed` verfügte sogar über gar keine Argumente – es ist klar, dass an dieser Stelle nur der Feuerknopf betätigt werden konnte.
Andere Entwickler übergeben statt des allgemeinen Objektarguments eine spezielle Klasse, die sie zu diesem Zweck entworfen haben. So wird etwa in der Gra-

[2] Es ist der Ansatz so manches Einsteigerbuchs, die Leserinnen und Leser zunächst mit einem „nützlichen“ Beispiel zu konfrontieren. Sie finden dann auf den ersten Seiten oft ein komplettes Programm mit grafischer Oberfläche und so viel Neuem und Unbekanntem, dass der Einsteiger gut daran tut, zunächst einmal gehörig beeindruckt zu sein. Anschließend verwenden die Autoren das halbe Buch (oder sogar mehr) darauf, den Lesern dieses eine gewaltige Programm zu erklären. Gegen Ende haben Sie dann oft einen Entwickler, der hervorragend mit Fenstern und Feuerknöpfen umgehen kann. Oft reicht sein Grundlagenwissen aber über die **`for`**-Schleife nicht hinaus.

fikprogrammierung von Java mit so genannten **Events** gearbeitet. Diese Objekte modellieren ganz spezielle Ereignisse in der Grafik, etwa das Drücken eines Mausknopfes oder eine Tastatureingabe. Durch die Verwendung dieser speziellen Objekte sind die Programmierer in der Lage, den Informationsfluss aus der Grafik heraus besser zu strukturieren.[3] Wer von einem speziellen Objekt Informationen erhalten will, registriert sich dort ganz einfach als `EventListener` (wie gesagt, die Namensgebung ist hier ein wenig anders).
Was Ihnen diese Beispiele aus der Arbeit mit Java zeigen sollen, ist die Notwendigkeit, flexibel zu sein. Ein Entwurfsmuster ist kein starres Konstrukt, das Sie auf Gedeih und Verderb so anwenden müssen, wie Sie es in der Fachliteratur vorfinden. Vergessen Sie nicht, dass es sich nur um eine Idee handelt – um einen kleinen Denkanstoß.

4.2.5 Zusammenfassung

Das **Observer-Pattern** beschreibt die Idee, bestimmte Objekte auf Veränderungen ihres Zustandes zu überwachen. Diese überwachten Objekte, die so genannten **Observables**, informieren alle ihnen bekannt gemachten **Observer** hinsichtlich der Veränderungen. Was die verschiedenen Observer mit diesen Änderungen anfangen, ist jedoch einzig und allein ihre eigene Aufgabe.
Das Observer-Pattern entkoppelt die Veränderung der Daten und die daraus resultierenden Aktionen. Ein Entwickler, der etwa Daten in einem Objekt setzt, muss sich nicht um die Aktualisierung aller mit diesen Daten verbundenen Objekte kümmern. Er geht davon aus, dass dies durch die angehängten Observer automatisch geschieht.
Das Observer-Pattern ist eines der in Java wohl am häufigsten verwendeten Entwurfsmuster. Die komplette Behandlung von Benutzereingaben (etwa Mausklicks und Tastatureingaben) in einer grafischen Oberfläche wird über diesen Mechanismus gesteuert. Hierbei wurde das Muster zwar leicht modifiziert (die Observer heißen Listener; es werden spezielle `Event`-Objekte übergeben), doch das Prinzip entspricht unserem einfachen Beispiel mit der Namensklasse und der aufgeschalteten Statistik.

4.2.6 Übungsaufgaben

Aufgabe 4.1

Stellen Sie sich vor, die Namensklasse wird in einer Behörde eingesetzt. Um groben Unfug zu vermeiden, existiert eine Liste von Namen (Mistkerl, Depp, Schnarchnase, ...), die nicht vergeben werden dürfen. Schreiben Sie einen speziellen Observer, der die Namensobjekte auf diese Texte hin überwacht. Entspricht

[3] In den ersten Tagen von Java haben die Entwickler noch ohne das Observer-Pattern gearbeitet. Konnte in einem grafischen Objekt beispielsweise eine Maus gedrückt werden, so verwendeten sie eine Methode namens `mouseClicked`. Dieser Ansatz, für jedes auftretende Ereignis eine eigene Methode zu definieren, erwies sich jedoch mit der Zeit als viel zu unflexibel.

ein eingegebener Name einem dieser Worte (verwenden Sie die `equals`-Methode zur Überprüfung), so setzen Sie das Objekt auf den alten Namen zurück.

4.3 Das Composite-Pattern

4.3.1 Zugrunde liegende Idee

Das Composite-Pattern ist auf den ersten Blick nur schwer zu verstehen. Wir wollen deshalb mit einem konkreten Beispiel beginnen, bevor wir auf das eigentliche Objektmodell eingehen.
Früher oder später haben wir uns in der Schule alle mit den so genannten Kurvendiskussionen befassen müssen. Dabei war eine Funktion f gegeben, etwa

$$f(x) = 3 \cdot x^2 + 7 \cdot x + 5$$

und wir mussten zu dieser Funktion Dinge wie Nullstellen, Hoch- und Tiefpunkte oder die Wendepunkte bestimmen. Zu diesem Zweck ermittelten wir die Ableitung f' der Funktion, in diesem Fall also

$$f'(x) = 6 \cdot x + 7$$

Diese Ableitung wird dann auf gewisse Kriterien (etwa Nullstellen oder Vorzeichenwechsel) hin untersucht.
Wir wollen in Java nun ein Programm schreiben, das die Ableitung der obigen Funktion automatisch berechnet und diese (etwa an der Stelle $x = 2$) auswertet. Zu diesem Zweck muss unser Programm in der Lage sein, eine Funktion automatisch abzuleiten. Aber wie um Himmels willen sollen wir das bewerkstelligen?
Wie so oft wollen wir uns hierbei nicht von komplizierten Anforderungen beeindrucken lassen, sondern arbeiten systematisch und fangen zunächst klein an. Wir wissen, dass unsere Funktion in der Lage sein soll, an einer gewissen Stelle ausgewertet zu werden – wir benötigen also eine Methode `getFunktionswert`, die den Wert von $f(x)$ berechnet. Ferner wollen wir in der Lage sein, die Ableitung zu erhalten. Bei der Ableitung handelt es sich wieder um eine Funktion, das heißt, eine entsprechende Methode `getAbleitung` müsste aus einem wie auch immer gearteten Funktionsobjekt wieder eine Funktion erstellen können. Mit Hilfe dieses Wissens können wir eine allgemeine Schnittstelle in Form einer abstrakten Klasse `Funktion` definieren:

```
1 /** Diese Klasse repraesentiert eine Funktion */
2 public abstract class Funktion {
3
4   /** Werte die Funktion an einer gewissen Stelle aus */
5   public abstract double getFunktionswert(double x);
6
7   /** Berechne die Ableitung der Funktion */
8   public abstract Funktion getAbleitung();
9 }
```

Kommen wir nun zu der wohl einfachsten aller Funktionen – wir beginnen mit den konstanten Funktionen

$$f_C(x) = C$$

für eine beliebig gewählte reelle Zahl C. Wie Sie aus der Schule wahrscheinlich noch wissen, ist die Ableitung einer Konstanten immer gleich Null, also

$$f'_C(x) = 0$$

und somit wieder konstant. Wir können also eine erste Funktionsklasse definieren, indem wir von unserer Klasse `Funktion` wie folgt ableiten:

```
1  /** Die konstante Funktion */
2  public class Konstante extends Funktion {
3
4    /** Diese Konstante wird immer zurueckgegeben */
5    private double konstante;
6
7    /** Konstruktor. Setzt die Konstante auf einen bestimmten Wert */
8    public Konstante(double konstante) {
9      this.konstante = konstante;
10   }
11   /** Werte die Funktion an einer gewissen Stelle aus */
12   public double getFunktionswert(double x) {
13     return konstante;
14   }
15   /** Berechne die Ableitung der Funktion (ist immer =0) */
16   public Funktion getAbleitung() {
17     return new Konstante(0);
18   }
19 }
```

Unsere Methoden `getFunktionsWert` und `getAbleitung` sind in diesem Fall denkbar einfach. Es ist egal, wie der Wert von `x` lautet; unsere Methode `getFunktionswert` wird immer den Wert zurückliefern, der in `konstante` gespeichert ist. Da die Ableitung einer konstanten Funktion an jeder Stelle (konstant) Null ist, wird die Ableitung unserer konstanten Funktion wieder eine konstante Funktion sein – nämlich die Konstante Null.
Wir wollen den Schwierigkeitsgrad nun etwas erhöhen und betrachten die Funktion

$$f_{id}(x) = x$$

Diese Funktion, auch als die identische Funktion bezeichnet, liefert als Ergebnis der Funktionsauswertung an der Stelle x immer wieder das originale x zurück. Die Ableitung der Funktion ist konstant 1, also

$$f'_{id}(x) = 1$$

Wir haben also eine Funktion, die als Wert immer x und als Ableitung immer 1 zurückliefert. Die entsprechende Realisierung in Java kostet uns nur wenige Zeilen:

```
1  /** Diese Klasse repraesentiert die identische Funktion */
2  public class Id extends Funktion {
3
4    /** Werte die Funktion an einer gewissen Stelle aus */
5    public double getFunktionswert(double x) {
6      return x;
7    }
8    /** Berechne die Ableitung der Funktion (ist immer =1) */
9    public Funktion getAbleitung() {
10     return new Konstante(1);
11   }
12 }
```

Wie soll man aber bei einer so komplizierten Funktion wie

$$f(x) = 3 \cdot x^2 + 7 \cdot x + 5$$

vorgehen? Soll man aus dieser Funktion ebenfalls eine eigene Klasse machen? Wenn ja, welchen Zweck hat dann die Programmierung mit Java? Sobald wir eine andere Funktion wie etwa

$$g(x) = 3 \cdot x^2 + 7 \cdot x + 25$$

realisieren müssen, müssen wir ja doch wieder eine neue Funktionsklasse schreiben und die Ableitung von Hand berechnen!
Natürlich hätten wir das Beispiel nicht gewählt, wenn dem wirklich so wäre! Obige Funktion (und jede andere entsprechende Funktion) f lässt sich nämlich in eine Anzahl von Grundfunktionen auseinander nehmen:

$$\begin{array}{lcll}
f_1(x) & & & = 3 \\
f_2(x) & & & = 5 \\
f_3(x) & & & = 7 \\
f_4(x) & & & = x \\
f_5(x) & = & x^2 & \\
 & = & x \cdot x & = f_4(x) \cdot f_4(x) \\
f_6(x) & = & 7 \cdot x & = f_3(x) \cdot f_4(x) \\
f_7(x) & = & 7 \cdot x + 5 & = f_6(x) + f_2(x) \\
f_8(x) & = & 3 \cdot x^2 & = f_1(x) \cdot f_5(x) \\
f_9(x) & = & 3 \cdot x^2 + 7 \cdot x + 5 & = f_8(x) + f_7(x)
\end{array}$$

Bei diesen Formeln handelt es sich keineswegs um Hexenwerk. Wenn wir den Wert einer Funktion mit Hilfe eines Taschenrechners, im Kopf oder auf einem Blatt Papier berechnen, dann nehmen wir diese mit Hilfe einiger weniger Regeln wie „Punkt- vor Strichrechnung“ auseinander. Wir berechnen die Ergebnisse vieler kleiner Bruchstücke (eine Teilsumme hier, ein Zwischenergebnis dort) und setzen diese **Komponenten** nach und nach zum Gesamtergebnis zusammen. Um eben dieses Zusammensetzen (englisch: **composite**) geht es im nachfolgenden Entwurfsmuster.

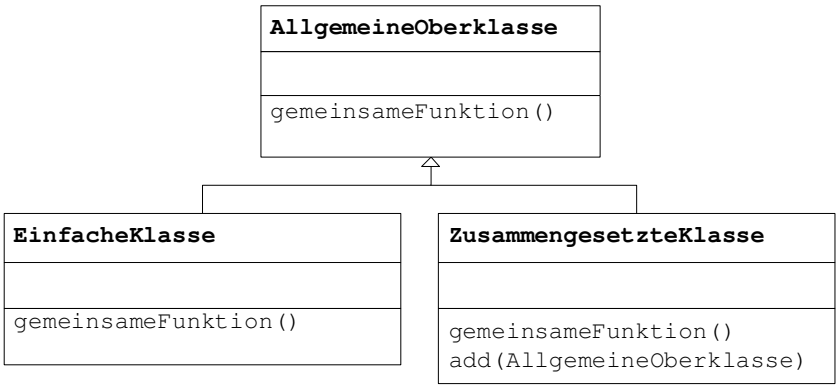


Abbildung 4.2: Das Composite-Pattern

4.3.2 Das Objektmodell

Da wir im Gegensatz zum Observer für das **Composite-Pattern** keine Standard-Implementierung in Java besitzen, werden wir uns in diesem Abschnitt mit eingedeutschten Begriffen begnügen.
Unsere Abbildung 4.2 zeigt das grundlegende Objektmodell, auf dem die Idee des Entwurfsmusters beruht. Wir beginnen mit einer abstrakten Klasse `AllgemeineOberklasse`. In dieser Klasse definieren wir Methoden, die für all unsere entworfenen Objekte gelten sollen. In unserem Anwendungsbeispiel wäre die Klasse also mit der Klasse `Funktion` gleichzusetzen. Bei den gemeinsamen Methoden handelt es sich entsprechend um die Methoden `getFunktionswert` und `getAbleitung`.
Im Anschluss starten wir damit, Subklassen unserer allgemeinen Oberklasse zu bilden. Wir beginnen mit einfachen, grundlegenden Klassen. Eine `EinfacheKlasse` stellt eine Implementierung der vorgegebenen Schnittstelle dar. In unserem Beispiel haben wir zwei derartige Klassen gebildet: die Klassen `Konstante` (konstante Funktion f_C) und `Id` (identische Funktion f_{id}).
Mit Hilfe dieser einfachen Klassen wollen wir nun beliebig komplexe Gebilde erschaffen. Zu diesem Zweck müssen wir in der Lage sein, die Klassen zusammenzusetzen. Eine (oder mehrere) `ZusammengesetzteKlasse`n erfüllen genau diesen Zweck. Diese Klassen sind dazu da, andere `AllgemeineKlasse`n aneinander zu fügen (sowohl einfache als auch zusammengesetzte Klassen). Hierzu wird oft eine Methode `add` formuliert, mit der man die verschiedenen Klassen aneinander reiht. Man kann sich aber auch andere Möglichkeiten überlegen (etwa im Konstruktor).
Bislang haben wir in unserem Anwendungsfall keine Beispiele für eine zusammengesetzte Klasse formuliert. Wir werden dies im folgenden Abschnitt nachholen.

4.3.3 Beispiel-Realisierung

4.3.3.1 Summe zweier Funktionen

Kommen wir also nun zu den zusammengesetzten Funktionen. Wir beginnen damit, eine Klasse `Summe` zu entwerfen, die zwei beliebige Funktionen $a(x)$ und $b(x)$ in der Form

$$s(x) = a(x) + b(x)$$

addiert.
Um die Addition zu bewerkstelligen, speichern wir die Funktionen $a(x)$ und $b(x)$ in gleichnamigen Instanzvariablen:

```
/** Diese Klasse repraesentiert eine Summe */
public class Summe extends Funktion {

  /** Der erste Summand */
  private Funktion a;

  /** Der zweite Summand */
  private Funktion b;

  /** Konstruktor */
  public Summe(Funktion a, Funktion b) {
    this.a = a;
    this.b = b;
  }
```

Wenn wir nun die Summe an einer Stelle x auswerten wollen, müssen wir lediglich das Resultat der beiden Teilfunktionen addieren:

```
  /** Werte die Funktion an einer gewissen Stelle aus */
  public double getFunktionswert(double x) {
    return a.getFunktionswert(x) + b.getFunktionswert(x);
  }
```

Das war doch eigentlich gar nicht so schwer. Wie sieht es aber mit der Ableitung aus?
Die Ableitung einer Summe berechnet sich nach der folgenden einfachen Regel:

$$s'(x) = a'(x) + b'(x)$$

Um also die Ableitung unserer Summe als Funktion darzustellen, müssen wir lediglich unsere beiden Teilfunktionen ableiten und diese in einer Summe zusammenfassen:

```
  /** Berechne die Ableitung der Funktion */
  public Funktion getAbleitung() {
    return new Summe(a.getAbleitung(), b.getAbleitung());
  }
```

Hierbei ist es vollkommen egal, wie kompliziert unsere Funktionen a und b sein mögen – die Methode `getAbleitung` funktioniert in jedem Fall. Wir können beispielsweise die Funktion

$$s_1(x) = x + 1$$

erzeugen, indem wir sie aus unseren Grundfunktionen

$$s_1(x) = f_{id}(x) + f_1(x)$$

zusammensetzen:

```
Funktion s1 = new Summe(new Id(),new Konstante(1));
```

Ebenso kann eine zusammengesetzte Funktion auch aus anderen zusammengesetzten Funktionen bestehen, wie etwa das Beispiel

$$s_2(x) = x + s_1(x)$$

zeigt:

```
Funktion s2 = new Summe(new Id(),s1);
```

Wir können sogar die Ableitung einer Funktion in die Summe mit einfließen lassen. So erzeugen wir etwa die Funktion

$$s_3(x) = s_1(x) + s_2'(x)$$

durch das Kommando

```
Funktion s3 = new Summe(s1,s2.getAbleitung());
```

Die Ableitung unserer Funktion erhalten wir durch den einfachen Aufruf

```
s3.getAbleitung();
```

Da das Ergebnis unserer Methode wieder ein Funktionsobjekt ist, können wir den Vorgang beliebig oft wiederholen. Die zweite Ableitung erhalten wir etwa durch

```
s3.getAbleitung().getAbleitung();
```

Der Computer setzt uns die entsprechenden Ableitungen automatisch korrekt zusammen.

4.3.3.2 Produkt zweier Funktionen

Kommen wir nun zurück zu unserer Ausgangsfunktion

$$f(x) = 3 \cdot x^2 + 7 \cdot x + 5.$$

Um diese Funktion auf dem Rechner korrekt darstellen zu können, fehlt uns nur noch eine letzte wichtige Grundfunktion: das *Produkt*

$$p(x) = a(x) \cdot b(x)$$

zweier Funktionen. Wir entwerfen zu diesem Zweck wieder eine Klasse `ZusammengesetzteKlasse` mit dem Namen `Produkt`, die die Ableitung nach der Formel

$$p'(x) = \underbrace{\underbrace{a'(x) \cdot b(x)}_{\texttt{teil1}} + \underbrace{a(x) \cdot b'(x)}_{\texttt{teil2}}}_{\texttt{gesamt}}$$

berechnet:

```
1  /** Diese Klasse repraesentiert ein Produkt */
2  public class Produkt extends Funktion {
3
4    /** Der Multiplikator */
5    private Funktion a;
6
7    /** Der Multiplikand */
8    private Funktion b;
9
10   /** Konstruktor */
11   public Produkt(Funktion a, Funktion b)  {
12     this.a = a;
13     this.b = b;
14   }
15   /** Werte die Funktion an einer gewissen Stelle aus */
16   public double getFunktionswert(double x) {
17     return a.getFunktionswert(x) * b.getFunktionswert(x);
18   }
19   /** Berechne die Ableitung der Funktion */
20   public Funktion getAbleitung() {
21     Funktion teil1 = new Produkt(a.getAbleitung(),b);
22     Funktion teil2 = new Produkt(a,b.getAbleitung());
23     Funktion gesamt = new Summe(teil1,teil2);
24     return gesamt;
25   }
26 }
```

Mit dieser letzten Klasse haben wir alle notwendigen Bauteile beisammen, um unsere Funktion f zusammensetzen zu können. Wir erinnern uns an die einzelnen Schritte, die es dabei zu tun gilt:

$$
\begin{array}{lcll}
f_1(x) & & & = \; 3 \\
f_2(x) & & & = \; 5 \\
f_3(x) & & & = \; 7 \\
f_4(x) & & & = \; x \\
f_5(x) & = & x^2 & \\
 & = & x \cdot x & = \; f_5(x) \cdot f_5(x) \\
f_6(x) & = & 7 \cdot x & = \; f_3(x) \cdot f_4(x) \\
f_7(x) & = & 7 \cdot x + 5 & = \; f_6(x) + f_2(x) \\
f_8(x) & = & 3 \cdot x^2 & = \; f_1(x) \cdot f_5(x) \\
f_9(x) & = & 3 \cdot x^2 + 7 \cdot x + 5 & = \; f_8(x) + f_7(x)
\end{array}
$$

Diese Schritte gilt es nun in Java umzusetzen:

```
Funktion f_1 = new Konstante(3);
Funktion f_2 = new Konstante(5);
Funktion f_3 = new Konstante(7);
Funktion f_4 = new Id();
Funktion f_5 = new Produkt(f_4,f_4);
Funktion f_6 = new Produkt(f_3,f_4);
Funktion f_7 = new Summe(f_6,f_2);
Funktion f_8 = new Produkt(f_1,f_5);
Funktion f_9 = new Summe(f_8,f_7);
```

Unser Funktionsobjekt `f_9` entspricht damit genau unserer Funktion f_9, die es zu konstruieren galt. Wir können diese durch die Zuweisung

```
Funktion f = f_9;
```

mit unserem Funktionsnamen f identifizieren. Die Ableitung f' dieser schon recht komplizierten Funktion erhalten wir dann einfach durch die Zuweisung

```
Funktion df = f.getAbleitung();
```

4.3.4 Variationen des Pattern

Eine der wohl bekanntesten Anwendungen des Composite-Pattern in Java ist das **Abstract Window Toolkit** (kurz **AWT**). Diese zu Java gehörende Klassensammlung ermöglicht es dem Benutzer, komplette Oberflächen mit Feuerköpfen, Schriftzügen, Fenstern und so weiter durch das Zusammensetzen einfacher Klassen zu modellieren.
Wir wollen hier kurz die Analogie zwischen dem uns bekannten Pattern und der Klassenstruktur dieser grafischen Komponenten hervorheben. Unsere `AllgemeineOberklasse` trägt hierbei den Namen `java.awt.Component`. Dabei handelt es sich um die Repräsentation eines beliebigen Objektes, das eine visuelle Darstellung auf dem Bildschirm besitzen kann. Entsprechend besitzt die Klasse (unter anderem) eine Methode `paint`, mit der das entsprechende Objekt gezeichnet werden kann.
Von der Klasse `Component` leiten sich verschiedene einfache Grundklassen ab. Dazu zählt etwa die Klasse `java.awt.Button`, die einen Knopf auf dem Bildschirm darstellt. Andere Beispiele wären `java.awt.Canvas` (engl. für Leinwand, also eine Zeichenfläche) oder `java.awt.Label` (ein Schriftzug). Hierbei handelt es sich also um einfache Klassen wie die Klassen `Id` und `Konstante` in unserem einführenden Beispiel.
Eine weitere Subklasse unserer Klasse `Component` ist die Klasse `Container`. Die Klasse `Container` ist die Oberklasse aller Grafikklassen, die einfache `Component`-Objekte zusammensetzen. Sie verfügt über verschiedene `add`-Methoden, mit deren Hilfe diverse Komponenten einem `Container` hinzugefügt werden können. Von der Klasse `Container` leiten sich nun die verschiedenen `ZusammengesetzteKlassen` ab, etwa die Klassen `java.awt.Panel` und `java.awt.Window`. In der entsprechenden Fachliteratur werden Sie nähere Informationen über die verschiedenen `Container`-Klassen und deren Anwendung finden.
Wir wollen an dieser Stelle natürlich nicht in die Tiefen der GUI-Programmierung einsteigen. Wie Abbildung 4.3 jedoch zeigt, handelt es sich hierbei um ein wunderschönes Beispiel für die Umsetzung des Musters. Wir haben tatsächlich eine `add`-Methode, mit der man beliebig viele Komponenten in einem `Container` zusammenfügen kann. Wir lernen ferner eine leichte Variation unseres Musters kennen: die Programmierer haben zur besseren Übersicht generalisiert und alle

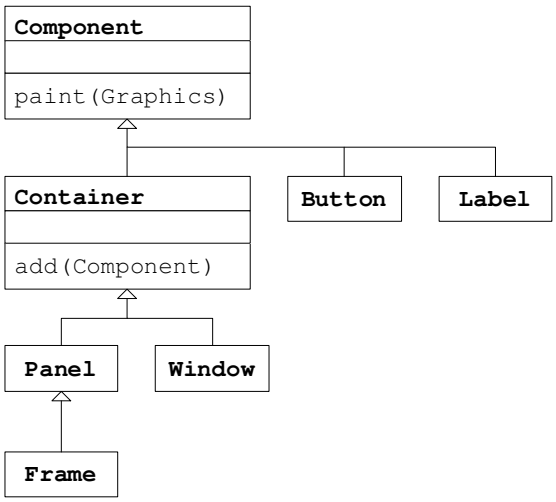


Abbildung 4.3: Das Composite-Pattern im Abstract Window Toolkit

Abbildung 4.4: Ein einfaches Fenster

zusammengesetzten Klassen in einer gemeinsamen Oberklasse `Container` zusammengefasst.
Eine einfache Anwendung zeigt, wie simpel der Einsatz dieser Grafikobjekte sein kann – sofern man das Composite-Pattern einmal verstanden hat. Wir wollen ein einfaches Fenster bauen, wie es in Abbildung 4.4 dargestellt ist. Zu diesem Zweck benötigen wir folgende Komponenten:

- ein `Label`, auf dem wir die Nachricht „mein erstes Fenster“ darstellen können,
- einen `Button`, der mit dem Text „Drueck mich“ beschriftet ist und
- einen Rahmen (`Frame`), der den Titel „simpleWindow“ trägt.

Diese grafischen Bestandteile können wir durch die Zeilen

```
Label label = new Label("Mein erstes Fenster");
Button button = new Button("Drueck mich");
Frame frame = new Frame("simpleWindow");
```

problemlos erzeugen. Nun müssen wir sie nur noch zusammensetzen.
Das Zusammensetzen der einzelnen Teile erfolgt mit der `add-Methode`. Hierbei sei erwähnt, dass in einem `Frame` üblicherweise nur eine einzige Kompo-

nente aufgenommen werden sollte.[4] Da wir es jedoch mit zweien (dem `Label` und dem `Button`) zu tun haben, bündeln wir diese einfach in einer der anderen `Container`-Klassen:

```
Panel panel = new Panel();
panel.add(label);
panel.add(button);
frame.add(panel);
```

Unser Fenster ist somit aus seinen einzelnen Komponenten erfolgreich zusammengebaut. Wir müssen es nun noch auf dem Bildschirm darstellen. Hierzu verwenden wir die Methoden `pack`[5] und `setVisible`. Unser komplettes Programm sieht nun wie folgt aus:

```
1  import java.awt.*;
2  /** Erzeugt ein einfaches Fenster */
3  public class simpleWindow {
4    /** Hauptprogramm */
5    public static void main(String[] args) {
6      // Folgende Dinge sollen zu sehen sein:
7      Label label = new Label("Mein erstes Fenster");
8      Button button = new Button("Drueck mich");
9      Frame frame = new Frame("simpleWindow");
10     // Setze die Objekte zusammen
11     Panel panel = new Panel();
12     panel.add(label);
13     panel.add(button);
14     frame.add(panel);
15     // Jetzt muss das Ganze noch gezeichnet werden
16     frame.pack();
17     frame.setVisible(true);
18   }
19 }
```

4.3.5 Zusammenfassung

Das **Composite-Pattern** stellt eine Art Baukastensystem dar. Aus einer Grundmenge von einfachen Klassen wird durch geschickte Kombination und Verkettung ein größeres Ganzes geschaffen.
Die Idee des **Kompositum**, wie das Entwurfsmuster gelegentlich ins Deutsche übersetzt wird, ist im täglichen Leben allgegenwärtig. Aus einer Menge von Grundbausteinen, den Atomen, entstehen „Verbände“ von Atomen: die Moleküle. Aus Aneinanderreihungen von Molekülen entstehen die verschiedensten Baustoffe – so unter anderem die Grundbausteine der menschlichen DNA. Aus den gerade einmal vier Grundinformationsträgern entsteht die so genannte Doppelhelix – und hiermit der gesamte „Bauplan“ eines menschlichen, tierischen oder pflanzlichen Lebens.

[4] Dies hat mit der Philosophie zu tun, dass der `Frame` (deutsch: Rahmen) den grafischen Inhalt eines Fensters umschließt, so wie ein Bilderrahmen ein Bild umschließt. Dieses Bild, das so genannte `ContentPane`, ist jenes Objekt, das dem Rahmen hinzugefügt wird.

[5] Berechnet die Größe der einzelnen zu zeichnenden Teile.

Auch im objektorientierten Programmieren ist dieses Vorgehen eines der am häufigsten verwendeten Entwurfsmuster. Bekannt ist es quasi jedem erfahrenen Java-Programmierer; selbst wenn er von Entwurfsmustern noch nie etwas gehört haben sollte. Allein das gesamte Abstract Window Toolkit, das die Programmierung grafischer Oberflächen ermöglicht, zwingt jeden Entwickler früher oder später dazu, sich mit der Grundidee des Kompositums zu befassen.

4.3.6 Übungsaufgaben

Aufgabe 4.2

Stellen Sie die Funktion

$$h(x) = 5 \cdot x + \sin(3 \cdot x + 4)$$

mit Hilfe der in diesem Kapitel vorgestellten Klassen dar. Für die hierbei verwendeten trigonometrischen Funktionen entwerfen Sie zwei neue Klassen `Sinus` und `Kosinus`, die entsprechend

$$\sin(a(x)) \text{ und } \cos(a(x))$$

darstellen. Für die Ableitungen verwenden Sie folgende Formeln:

$$\begin{aligned} \sin'(a(x)) &= a'(x) \cdot \cos(a(x)) \\ \cos'(a(x)) &= -a'(x) \cdot \sin(a(x)) \end{aligned}$$

Aufgabe 4.3

Erzeugen Sie ein Fenster, in dem drei Buttons mit der Aufschrift „Eins“, „Zwei“ und „Drei“ zu sehen sind.

Teil II

Praxis

Ergänzung 5

Praxisbeispiele: Einzelne Etüden

5.1 Teilbarkeit zum Ersten

5.1.1 Vorwissen aus dem Buch

Am Anfang jeder Aufgabe werden wir wir kurz zusammenfassen, welche neuen Konzepte in diesem Abschnitt behandelt werden. Sollte der Leser mit diesen Teilen des Buches noch nicht vertraut sein, kann dies entsprechend dann nachgeholt werden. Für diese Aufgabe handelt es sich um

- Abschnitt 4.3 (Einfache Datentypen)
- Abschnitt 4.4 (Der Umgang mit einfachen Datentypen) sowie
- Abschnitt 4.5 (Anweisungen und Ablaufsteuerung).

5.1.2 Aufgabenstellung

Gegeben sei eine dreistellige ganze Zahl zwischen 100 und 999. Durch welche ihrer Ziffern ist diese Zahl teilbar?

5.1.3 Analyse des Problems

Wir haben in dieser Aufgabe zwei Nüsse zu knacken:

- Wie spalte ich eine Zahl in ihre Ziffern auf?
- Wie prüfe ich, ob eine Zahl durch eine andere teilbar ist?

Wir wollen uns mit dem ersten Problem näher beschäftigen. Nehmen wir etwa die Zahl 123 und versuchen, sie mit den uns bekannten Operatoren zu unterglie-

dern. Wie kommen wir etwa an die Einerstelle heran? Wir erinnern uns an die Schulmathematik, nach der wir Division mit Rest wie folgt durchgeführt haben:

Konsole

```
123 : 10 = 12 mit Rest 3.
```

Wir sehen also, dass eine simple Division uns die rechte Ziffer bescheren kann (diese ist nämlich der Rest). Wir können in Java diese Berechnung mit Hilfe des Rest-Operators `%` durchführen.
Wie kommen wir aber an die Zehner- oder Hunderterstelle heran? Auch diese Frage haben wir mit obiger Rechnung beantwortet. Teilen wir 123 durch 10, so erhalten wir als Ergebnis 12. Die Zehnerstelle ist also um eine Position nach rechts gerückt und kann somit wieder durch den Rest-Operator berechnet werden.
Es bleibt noch die Frage, wie wir die Teilbarkeit überprüfen. Nehmen wir zu Anfang einmal an, die Ziffer sei ungleich der Null (eine Division wäre sonst schließlich nicht möglich). In diesem Fall ist eine Zahl durch eine andere offensichtlich teilbar, wenn bei der Division kein Rest entsteht. Ist der Rest also gleich 0, ist die Teilbarkeit erfüllt.

5.1.4 Algorithmische Beschreibung

Was ist ein Algorithmus? Wir wollen einen Algorithmus als eine Verfahrensvorschrift zur Lösung eines bestimmten Problems bezeichnen. Bevor wir das Programm in Java umsetzen, werden wir stets die wichtigsten Punkte in einer Kurzbeschreibung zusammenfassen, um uns einen Überblick über das Verfahren zu verschaffen.
Folgendes Vorgehen erscheint nach unseren Überlegungen als logisch:

1. *Initialisierung* (d. h. vorbereitende Maßnahmen)
   (a) Weise der Variablen `zahl` den zu prüfenden Wert zu
2. *Bestimmung der einzelnen Ziffern*
   (a) Bestimme die Einerstelle: `einer = zahl % 10`
   (b) Bestimme die Zehnerstelle: `zehner = (zahl / 10) % 10`
   (c) Bestimme die Hunderterstelle: `hunderter = (zahl / 100)`
3. *Teilbarkeitstest*
   (a) Ist `einer` ungleich Null und ergibt `zahl/einer` keinen Rest, gib `einer` aus
   (b) Ist `zehner` ungleich Null und ergibt die Division keinen Rest, gib `zehner` aus
   (c) verfahre genauso mit den Hundertern

Wir wollen nun überlegen, wie wir dieses Programm in Java implementieren.

5.1.5 Programmierung in Java

Wir öffnen im Editor eine Datei `Teilbarkeit1.java` und beginnen wie üblich zuerst mit dem Programmkopf

```
/**
 @author Jens Scheffler
 @version 1.0
 */

/**
 Dieses Programm spaltet eine dreistellige Zahl in ihre Ziffern auf
 und testet, ob die Zahl durch ihre einzelnen Ziffern teilbar ist.
 */

public class Teilbarkeit1 {
  public static void main(String[] args) {
```

Als Nächstes beginnen wir mit der Initialisierung. Wir benötigen eine Zahl, die es zu testen gilt. Wir können diese entweder im Programm zuweisen oder von der Tastatur einlesen. Im letzteren Fall verwenden wir die IOTools, wie es im Anhang beschrieben wird. Wir schreiben also eine der folgenden Zeilen:

```
int zahl = 123;                   // Konstante festlegen
int zahl = IOTools.readInteger(); // Eingabe per Tastatur
```

Als Nächstes müssen wir die Zahl in ihre einzelnen Ziffern aufspalten. Hierzu können wir die Rechenvorschriften aus dem letzten Abschnitt übernehmen.

```
int einer     =  zahl        % 10;  // Bestimme die Einer
int zehner    = (zahl /  10) % 10;  // Bestimme die Zehner
int hunderter = (zahl / 100);       // Bestimme die Hunderter
```

Wir haben jetzt also die einzelnen Ziffern und die ganze Zahl in Variablen gespeichert. Kommen wir also zum letzten Punkt des Algorithmus – dem Teilbarkeitstest. Für die erste Stelle sieht dieser etwa wie folgt aus:

```
if (einer != 0 &&                 // Ist Division moeglich?
    zahl % einer == 0)            // Ist der Rest =0 ?
  System.out.println("Die Zahl " + zahl + " ist durch "
                     + einer + " teilbar!");
```

Was passiert hierbei in der Bedingung? Kann jemals eine Division durch Null auftreten, falls die „Einsstelle“ `einer` gleich 0 ist? Dies würde schließlich zu einem Programmabsturz führen!
Die beruhigende Antwort ist: nein! Der `&&`-Operator wertet den zweiten Operanden nämlich nur dann aus, wenn nach Auswertung des ersten Operanden der endgültige Wert der `&&`-Operation noch nicht feststeht. Ist `einer==0`, so ist der erste Teil der Bedingung ohnehin schon falsch – sie kann also nicht mehr erfüllt werden. Das Programm rechnet an dieser Stelle nicht weiter und der Fehler tritt nicht auf.
Schließlich dürfen wir natürlich nicht vergessen, die geöffneten Klammern auch wieder zu schließen. Unser fertiges Programm sieht nun wie folgt aus:

```
1  import Prog1Tools.IOTools;
2
3  public class Teilbarkeit1 {
4    public static void main(String[] args) {
5      int zahl = IOTools.readInteger();   // Eingabe per Tastatur
6      int einer     =  zahl        % 10;  // Bestimme die Einer
7      int zehner    = (zahl /  10) % 10;  // Bestimme die Zehner
8      int hunderter = (zahl / 100);       // Bestimme die Hunderter
9      if (einer != 0 &&                   // Ist Division moeglich?
10         zahl % einer == 0)              // Ist der Rest =0 ?
11       System.out.println("Die Zahl " + zahl + " ist durch "
12                          + einer + " teilbar!");
13     if (zehner != 0 &&                  // Ist Division moeglich?
14         zahl % zehner == 0)             // Ist der Rest =0 ?
15       System.out.println("Die Zahl " + zahl + " ist durch "
16                          + zehner + " teilbar!");
17     if (hunderter != 0 &&               // Ist Division moeglich?
18         zahl % hunderter == 0)          // Ist der Rest =0 ?
19       System.out.println("Die Zahl " + zahl + " ist durch "
20                          + hunderter + " teilbar!");
21   }
22 }
```

Wir übersetzen das Programm mit dem Befehl `javac Teilbarkeit1.java` und starten es anschließend. Geben wir etwa die Zahl 123 ein, erhalten wir als Ausgabe

Konsole

```
Die Zahl 123 ist durch 3 teilbar!
Die Zahl 123 ist durch 1 teilbar!
```

5.1.6 Vorsicht, Falle!

Welche Stolperstricke haben sich in dieser Aufgabe für uns ergeben? An folgenden Stellen treten beim Programmieren gerne Fehler auf:

- Wir vergessen das eine oder andere Semikolon. Der Compiler bedankt sich mit einer Fehlermeldung der Form

  Konsole

  ```
  ';' expected.
  ```

- Wir verwechseln in einer **`if`**-Abfrage die Operatoren == und =. Wir erhalten die Fehlermeldung

  Konsole

  ```
  Invalid left hand side of assignment.
  ```

- Wir schreiben in der **`if`**-Abfrage anstelle von `&&` den `&`-Operator. Der Compiler beschwert sich zwar nicht – das Programm ist syntaktisch vollkommen

korrekt. Starten wir aber das Programm und geben an der falschen Stelle eine Null ein, so erhalten wir

Konsole
```
java.lang.ArithmeticException: / by zero
```

und das Programm stürzt ab.

5.1.7 Übungsaufgaben

Aufgabe 5.1

Statt der Zahl wollen wir testen, ob deren Quersumme durch eine der einzelnen Ziffern teilbar ist.

5.2 Teilbarkeit zum Zweiten

5.2.1 Vorwissen aus dem Buch

- Abschnitt 4.3 (Einfache Datentypen)
- Abschnitt 4.4 (Der Umgang mit einfachen Datentypen) sowie
- Abschnitt 4.5 (Anweisungen und Ablaufsteuerung).

5.2.2 Aufgabenstellung

Wir wollen das vorherige Problem noch einmal lösen – nur darf die Zahl diesmal beliebig lang sein.

5.2.3 Analyse des Problems

Wir wissen nicht, wievielstellig die neue Zahl ist. Wie sollen wir sie also in einzelne Ziffern aufteilen?
An dieser Stelle müssen wir deshalb ein wenig umdenken. Haben wir im vorigen Abschnitt zuerst *alle* Ziffern berechnet und dann den Teilbarkeitstest gemacht, werden wir nun eine Ziffer nach der anderen betrachten müssen. Die Rechenvorschrift für das Erhalten der einzelnen Ziffern ist hierbei identisch. Wir berechnen den Rest der Division, um die Einerstelle zu erhalten. Danach teilen wir die Zahl durch 10, um alle Ziffern nach rechts zu schieben.
Diese Vorgehensweise wirft natürlich wieder zwei neue Probleme auf:

- Wenn wir die Zahl durch 10 teilen, wie sollen wir sie dann noch vergleichen?
- Wann können wir mit der Ziffernberechnung aufhören?

Der erste Fall ist relativ einfach zu lösen. Wir kopieren den Inhalt der Variable `zahl` einfach in eine andere Variable `dummy`, die wir nun nach Belieben verändern können. Das Original bleibt uns jedoch erhalten.
Auch auf die zweite Frage (die Frage nach dem so genannten Abbruchkriterium) ist schnell eine Antwort gefunden. Wir hören auf, sobald alle Ziffern abgearbeitet sind. Dies ist der Fall, wenn in `dummy` keine weiteren Ziffern stehen, also `dummy == 0` gilt.

5.2.4 Algorithmische Beschreibung

1. *Initialisierung*
   (a) Weise der Variablen `zahl` den zu prüfenden Wert zu.
   (b) Erstelle eine Kopie von `zahl` in der Variablen `dummy`.
2. *Schleife*
   Wiederhole die folgenden Instruktionen, solange `dummy!=0` ist:
   (a) Bestimme die Einerstelle: `einer = dummy % 10`
   (b) Schiebe die Ziffern nach rechts: `dummy = dummy / 10`
   (c) Führe den Teilbarkeitstest durch.

5.2.5 Programmierung in Java

Da sich das Programm in vielen Punkten nicht von dem vorherigen unterscheidet, stellen wir es gleich in einem Stück vor:

```
1  /**
2   @author Jens Scheffler
3   @version 1.0
4   */
5
6  /**
7   Dieses Programm spaltet eine Zahl in ihre Ziffern auf
8   und testet, ob die Zahl durch ihre einzelnen Ziffern teilbar ist.
9   */
10
11 import Prog1Tools.IOTools;
12
13 public class Teilbarkeit2 {
14   public static void main(String[] args) {
15     // 1. INITIALISIERUNG
16     // ==================
17     int zahl = IOTools.readInteger(); // Eingabe per Tastatur
18     int dummy = zahl;                 // Kopie erstellen
19     // 2. SCHLEIFE
20     // ===========
21     while (dummy != 0) {           // Schleifenbedingung
22       int einer = dummy % 10;      // Berechne einer
```

```
23        dummy = dummy / 10;          // Schiebe Ziffern nach rechts
24        if (einer != 0 &&            // Ist Division moeglich?
25            zahl % einer == 0)       // Ist der Rest =0 ?
26          System.out.println("Die Zahl " + zahl + " ist durch "
27                             + einer + " teilbar!");
28      }                              // Schleifenende
29    }
30  }
```

Wir sehen, dass wir den Teilbarkeitstest wortwörtlich aus dem Programm `Teilbarkeit1` übernehmen konnten. Ansonsten ist das Programm durch die Verwendung der Schleife sogar noch etwas kürzer geworden. Starten wir das Programm nun und geben etwa die Zahl `123456` ein, so erhalten wir folgende Ausgabe:

Konsole

```
Die Zahl 123456 ist durch 6 teilbar!
Die Zahl 123456 ist durch 4 teilbar!
Die Zahl 123456 ist durch 3 teilbar!
Die Zahl 123456 ist durch 2 teilbar!
Die Zahl 123456 ist durch 1 teilbar!
```

5.2.6 Vorsicht, Falle!

Neben den bereits von Aufgabe 1 bekannten Problemen gibt es hier noch einen weiteren Punkt, auf den zu achten ist. Hatten wir im ersten Programm noch lediglich zwei Klammern geöffnet (eine für den Programmbeginn, eine für den Start der Hauptmethode), so ist durch die Schleife noch eine weitere Klammer hinzugekommen. Schließen wir diese nicht, so erhalten wir die Fehlermeldung

Konsole

```
'}' expected.
```

Wir könnten natürlich auch auf die Idee kommen, die zu der Schleife gehörigen Klammern ganz wegzulassen. In diesem Fall würde sich die Schleife aber nur auf die erste Instruktion auswirken. Da die erste Instruktion jedoch eine neue Variable definiert und diese nun nicht mehr Teil eines eigenständigen Blockes ist, erhalten wir auf einen Schlag gleich einen Haufen von Fehlern:

Konsole

```
Teilbarkeit2.java:20: Invalid declaration.
       int einer = dummy % 10;       // Berechne einer
       ^
Teilbarkeit2.java:22: Undefined variable: einer
       if (einer != 0 &              // Ist Division moeglich?
           ^
Teilbarkeit2.java:23: Undefined variable: einer
           zahl % einer == 0)        // Ist der Rest =0 ?
```

```
                    ^
Teilbarkeit2.java:23: Incompatible type for ==. Can't convert int
                                                          to boolean.
            zahl % einer == 0)        // Ist der Rest =0 ?
                          ^
Teilbarkeit2.java:25: Undefined variable: einer
                             + einer + " teilbar!");
```

Von diesen fünf Fehlern ist nur einer in unserer Unachtsamkeit begründet; die anderen ergeben sich allesamt aus dem ersten als Folgefehler. Deshalb ein Tipp: *Niemals einen Fehler korrigieren, wenn man sich nicht hundertprozentig sicher ist, dass es sich um keinen Folgefehler handelt. Meist lässt schon die Korrektur des ersten Fehlers eine Menge anderer Fehlermeldungen verschwinden!*

5.2.7 Übungsaufgaben

Aufgabe 5.2

Das Programm soll so erweitert werden, dass es zusätzlich überprüft, ob die Zahl auch durch ihre Quersumme teilbar ist.

5.3 Dreierlei

5.3.1 Vorwissen aus dem Buch

- Abschnitt 4.3 (Einfache Datentypen)
- Abschnitt 4.4 (Der Umgang mit einfachen Datentypen) sowie
- Abschnitt 4.5 (Anweisungen und Ablaufsteuerung).

5.3.2 Aufgabenstellung

Wegen großem Verletzungspech muss der Trainer einer Bundesligamannschaft für ein Pokalspiel drei Nachwuchsspieler aus der Amateurmannschaft rekrutieren. Er setzt sich deshalb mit dem Betreuer der Jugendlichen zusammen, der ihm die fünf verheißungsvollsten Talente vorstellt:

> *„Da hätten wir als Erstes Al. Al ist ein wirklich guter Stürmer, aber manchmal etwas überheblich. Sie sollten auf jeden Fall Cid einsetzen, falls Al spielt. Cid ist der ruhende Pol bei uns; er sorgt dafür, dass die Jungs auf dem Teppich bleiben. Das gilt übrigens besonders auch für Daniel! Wenn Sie Daniel einsetzen, darf Cid auf keinen Fall fehlen.*
>
> *Apropos Daniel: Nachdem ihm Bert seine Freundin ausgespannt hat, sind die beiden nicht gut aufeinander zu sprechen. Die beiden giften sich nur an und sollten auf keinen Fall in einer Mannschaft sein. Sollten Sie aber trotzdem Bert wollen, so*

müssen Sie auf jeden Fall auch Ernst einsetzen. Ernst und Bert sind ein langjähriges Team – ihr Kombinationsspiel ist einfach traumhaft!“

Welche drei Spieler sollte der Trainer nun aufstellen?

5.3.3 Analyse des Problems

Wie bei jeder Textaufgabe müssen wir auch hier zuerst einmal alle relevanten Informationen herausfinden, die uns die Lösung des Problems erst ermöglichen.

1. Der Trainer braucht genau drei Spieler – nicht mehr und nicht weniger!
2. Wenn Al spielt, muss auch Cid spielen.
3. Wenn Daniel spielt, muss auch Cid spielen.
4. Bert und Daniel dürfen nicht gemeinsam spielen.
5. Ernst und Bert dürfen nur gemeinsam spielen.

Wie kann man diese Informationen nun nutzen, um ein Ergebnis zu erzielen? Die erste Möglichkeit ist, sich Papier und Bleistift zu nehmen, alle Bedingungen in einem logischen Ausdruck zusammenzufassen und diesen zu vereinfachen. Dies wollen wir aber nicht tun – wir wollen programmieren.
Die einfachste Art, alle möglichen Lösungen zu erhalten, ist wohl simples Ausprobieren. Wir kombinieren alle Spieler miteinander und schauen, welche Kombinationen die Bedingungen erfüllen. Hierzu definieren wir pro Spieler eine **`boolean`**-Variable. Ist der Inhalt der Variable **`true`**, bedeutet dies, dass er spielt, ist der Inhalt **`false`**, dass er nicht spielt.

5.3.4 Algorithmische Beschreibung

Die Idee des Verfahrens ist so einfach, dass man versucht ist, das gesuchte Programm direkt zu erstellen. Da in diesem Vorgehen jedoch eine Quelle vieler Fehler liegt, wollen wir auch diese Situation erst analysieren. Was ist zu tun?
Konstruiere eine fünffach geschachtelte Schleife, die für alle fünf Spieler alle möglichen Belegungen (**`true`** und **`false`**) durchläuft. Mache darin folgende Tests:

- Teste, ob genau drei der fünf Variablen wahr sind.
- Teste, ob C. spielt, falls A. spielt.
- Teste, ob C. spielt, falls D. spielt.
- Teste, ob B. und D. nicht zusammen spielen.
- Teste, ob B. und E. gemeinsam spielen (falls einer spielt).

Treffen alle fünf Tests zu, gib die Kombination aus.

5.3.5 Programmierung in Java

Die erste Frage, die sich stellt, ist: Wie können wir alle Kombinationen der fünf Variablen erhalten? Wir verschachteln hierzu fünf **do-while**-Schleifen. Folgende Schleife würde beispielsweise die Variable `bert` hintereinander auf **true** und **false** setzen.

```
boolean bert = true;
do {
  // Hier eventuelle weitere Schleifen oder Test einfuegen
  bert = !bert;
} while (bert != true);
```

Zu Beginn der Schleife ist `bert == true`, wird aber negiert, bevor die Schleifenbedingung das erste Mal überprüft wird. Auf diese Weise wird die Schleife noch ein zweites Mal für `bert == false` durchlaufen. Bevor die Bedingung ein zweites Mal abgefragt wird, setzt die Anweisung `bert = !bert;` die Variable wieder auf **true**. Die Schleife bricht ab.
Wir wollen zuerst den Programmrumpf und die fünf verschachtelten Schleifen implementieren. Es ergibt sich folgendes Listing:

```
1  /**
2   @author Jens Scheffler
3   @version 1.0
4   */
5
6  import Prog1Tools.IOTools;
7
8  public class ThreeGuys {
9    public static void main(String[] args) {
10     boolean al = true;
11     do {
12       boolean bert = true;
13       do {
14         boolean cid = true;
15         do {
16           boolean daniel = true;
17           do {
18             boolean ernst = true;
19             do {
20 // das Ergebnis der Tests steht in dieser Variable
21               boolean testergebnis;
22 // ================================
23 // HIER DIE FUENF TESTS EINFUEGEN!!!
24 // ================================
25               // Ausgabe, falls testergebnis==true
26               if (testergebnis)
27                 System.out.println("A:" + al + "  B:" +
28                                    bert + "  C:" + cid +
29                                    "  D:" + daniel +
30                                    "  E:" + ernst);
31               ernst = !ernst;        // negiere Variable
32             } while (ernst != true);
33             daniel = !daniel;        // negiere Variable
34           } while (daniel != true);
```

```
35            cid = !cid;                 // negiere Variable
36          } while (cid != true);
37          bert = !bert;                 // negiere Variable
38        } while (bert != true);
39        al = !al;                       // negiere Variable
40      } while (al != true);
41    }
42  }
```

Es stellt sich nun die Frage, wie man die verbliebenen fünf Tests am besten implementiert. Beginnen wir mit dem ersten. Wir müssen die Anzahl der Variablen herausfinden, die den Wert **true** besitzen. Dies lässt sich recht einfach wie folgt bewerkstelligen:

```
int counter = 0;
if (al) counter++;
if (bert) counter++;
if (cid) counter++;
if (daniel) counter++;
if (ernst) counter++;
```

Nach Durchlaufen der letzten Zeile steht in `counter` die gesuchte Zahl der Variablen. Das Testergebnis ergibt sich durch den Vergleich `counter == 3`.
Kommen wir zum zweiten Punkt: Wenn Al spielt (**if** `(al)`), so muss auch Cid (`testergebnis=cid`) spielen. Wir dürfen an dieser Stelle jedoch nicht vergessen, dass das Endergebnis nur stimmt, wenn *alle* einzelnen Tests korrekt sind. Sofern wir also die Variable `testergebnis` verändern, müssen wir ihren alten Wert mit einfließen lassen:

```
if (al)
  testergebnis = testergebnis && cid;
```

Der dritte Test verläuft analog zum zweiten. Im vierten Fall können wir das Testergebnis auf **false** setzen, sofern Bert und Daniel spielen (also `bert && daniel`). Fall fünf läuft auf einen Vergleich der Inhalte von `bert` und `ernst` hinaus – diese müssen gleich sein.
Hier das komplette Listing mit allen fünf Tests:

```
1  import Prog1Tools.IOTools;
2  public class ThreeGuys {
3    public static void main(String[] args) {
4      boolean al = true;
5      do {
6        boolean bert = true;
7        do {
8          boolean cid = true;
9          do {
10           boolean daniel = true;
11           do {
12             boolean ernst = true;
13             do {
14   // das Ergebnis der Tests steht in dieser Variable
15               boolean testergebnis;
16
```

```
17                // Test 1: Zaehle die aufgestellten Spieler
18                int counter = 0;
19
20                if (al) counter++;
21                if (bert) counter++;
22                if (cid) counter++;
23                if (daniel) counter++;
24                if (ernst) counter++;
25
26                testergebnis = (counter == 3);
27
28                // Test 2: Wenn A spielt, spielt auch C?
29                if (al)
30                  testergebnis = testergebnis && cid;
31                // Test 3: Wenn D spielt, spielt auch C?
32                if (daniel)
33                  testergebnis = testergebnis && cid;
34                // Test 4: Wenn B spielt, darf D nicht spielen
35                // (und umgekehrt)
36                if (bert && daniel)
37                  testergebnis = false;
38                // Test 5: Spielen B und E gemeinsam?
39                testergebnis = testergebnis & (bert == ernst);
40                // Ausgabe, falls testergebnis==true
41                if (testergebnis)
42                  System.out.println("A:" + al + "  B:" +
43                                     bert + "  C:" + cid +
44                                     "  D:" + daniel +
45                                     "  E:" + ernst);
46                ernst = !ernst;         // negiere Variable
47              } while (ernst != true);
48              daniel = !daniel;         // negiere Variable
49            } while (daniel != true);
50            cid = !cid;                 // negiere Variable
51          } while (cid != true);
52          bert = !bert;                 // negiere Variable
53        } while (bert != true);
54        al = !al;                       // negiere Variable
55      } while (al != true);
56    }
57  }
```

Übersetzen wir das Programm und lassen es laufen, so erhalten wir folgende Ausgabe:

Konsole

```
A:true  B:false  C:true  D:true  E:false
A:false  B:true  C:true  D:false  E:true
```

Der Trainer kann also entweder Al, Cid und Daniel oder aber Bert, Cid und Ernst aufstellen.
Hier könnte man sich natürlich fragen, warum eine so kleine Aufgabe, für die man durch einiges Nachdenken leicht eine Lösung hätte finden können, durch ein so langes Programm gelöst werden soll. All diesen Zweiflern sei ans Herz

gelegt, einmal zu versuchen, ein wesentlich kürzeres Programm zu finden, das alle Aufgaben dieses Typs in möglichst kurzer Rechenzeit löst. Gelingt es Ihnen bei n Spielern und einer beliebigen Anzahl von Restriktionen, ein Programm zu finden, das weniger als 2^n Schritte benötigt?
Hinweis: Hierbei handelt es sich um einen Teilfall eines bisher noch nicht gelösten wissenschaftlichen Problems. Weitere Informationen können Sie unter den Stichworten *Erfüllbarkeitsproblem* oder *P-NP-Problem* in der Fachliteratur, etwa in [7] oder in [1] finden.

5.3.6 Vorsicht, Falle!

Noch intensiver als in der vorherigen Aufgabe arbeiten wir hier mit Blöcken, geöffneten und geschlossenen Klammern. Das Hauptproblem an dieser Stelle ist deshalb wirklich, diese zum richtigen Zeitpunkt zu öffnen und wieder zu schließen. Eine strukturierte Programmierung (Einrücken!) wirkt dabei Wunder.

5.3.7 Übungsaufgaben

Aufgabe 5.3

Bert und Daniel haben sich urplötzlich wieder vertragen. Welche neuen Möglichkeiten ergeben sich für den Trainer?

5.4 Das Achtdamenproblem

5.4.1 Vorwissen aus dem Buch

- Vorwissen der vorigen Kapitel
- Abschnitt 5.1 (Felder) sowie
- Kapitel 6 (Methoden).

5.4.2 Aufgabenstellung

Auch wenn nicht jeder ein Großmeister des Schach sein dürfte, haben wir doch wohl schon alle von diesem Spiel gehört. Wir betrachten also ein Schachbrett, das bekanntlich $8 * 8 = 64$ Felder besitzt. Auf diesem Brett wollen wir acht Damen so verteilen, dass keine die andere schlagen kann.
Zur Lösung dieses Problems werden wir spaltenweise „von links nach rechts“ vorangehen, d. h. wir setzen unsere erste Dame in die linke Spalte. Da eine Dame senkrecht, waagerecht und diagonal schlagen kann, darf in dieser Spalte nun keine Dame mehr stehen. Wir setzen unsere nächste Dame deshalb in die nächste Zeile (und so weiter). Stehen die bereits gesetzten Damen so ungünstig, dass wir keine weitere mehr setzen können, gehen wir einfach einen Schritt zurück und

versetzen die letzte Dame. Kann diese nicht versetzt werden, gehen wir wieder einen Schritt zurück und so weiter. Man bezeichnet diese Vorgehensweise auch als **Rückverfolgung** oder **Backtracking**.

5.4.3 Lösungsidee

Unsere Vorgehensweise entspricht also einem systematischen Ausprobieren. Wir teilen den Algorithmus in drei Teilbereiche auf:

- eine Methode `bedroht`, in der wir überprüfen, ob die zuletzt gesetzte Dame im Zugbereich einer der anderen Damen steht.
- eine Methode `ausgabe`, in der wir die einmal gefundene Lösung auf dem Bildschirm ausgeben, und
- eine Methode `setze`, mit der wir versuchen, die Damen an den richtigen Stellen zu platzieren.

Um die Positionen der einzelnen Damen zu speichern, definieren wir ein eindimensionales Feld der Länge 8 mit Namen `brett`. Jede Feldkomponente entspricht der Position einer Dame. Dabei steht der Index der Feldkomponente für die Spalte des Schachbretts und der Wert der Feldkomponente für die Zeile des Schachbretts in der sich die Dame befindet. So steht etwa `brett[0]` für die Zeilennummer der Dame in der ersten Spalte (Spalte 0), `brett[1]` für die Zeilennummer der Dame in der zweiten Spalte (Spalte 1) und so weiter. Falls also etwa `brett[0]` den Wert 0 hat, so steht eine Dame in der linken oberen Ecke.

5.4.4 Erste Vorarbeiten: Die Methoden `ausgabe` und `bedroht`

Wir wollen uns als Erstes an die Formulierung der Methode `ausgabe` machen, da diese am wenigsten Arbeit erfordert. Um eine halbwegs übersichtliche Ausgabe zu gewährleisten, wollen wir das Brett zeilenweise wie folgt auf dem Bildschirm ausgeben:

- Steht auf der Brettposition `(i,j)` eine Dame, d. h. ist `brett[j] == i`, so gib ein `D` auf dem Bildschirm aus.
- Ist dem nicht so, gib ein Leerzeichen aus.

Um später das Ergebnis leichter überprüfen zu können, trennen wir die einzelnen Spalten durch einen Balken. Die Ausgabe ist leicht durch zwei geschachtelte **`for`**-Schleifen zu bewerkstelligen. Folgende Methode liefert das gewünschte Ergebnis:

```
public static void ausgabe(int[] brett) {
  for (int i=0; i < 8; i++) {      // Anzahl der Zeilen
    for (int j=0; j < 8; j++)      // Anzahl der Spalten
      System.out.print("|" + ((i == brett[j]) ? 'D' : ' '));
    System.out.println("|");       // Zeilenende
  }
}
```

Wir verwenden hierbei den ternären Operator `?:`, um uns eine **`if`**-Abfrage zu ersparen.
Als Nächstes gehen wir die Umsetzung der Methode `bedroht` an. Diese soll einen **`boolean`**-Wert zurückliefern, und zwar **`true`**, falls eine Bedrohung der zuletzt gesetzten Dame vorliegt. Wir müssen der Methode also neben dem Feld auch die Nummer der aktuellen Spalte als Parameter übergeben. Wir tun dies mit einem ganzzahligen Parameter namens `spalte`.
Um herauszufinden, ob die in der aktuellen Spalte gesetzte Dame durch eine andere Dame bedroht wird, müssen wir drei Tests durchführen:

1. Befindet sich in der gleichen Zeile noch eine andere Dame, die also waagerecht schlagen könnte? Dies wäre der Fall, wenn wir in einem vorherigen Schritt bereits eines der Elemente von `brett` auf die gleiche Zahl gesetzt hätten. Es gäbe also eine Zahl `i` zwischen 0 und `spalte`, für die `brett[i]==brett[spalte]` gilt. Wir können den Test wie folgt formulieren:

   ```
   for (int i=0; i < spalte; i++)
     if (brett[i] == brett[spalte])
       return true;
   ```

2. Befindet sich in der Diagonale, die schräg nach oben links verläuft, eine Dame? Dieser Test ist nicht ganz so einfach, da wir die Testbedingung nicht so leicht wie oben angeben können.

   Wir überlegen uns deshalb an einem Beispiel, wie die Schleife auszusehen hat. Angenommen, wir befinden uns in der dritten Spalte (also `spalte==2`, da wir von der Null aus zählen) und setzen die Dame auf die fünfte Zeile (also `brett[2]==4`). Genau dann befindet sich eine Dame in der Diagonale, wenn `brett[1]==3` oder `brett[0]==2` ist. Wir müssen also sowohl bei der Spaltenzahl als auch bei der zu überprüfenden Zeilennummer jeweils um den Wert `1` heruntergehen. Deshalb führen wir neben der Laufvariablen `i` noch eine Variable `j` ein, in der wir für jede zu prüfende Spalte die zugehörige Zeilennummer speichern. Unsere Überprüfung funktioniert nun wie folgt:

   ```
   for (int i = spalte-1, j = brett[spalte]-1; i >= 0; i--,j--)
     if (brett[i] == j)
       return true;
   ```

3. Befindet sich in der Diagonale, die schräg nach unten links verläuft, eine Dame? Die Überprüfung dieser Bedingung funktioniert genau wie die andere Diagonalrichtung – mit dem Unterschied, dass wir die Variable `j` nun erhöhen statt erniedrigen müssen:

   ```
   for (int i = spalte-1, j = brett[spalte]+1; i >= 0; i--,j++)
     if (brett[i] == j)
       return true;
   ```

Hat eine Situation auf dem Spielbrett alle drei Tests überstanden, so ist die zu testende Dame nicht bedroht; wir können also den Wert **false** zurückgeben. Unsere Methode sieht nun wie folgt aus:

```
public static boolean bedroht(int[] brett, int spalte) {
  // Teste als Erstes, ob eine Dame in derselben Zeile steht
  for (int i=0; i < spalte; i++)
    if (brett[i] == brett[spalte])
      return true;

  // Teste nun, ob in der oberen Diagonale eine Dame steht
  for (int i = spalte-1, j = brett[spalte]-1; i >= 0; i--,j--)
    if (brett[i] == j)
      return true;

  // Teste, ob in der unteren Diagonale eine Dame steht
  for (int i = spalte-1, j = brett[spalte]+1; i >= 0; i--,j++)
    if (brett[i] == j)
      return true;

  // Wenn das Programm hier angekommen ist, steht die Dame "frei"
  return false;
}
```

5.4.5 Die Rekursion

Wir kommen nun zur letzten und allem Anschein nach schwierigsten Methode: der Methode `setze`. Wir haben uns bereits überlegt, wie der Algorithmus auszusehen hat. Wir beginnen bei `spalte=0` und setzen die Dame in die Zeile 0. Als Nächstes setzen wir die Dame in der zweiten Zeile auf das erste Feld, das nicht besetzt ist (und so weiter). Gibt es keine Möglichkeit mehr, eine Dame zu setzen, gehen wir wieder eine Spalte zurück und versuchen, die letzte gesetzte Dame auf einen anderen Platz zu bringen. Gibt es hierfür wieder keine Möglichkeit, gehen wir wieder eine Spalte zurück.
Wie wir sehen, ist der Algorithmus ziemlich kompliziert. Die einzelnen Spalten beeinflussen sich gegenseitig und wir wissen nicht im Voraus, bis zu welcher Zeile wir etwa die Suche in der fünften Spalte durchzuführen haben. Das Resultat ist eine Verschachtelung von mindestens *acht* **for**-Schleifen mit diversen **break**-Anweisungen, bei denen man sehr leicht den Überblick verliert. Geht das nicht auch einfacher?
Dank rekursiver Programmiertechnik können wir diese Frage mit reinem Gewissen bejahen. Wenn wir uns die Vorgehensweise nämlich etwas genauer betrachten, so stellen wir eine Struktur fest, die für alle Spalten gleich ist:

1. Setze die Dame in die erste Zeile, also `brett[spalte] = 0`.
2. Wird die neu gesetzte Dame `bedroht`, versuche es eine Zeile tiefer.
3. Steht die neu gesetzte Dame frei, beginne für die nächste Spalte wieder von vorne.

4. Hat die Suche dort keinen Erfolg, gehe wieder zum Schritt 2. Hatte die Suche Erfolg, sind wir fertig. Melde den „Erfolg“ als Ergebnis zurück.

5. Sind wir erfolglos bei der achten Spalte angekommen, stecken wir in einer „Sackgasse“. Melde den „Misserfolg“ als Ergebnis zurück.

Wir sehen nicht nur, dass die Suche nach der passenden Dame für alle Spalten gleich aufgebaut ist; wir erkennen vielmehr auch, dass sich die „Kommunikation“ zwischen den einzelnen Spaltensuchen auf ein einfaches **`true`** (= die Suche war erfolgreich) bzw. **`false`** (= die Suche war nicht erfolgreich) zurückführen lässt. Einen einzelnen **`boolean`**-Wert kann man wiederum sehr bequem von einer Methode zurückgeben lassen.
Wir definieren unsere Methode `setze` zuerst einmal so, als wollten wir die Lösung nur für eine ganz bestimmte Spalte suchen. Wir benötigen als Parameter also die Nummer der Spalte, in der wir suchen, und das Feld, in dem wir setzen sollen. Der Rückgabewert ist (wie oben gefordert) ein **`boolean`**-Wert:

```
public static boolean setze(int[] brett, int spalte) {
```

Wir wollen nun überlegen, wie wir obige fünf Schritte am besten in ein Java-Programm kleiden. „Beginne in der ersten Zeile“ und „versuche es eine Zeile tiefer“ – das klingt verdächtig nach einer Schleife! Wir formulieren also eine **`for`**-Schleife, die über die einzelnen Zeilennummern läuft:

```
for (int i=0; i < 8; i++) {
  brett[spalte] = i;          // Probiere jede Stelle aus
  if (bedroht(brett,spalte))  // Falls die Dame nicht frei steht
    continue; // versuche es an der naechsten Stelle
```

Nun haben wir innerhalb der Schleife also ein `i` gefunden, an der die Dame von keiner anderen bedroht wird. Was sagte obiger Algorithmus noch für diesen Fall? „Beginne wieder von vorne für die nächste Spalte“. Wir können also durch den rekursiven Aufruf `setze(brett,spalte+1)` bewirken, dass die gleichen Schritte auch für die nächste Spalte durchgeführt werden. Wir brauchen hierbei keine Kopie des Feldes zu übergeben, da die Methode ja nur nach rechts hin Veränderungen vornimmt, die uns eben nicht interessieren. Da wir natürlich auch wissen wollen, ob die Suche erfolgreich war, müssen wir das Ergebnis der Methode in einer Variablen sichern:

```
boolean success =
   setze(brett,spalte+1);
```

Ist der Inhalt der Variablen **`true`**, so haben wir Erfolg gehabt und können unsere Suche beenden (und damit auch die Methode):

```
if (success)
  return true;
```

Andernfalls müssen wir unsere Dame weiter verschieben, also die Schleife weiterhin ausführen. Sind wir am Ende der Schleife angekommen – d. h. es gibt keine weiteren Kombinationen mehr – stecken wir in einer Sackgasse. Der Rückgabewert ist somit **`false`**.

Wir haben nun alle Voraussetzungen, eine Lösung unseres Problems zu finden. Hiermit sind wir jedoch noch nicht fertig. Zwei Fragen bleiben (noch) unbeantwortet:

- Woran erkennen wir, ob wir eine Lösung gefunden haben oder noch weiterrechnen müssen?
- Terminiert unsere Methode? Haben wir uns auch wirklich keine Endlosschleife geschaffen?

Wir beschäftigen uns vorerst mit der ersten Frage, denn die zweite wird sich dann von selbst beantworten. Wir haben eine Lösung gefunden, wenn wir insgesamt acht Damen auf das Feld gesetzt haben, d. h. wenn gilt: `spalte==7` und `bedroht(brett,spalte)==false`. Wir könnten diese Abfrage in unsere `for-Schleife` einbauen, müssen es aber nicht. Falls wir nämlich acht Damen gesetzt haben, die sich gegenseitig nicht bedrohen, wird in unserer Schleife die Methode `setze` mit dem Parameter `spalte == 8` ein weiteres Mal aufgerufen. Es reicht also eine einzige Abfrage zu Beginn unserer Methode:

```
public static boolean setze(int[] brett, int spalte) {
  // Sind wir fertig?
  if (spalte == 8) {
    ausgabe(brett);
    return true;
  }

  // Suche die richtige Position fuer die neue Dame
  for (int i=0; i < 8; i++) {
    brett[spalte] = i;          // Probiere jede Stelle aus

    if (bedroht(brett,spalte)) // Falls die Dame nicht frei steht
      continue;  // versuche es an der naechsten Stelle

    boolean success =          // moeglicher Kandidat gefunden? --
      setze(brett,spalte+1);   // teste noch die folgenden Spalten

    if (success)               // falls es geklappt hat
      return true;             // Ende
  }

  // Wenn das Programm hier angekommen ist,
  // stecken wir in einer Sackgasse
  return false;
}
```

Unsere fertige Methode terminiert selbstverständlich, denn die **`for`**-Schleifen gehen alle nur bis `i==7` und die Methode ruft sich selbst maximal achtmal hintereinander auf. Selbst wenn für jede Kombination ein Aufruf stattfinden würde (was wegen der Methode `bedroht` nicht geschieht), würde die Methode also allerhöchstens $1+8^8 = 16777217$ Male aufgerufen – aber natürlich ist das bei weitem nicht so viel.

5.4.6 Die Lösung

Wir haben jetzt also eine rekursive Methode definiert, die acht Damen wie gefordert auf dem Schachbrett platziert und die Lösung ausgibt. Sind wir nun fertig? Haben wir nichts vergessen?
Vor lauter Methoden und rekursivem Aufruf haben wir noch nicht daran gedacht, unser Feld zu vereinbaren. Auch müssen wir die Berechnung natürlich irgendwie „anstoßen“, d. h. einen ersten Aufruf von `setze` mit leerem Schachbrett und `spalte=0` ausführen. Für diese Dinge soll die Hauptmethode zuständig sein:

```
public static void main(String[] args) {
  int[] feld = {0,0,0,0,0,0,0,0}; // Initialisiere das Spielfeld
  setze(feld,0); // Starte die Suche am linken Rand
}
```

Wir haben nun ein Programm geschrieben, das unser gestelltes Achtdamenproblem löst (natürlich nur, sofern wir keine Fehler gemacht haben). Wir speichern das Programm in einer Datei `Achtdamen.java` ab, übersetzen es und lassen es laufen. Wir erhalten folgendes Ergebnis:

Konsole

```
|D| | | | | | | |
| | | | | | |D| |
| | | | |D| | | |
| | | | | | | |D|
| |D| | | | | | |
| | | |D| | | | |
| | | | | |D| | |
| | |D| | | | | |
```

Man kann relativ leicht nachprüfen, dass obige Schachbrettkonstellation natürlich nicht die einzige mögliche Lösung ist (man muss das Brett hierzu lediglich um 90 Grad drehen). Wir wollten aber schließlich auch nur *eine* und nicht *alle* Lösungen. Diese zu erhalten ist Teil einer der folgenden Übungsaufgaben.
Hier noch einmal das gesamte Programm im Überblick:

```
1  public class Achtdamen {
2    /** Testet, ob eine der Damen eine andere schlagen kann. */
3    public static boolean bedroht(int[] brett, int spalte) {
4      // Teste als Erstes, ob eine Dame in derselben Zeile steht
5      for (int i=0; i < spalte; i++)
6        if (brett[i] == brett[spalte])
7          return true;
8
9      // Teste nun, ob in der oberen Diagonale eine Dame steht
10     for (int i = spalte-1, j = brett[spalte]-1; i >= 0; i--,j--)
11       if (brett[i] == j)
12         return true;
13
14     // Teste, ob in der unteren Diagonale eine Dame steht
15     for (int i = spalte-1, j = brett[spalte]+1; i >= 0; i--,j++)
```

```
16        if (brett[i] == j)
17          return true;
18
19      // Wenn das Programm hier angekommen ist, steht die Dame "frei"
20      return false;
21    }
22
23    /** Sucht rekursiv eine Loesung des Problems. */
24    public static boolean setze(int[] brett, int spalte) {
25      // Sind wir fertig?
26      if (spalte == 8) {
27        ausgabe(brett);
28        return true;
29      }
30
31      // Suche die richtige Position fuer die neue Dame
32      for (int i=0; i < 8; i++) {
33        brett[spalte] = i;          // Probiere jede Stelle aus
34        if (bedroht(brett,spalte)) // Falls die Dame nicht frei steht
35          continue; // versuche es an der naechsten Stelle
36        boolean success = // moeglicher Kandidat gefunden? --
37          setze(brett,spalte+1);   // teste die folgenden Spalten
38        if (success)               // falls es geklappt hat
39          return true;
40      }
41
42      // Wenn das Programm hier angekommen ist,
43      // stecken wir in einer Sackgasse
44      return false;
45    }
46
47    /** Gibt das Schachbrett auf dem Bildschirm aus. */
48    public static void ausgabe(int[] brett) {
49      for (int i=0; i < 8; i++) {      // Anzahl der Zeilen
50        for (int j=0; j < 8; j++)      // Anzahl der Spalten
51          System.out.print("|" + ((i == brett[j]) ? 'D' : ' '));
52        System.out.println("|");       // Zeilenende
53      }
54    }
55
56    /** Initialisiert das Schachbrett und ruft Methode "setze" auf */
57    public static void main(String[] args) {
58      int[] feld = {0,0,0,0,0,0,0,0}; // Initialisiere das Spielfeld
59      setze(feld,0); // Starte die Suche am linken Rand
60    }
61  }
```

5.4.7 Übungsaufgaben

Aufgabe 5.4

Modifizieren Sie die Methode `setze`, indem Sie die Zeile

```
setze(brett,spalte+1);   // teste noch die folgenden Spalten
```

durch die Zeile

```
setze(brett,++spalte);   // teste noch die folgenden Spalten
```

ersetzen. Liefert das Programm jetzt noch eine Lösung? Versuchen Sie, die Antwort *ohne* den Rechner zu finden.
Machen Sie die Ersetzung rückgängig, und verändern Sie das Programm so, dass es *alle* Lösungen des Problems findet. Ein kleiner Tipp: Sie müssen dazu nur eine einzige Programmzeile verändern.

5.5 Black Jack

5.5.1 Vorwissen aus dem Buch

- Vorwissen der vorigen Kapitel
- Abschnitt 5.1 (Felder) sowie
- Kapitel 6 (Methoden).

5.5.2 Aufgabenstellung

Von Las Vegas über Monte Carlo bis zum Casino in Baden-Baden gehört das Kartenspiel Black Jack zum Standardprogramm. Auch wenn sich die Regeln im Detail von Haus zu Haus unterscheiden, folgen sie doch alle dem folgenden Grundprinzip:
Ein oder mehrere Spieler spielen gegen die Bank (den Croupier) und versuchen, eine höhere Punktzahl zu erhalten als das Haus. Zu Anfang erhalten alle Spieler und der Croupier eine offen liegende Karte. Danach erhalten alle Spieler eine zweite offene, der Croupier eine verdeckt liegende Karte. Man versucht, die ideale Punktzahl von 21 Punkten zu erreichen. Hat man mit seinem Blatt diese überboten, so hat man verloren. Asse zählen 11 Punkte[1], sonstige Bilder 10 Punkte. Die anderen Karten zählen ihren aufgedruckten Wert.
Der Spieler bzw. die Spielerin kann vom Croupier weitere Karten fordern („bleiben“) oder sich mit seinem Blatt zufrieden geben („danke“). Er sollte versuchen, so nahe wie möglich an die 21 Punkte heranzukommen, darf die Grenze aber, wie gesagt, nicht überschreiten.
Sind alle Spieler fertig, kann auch der Croupier Karten nehmen. Er muss so lange Karten nehmen, wie er höchstens 16 Punkte hat. Hat er mehr als 16 Punkte, darf er keine weiteren Karten nehmen.
Hat ein Spieler oder eine Spielerin bereits mit den ersten beiden Karten 21 Punkte erreicht, bezeichnet man dies als „Black Jack“. In diesem Fall darf der Croupier keine weiteren Karten nehmen; er hat also auch nur zwei Karten auf der Hand.
Der Spieler bzw. die Spielerin gewinnt, wenn er bzw. sie nicht über 21 liegt und mehr Punkte als der Croupier hat.[2] Haben Spieler und Croupier die gleiche An-

[1] Es gibt auch Spielregeln, in denen das As nur einen Punkt zählt.
[2] Natürlich darf auch der Croupier nicht überbieten.

	Kreuz	Pik	Herz	Karo
zwei	0	13	26	39
drei	1	14	27	40
vier	2	15	28	41
fünf	3	16	29	42
sechs	4	17	30	43
sieben	5	18	31	44
acht	6	19	32	45
neun	7	20	33	46
zehn	8	21	34	47
Bube	9	22	35	48
Dame	10	23	36	49
König	11	24	37	50
As	12	25	38	51

Tabelle 5.1: Codierung von Spielkarten

zahl von Punkten, handelt es sich um ein Unentschieden („Egalité“).
Wir wollen auf dem Computer nun ein solches Black-Jack-Spiel für einen Spieler und Croupier realisieren.

5.5.3 Analyse des Problems

Black Jack ist eines der wenigen Kartenspiele, die keine intelligenten Handlungen vom Croupier erfordern. Er handelt nach festen Regeln; wir können seinen Part also ohne Schwierigkeiten vom Computer übernehmen lassen.
Für die Realisierung mit Java stellen sich jedoch verschiedene wichtige Fragen, die wir (ganz getreu unserer goldenen Regel der Planung) im Vorfeld überdenken müssen:

- Wie realisiert man eine Karte auf dem Computer?
- Wie realisiert man ein Kartenspiel auf dem Computer?
- Wie mischt man Karten?
- Wie verteilt man Karten?

Wir behandeln ein Kartenspiel mit vier Farben (Kreuz, Pik, Herz, Karo) und dreizehn Karten pro Farbe (zwei, drei, vier, fünf, sechs, sieben, acht, neun, zehn, Bube, Dame, König, As). Diese 52 Karten werden wir schlicht und ergreifend nummerieren.
Tabelle 5.1 zeigt, wie wir die 52 Karten auf ganze Zahlen abbilden. Diese Verbindung hat wichtige Auswirkungen auf unseren Umgang mit den Karten:

1. Farben und Werte von Karten hängen unmittelbar mit der Zahl 13 zusammen. So hat etwa ein Bube immer die Nummer

$$9 + 13 \cdot x,\ x \in \{0, 1, 2, 3\},$$

das heißt, unsere Karte `karte` ist genau dann ein Bube, wenn

```
karte % 13 == 9
```

gilt. Analog ist die Karte etwa von der Farbe Herz, wenn in $x = 2$ ist, also

```
karte / 13 == 2
```

ist. Wir können also aus der Kartennummer sowohl Farbe als auch Bild direkt ablesen.

2. Für die Bewertung von Karten lässt sich ein einfaches Kriterium erstellen:

   - Gilt für die Karte

     ```
     karte % 13 < 9
     ```

     so haben wir es mit keiner Bildkarte zu tun, das heißt, wir können den Wert direkt ablesen:

     ```
     wert = 2 + karte % 13;
     ```

   - Für ein As muss

     ```
     karte % 13 == 12
     ```

     gelten; wir können in diesem Fall also den Wert auf 11 setzen.
   - In allen anderen Fällen haben wir eine Bildkarte mit dem Wert 10.

   Wir wollen diesen Zusammenhang gleich in Form einer Methode festhalten:

```
public static int wert(int n) {
  int w=n%13;
  if (w<=8) // zwischen zwei und zehn
    return w+2;
  else
    if (w==12) // As
      return 11;
    else // sonstige Bildkarte
      return 10;
}
```

Da wir nun eine einzelne Spielkarte mit einem einfachen **`int`**-Wert gleichsetzen können, werden wir auch die weiteren Fragen relativ einfach beantworten können. Unsere Karten werden in einem Feld von ganzen Zahlen abgelegt, das wir wie in Abschnitt 5.5.4 beschrieben mischen. Die Ausgabe der einzelnen Karten wird in einem Zähler `pos` vermerkt, der die aktuelle Position im Kartenstapel markiert. Wir werden im Abschnitt 5.5.5 auch auf dieses Thema genauer eingehen.

5.5.4 Mischen eines Kartenspiels

Wir werden uns nun damit befassen, wie man ein Päckchen Karten auf dem Computer mischt. Zu diesem Zweck orientieren wir uns an der Realität und fragen uns, was den Vorgang des Mischens ausmacht?
Ziel des Mischens von Karten ist es hauptsächlich, dass sich jede Karte an jedem beliebigen Ort des Stapels befinden kann. Ein kleines Kind geht zu diesem Zweck nach einem einfachen Muster vor: es nimmt die oberste Karte des Stapels und schiebt sie an einer beliebigen Stelle in das Päckchen zurück.
Wir wollen es dem Kind gleichtun und definieren eine Methode

```
public static void mischen(int[] feld) {
```

zum Mischen eines Feldes von ganzen Zahlen (unserem Stapel). Wir wollen jeder Karte (also jedem Feldelement) die Chance geben, an beliebiger Stelle eingefügt zu werden. Aus diesem Grund gehen wir die einzelnen Feldelemente in einer Schleife durch:

```
  for (int i=0;i<feld.length;i++) {
```

Der Index `i` unserer Schleife steht für das aktuelle Element, das wir aus dem Stapel nehmen wollen. Mit Hilfe der Zufallsfunktion[3] `Math.random` bestimmen wir die neue Stelle, an der wir die Karte einfügen wollen:

```
    int j=(int)(feld.length*Math.random());
```

Nun müssen wir die Karten mit den Indizes `i` und `j` vertauschen:

```
    int dummy=feld[i];
    feld[i]=feld[j];
    feld[j]=dummy;
```

Nach Ablauf dieser Schleife ist das Feld gut durchmischt. Machen Sie sich an dieser Stelle noch einmal bewusst, dass wir aufgrund des Referenzcharakters von Feldern das vertauschte Array nicht über eine **`return`**-Anweisung zurückgeben müssen. Unsere komplette Methode sieht also wie folgt aus:

```
public static void mischen(int[] feld) {
  for (int i=0;i<feld.length;i++) {
    int j=(int)(feld.length*Math.random());
    int dummy=feld[i];
    feld[i]=feld[j];
    feld[j]=dummy;
  }
}
```

5.5.5 Die Pflichten des Gebers

Wir wollen uns nun damit beschäftigen, wie wir einen Kartenstapel auf dem Computer darstellen. Hierbei machen wir uns zuerst bewusst, dass in einem solchen

[3] Wir wollen hierbei nicht erläutern, wie diese Methode die „Zufallszahlen“ berechnet. Genau genommen werden hierbei keine wirklichen Zufallszahlen berechnet, sondern nur Pseudozufallszahlen, d. h. die ausgegebenen Zahlen werden durch ein einfaches arithmetisches Programm berechnet.

Stapel üblicherweise mehr als ein Päckchen verwendet wird, denn je mehr Kartenspiele der Croupier verwendet, desto schwerer fällt es dem Spieler, sich die im Stapel verbliebenen Karten zu merken und hierdurch Wahrscheinlichkeitsberechnungen anzustellen. Wir entwickeln also eine Methode

```
public static int[] schlitten(int n) {
```

zum Füllen eines Kartenschlittens mit insgesamt `n` Kartenspielen. Wir erzeugen ein Feld

```
int[] schlitten=new int[n*52];
```

und initialisieren es über zwei geschachtelte Schleifen:

```
for (int i=0;i<schlitten.length;i+=52)
  for (int j=0;j<52;j++)
    schlitten[i+j]=j;
```

Nun müssen wir das Feld lediglich noch mischen (die hierzu notwendige Methode haben wir bereits entwickelt) und können es dann zurückgeben. Wie aber realisieren wir das Austeilen von Karten durch den Geber? Zuerst entwerfen wir eine Klasse `Schlitten` wie folgt:

```
public static class Schlitten {
  public int[] karten; // Karten im Schlitten
  public int pos; // Position im Schlitten
}
```

Instanzen dieser Klasse stellen einen Schlitten mit Karten dar. Die Variable `pos` soll zu Anfang auf `0` gesetzt werden und speichert jeweils die Position der nächsten auszugebenden Karte im Schlitten. Wenn wir eine Karte aus dem Schlitten nehmen wollen, gehen wir wie folgt vor:

1. Wähle die aktuelle Karte `karten[pos]` als auszugebende Karte.
2. Erhöhe den Zähler `pos`

Auf diese Weise ersparen wir es uns, die einzelnen Karten aus dem Schlitten physikalisch zu entfernen, also das Feld manipulieren zu müssen. Der Schlitten ist leer, wenn `pos` an der Stelle `karten.length` angelangt ist. In diesem Fall können wir unseren Schlitten einfach neu „füllen“, indem wir das vorhandene Feld neu durchmischen:

```
public static int karte(Schlitten schlitten) {
  if (schlitten.pos==schlitten.karten.length) { // Schlitten leer
    System.out.println("\nSchlitten wird neu gefuellt...\n");
    mischen(schlitten.karten);
    schlitten.pos=0;
  } // Andernfalls gib die aktuelle Karte zurueck
  return schlitten.karten[schlitten.pos++];
}
```

Wir sind an dieser Stelle allerdings noch immer nicht fertig: Es reicht nicht aus, die Karte nur aus dem Schlitten zu ziehen. Der Spieler ist durch eine Bildschirmausgabe zu informieren, welche Karten ausgespielt werden. Hierzu definieren

wir zuerst eine Methode `name`, die aus der Kartennummer den Namen der Karte (z. B. Pik As) errechnet:

```
public static String name(int n) {
  String[] farben={"Kreuz","Pik","Herz","Karo"};
  String[] werte={
    "Zwei","Drei","Vier","Fuenf","Sechs","Sieben",
    "Acht","Neun","Zehn","Bube","Dame","Koenig",
    "As"
  };
  return farben[n/13]+" "+werte[n%13];
}
```

Die eigentliche Ausgabe betten wir nun in eine Methode `ausgabe` ein, die wir wie folgt formulieren:

```
/** Gib eine Karte an die Person p aus */
public static int ausgabe(String p,Schlitten s) {
  int karte=karte(s);
  System.out.println(p+" erhaelt "+name(karte)+
                     " (Wert="+wert(karte)+")");
  return wert(karte);
}
```

Rückgabewert der Methode ist hierbei eine ganze Zahl, die jedoch nicht die Nummer der Karte, sondern ihr Wert (vgl. 5.5.3) bezüglich der Spielregeln von Blackjack ist. Wir werden sehen, dass wir für unser Hauptprogramm keine weiteren Informationen benötigen.

5.5.6 Zum Hauptprogramm

Kommen wir nun zum eigentlichen Hauptprogramm. Zuerst initialisieren wir unseren Kartenschlitten:

```
int packs=
  IOTools.readInteger("Wie viele Paeckchen Karten "+
                      "sollen im Schlitten sein? ");
Schlitten schlitten=new Schlitten();// Erzeuge die Instanz
schlitten.karten=schlitten(packs);   // Erzeuge die Karten
schlitten.pos=0;                   // Aktuelle Position =0
```

Hierzu erfragen wir, wie viele Kartenpäckchen (`packs`) unser Schlitten fassen soll. Nun erzeugen wir mit dem **`new`**-Operator ein Objekt `schlitten`, dessen Feld wir mit der gleichnamigen Methode `schlitten` initialisieren. Den Zähler `pos` setzen wir wie geplant auf 0.
Das eigentliche Spiel realisieren wir nun in einer Schleife. Wir definieren eine **`boolean`**-Variable `nochEinSpiel`, die wir mit **`true`** initialisieren. Unsere Schleife führen wir so lange durch, wie eben diese Variable **`true`** ist. Innerhalb der Schleife benötigen wir zwei wichtige Variablen:

- eine Variable `sblatt`, die den Punktestand des Spielers in der aktuellen Runde sichert (Initialwert ist 0).

- eine Variable `cblatt`, in der der Punktstand des Croupiers steht (auch mit 0 initialisiert)

Wir beginnen nun damit, dem Spieler zwei und dem Croupier eine Karte zu geben:

```
// Gib Karte an Spieler aus
sblatt=ausgabe("Spieler",schlitten);
// Gib Karte an Croupier aus
cblatt=ausgabe("Croupier",schlitten);
// Gib Karte an Spieler aus
sblatt+=ausgabe("Spieler",schlitten);
```

Wir speichern hierbei nicht die einzelnen Karten, sondern nur die Summe ihrer Werte – dies ist vollkommen ausreichend, da wir im Folgenden nur noch die Summe der Punkte betrachten müssen.
An dieser Stelle müssen wir den ersten Sonderfall betrachten: Hat der Spieler einen Black Jack? Wenn ja (also `sblatt==21` gilt), darf der Croupier nur noch eine weitere Karte nehmen:

```
if (sblatt==21) {
  System.out.println("\nBLACKJACK!\n");
  cblatt+=ausgabe("Croupier",schlitten);
```

Aus dem Punktestand lässt sich nun der Ausgang des Spiels ablesen: Hat der Croupier weniger als 21 Punkte oder aber mehr (überboten), so gewinnt der Spieler. Andernfalls herrscht ein Gleichstand, also „Egalité“:

```
  if (cblatt<21 || cblatt>22)
    System.out.println("Spieler gewinnt!\n");
  else
    System.out.println("EGALITE");
```

Hat der Spieler bzw. die Spielerin keinen Black Jack, darf er bzw. sie neue Karten ordern. Wir formulieren dies in einer Schleife:

```
else { // keinen Black Jack
  // der Spieler darf neue Karten ordern
  while(true) {
    System.out.println(); // Leerzeile
    // Schaue, ob der Spieler bereits fertig ist
    if (sblatt==21) {
      System.out.println("Spieler hat 21 Punkte.");
      break;
    }
    if (sblatt>21) {
      System.out.println("Spieler liegt ueber 21 Punkte.");
      break;
    }
    // Lies den Benutzerwunsch ein
    IOTools.flush();
    char antwort=' ';
    while (antwort!='J' && antwort!='N')
      antwort=
        IOTools.readChar("Noch eine Karte (J/N) ?");
```

```
    // Ist der Benutzer zufrieden ?
    if (antwort=='N') {
      System.out.println("Spieler sagt: danke");
      break;
    }
    // Andernfalls erhaelt er noch eine Karte
    System.out.println("Spieler sagt: bleiben");
    sblatt+=ausgabe("Spieler",schlitten);
  }
```

Ist der Benutzer bzw. die Benutzerin fertig, wird also die Schleife verlassen, dann liegt es am Croupier, sich weitere Karten zu nehmen. Er muss so lange Karten nehmen, bis sein Punktestand größer als 16 ist:

```
// der Croupier muss nachziehen
if (sblatt<=21) { // Spieler hat nicht ueberboten
  System.out.println("\nCroupier ist am Zug:");
  while (cblatt<=16)
    cblatt+=ausgabe("Croupier",schlitten);
}
```

Anschließend ziehen wir Bilanz. Der Spieler bzw. die Spielerin hat gewonnen, wenn er bzw. sie weniger als 22 und mehr Punkte als der Croupier hat (oder der Croupier überboten hat). Der Croupier hat gewonnen, wenn er einen besseren Punktestand als der Spieler hat (oder dieser überboten hat). Andernfalls liegt ein Gleichstand vor:

```
if (sblatt>21 || (cblatt<=21 && cblatt>sblatt))
  System.out.println("Spieler verliert.");
else if (cblatt>21 || cblatt<sblatt)
  System.out.println("Spieler gewinnt.");
else
  System.out.println("EGALITE");
```

Damit ist das Spiel zuende, das Black Jack - Programm wurde auf dem Rechner realisiert. Wir müssen den Spieler bzw. die Spielerin nur noch fragen, ob er bzw. sie eine weitere Partie wünscht. Entsprechend wird die Variable `nochEinSpiel` angepasst:

```
char antwort=' ';
while (antwort!='J' && antwort!='N')
  antwort=
    IOTools.readChar("Noch ein Spiel (J/N) ?");
nochEinSpiel=(antwort=='J');
```

Unser Programm ist somit komplett – wir können es übersetzen und ausführen:

Konsole

```
Wie viele Paeckchen Karten sollen im Schlitten sein? 5
Spieler erhaelt Karo Neun (Wert=9)
Croupier erhaelt Kreuz Zwei (Wert=2)
Spieler erhaelt Kreuz Koenig (Wert=10)

Noch eine Karte (J/N) ?N
```

```
Spieler sagt: danke

Croupier ist am Zug:
Croupier erhaelt Pik Fuenf (Wert=5)
Croupier erhaelt Herz Sieben (Wert=7)
Croupier erhaelt Pik Acht (Wert=8)

Spieler hat 19 Punkte.
Croupier hat 22 Punkte.
Spieler gewinnt.

Noch ein Spiel (J/N) ?N
```

5.5.7 Das komplette Programm im Überblick

```
1  import Prog1Tools.IOTools;
2
3  /** Ein einfaches Blackjack-Spiel */
4  public class Blackjack {
5
6    /** Diese Klasse repraesentiert den Kartenschlitten */
7    public static class Schlitten {
8      public int[] karten; // Karten im Schlitten
9      public int pos; // Position im Schlitten
10   }
11
12   /** Berechnet aus der Kartennummer den Namen der Karte */
13   public static String name(int n) {
14     String[] farben={"Kreuz","Pik","Herz","Karo"};
15     String[] werte={
16       "Zwei","Drei","Vier","Fuenf","Sechs","Sieben",
17       "Acht","Neun","Zehn","Bube","Dame","Koenig",
18       "As"
19     };
20     return farben[n/13]+" "+werte[n%13];
21   }
22
23   /** Liefert aus der Kartennummer den Wert */
24   public static int wert(int n) {
25     int w=n%13;
26     if (w<=8) // zwischen zwei und zehn
27       return w+2;
28     else
29       if (w==12) // As
30         return 11;
31       else // sonstige Bildkarte
32         return 10;
33   }
34
35   /** Mische ein Feld von ganzen Zahlen */
36   public static void mischen(int[] feld) {
37     for (int i=0;i<feld.length;i++) {
```

```
38        int j=(int)(feld.length*Math.random());
39        int dummy=feld[i];
40        feld[i]=feld[j];
41        feld[j]=dummy;
42      }
43    }
44
45    /** Erzeugt einen Schlitten aus n Kartenspielen */
46    public static int[] schlitten(int n) {
47      // Initialisiere das Feld
48      int[] schlitten=new int[n*52];
49      for (int i=0;i<schlitten.length;i+=52)
50        for (int j=0;j<52;j++)
51          schlitten[i+j]=j;
52      // Mische das Feld
53      mischen(schlitten);
54      // Gib das gemischte Feld zurueck
55      return schlitten;
56    }
57
58    /** Ziehe eine Karte aus dem Schlitten */
59    public static int karte(Schlitten schlitten) {
60      if (schlitten.pos==schlitten.karten.length) { // Schlitten leer
61        System.out.println("\nSchlitten wird neu gefuellt...\n");
62        mischen(schlitten.karten);
63        schlitten.pos=0;
64      } // Andernfalls gib die aktuelle Karte zurueck
65      return schlitten.karten[schlitten.pos++];
66    }
67
68    /** Gib eine Karte an die Person p aus */
69    public static int ausgabe(String p,Schlitten s) {
70      int karte=karte(s);
71      System.out.println(p+" erhaelt "+name(karte)+
72                         " (Wert="+wert(karte)+")");
73      return wert(karte);
74    }
75
76    /** Hauptprogramm */
77    public static void main(String[] args) {
78      // Zuerst initialisiere den Schlitten
79      int packs=
80        IOTools.readInteger("Wie viele Paeckchen Karten "+
81                            "sollen im Schlitten sein? ");
82      Schlitten schlitten=new Schlitten();// Erzeuge die Instanz
83      schlitten.karten=schlitten(packs);   // Erzeuge die Karten
84      schlitten.pos=0;                   // Aktuelle Position =0
85      // Weitere benoetigte Variablen
86      boolean nochEinSpiel=true;
87      // Jetzt beginnt das eigentliche Spiel
88      while (nochEinSpiel) {
89        // benoetigte Variablen
90        int sblatt=0; // Wert des Blattes des Spielers
91        int cblatt=0; // Wert des Blattes des Croupiers
92        // Gib Karte an Spieler aus
```

```
93        sblatt=ausgabe("Spieler",schlitten);
94        // Gib Karte an Croupier aus
95        cblatt=ausgabe("Croupier",schlitten);
96        // Gib Karte an Spieler aus
97        sblatt+=ausgabe("Spieler",schlitten);
98        // Teste, ob der Spieler Blackjack hat
99        if (sblatt==21) {
100         System.out.println("\nBLACKJACK!\n");
101         cblatt+=ausgabe("Croupier",schlitten);
102         if (cblatt<21 || cblatt>22)
103           System.out.println("Spieler gewinnt!\n");
104         else
105           System.out.println("EGALITE");
106       }
107       else { // keinen Black Jack
108         // der Spieler darf neue Karten ordern
109         while(true) {
110           System.out.println(); // Leerzeile
111           // Schaue, ob der Spieler bereits fertig ist
112           if (sblatt==21) {
113             System.out.println("Spieler hat 21 Punkte.");
114             break;
115           }
116           if (sblatt>21) {
117             System.out.println("Spieler liegt ueber 21 Punkte.");
118             break;
119           }
120           // Lies den Benutzerwunsch ein
121           IOTools.flush();
122           char antwort=' ';
123           while (antwort!='J' && antwort!='N')
124             antwort=
125               IOTools.readChar("Noch eine Karte (J/N) ?");
126           // Ist der Benutzer zufrieden ?
127           if (antwort=='N') {
128             System.out.println("Spieler sagt: danke");
129             break;
130           }
131           // Andernfalls erhaelt er noch eine Karte
132           System.out.println("Spieler sagt: bleiben");
133           sblatt+=ausgabe("Spieler",schlitten);
134         }
135         // der Croupier muss nachziehen
136         if (sblatt<=21) { // Spieler hat nicht ueberboten
137           System.out.println("\nCroupier ist am Zug:");
138           while (cblatt<=16)
139             cblatt+=ausgabe("Croupier",schlitten);
140         }
141         // Jetzt wird Bilanz gezogen
142         System.out.println();
143         System.out.println("Spieler hat "+sblatt+" Punkte.");
144         System.out.println("Croupier hat "+cblatt+" Punkte.");
145         if (sblatt>21 || (cblatt<=21 && cblatt>sblatt))
146           System.out.println("Spieler verliert.");
147         else if (cblatt>21 || cblatt<sblatt)
```

```
148             System.out.println("Spieler gewinnt.");
149           else
150             System.out.println("EGALITE");
151           System.out.println();
152         }
153         // Will der Benutzer noch ein Spiel?
154         char antwort=' ';
155         while (antwort!='J' && antwort!='N')
156           antwort=
157             IOTools.readChar("Noch ein Spiel (J/N) ?");
158         nochEinSpiel=(antwort=='J');
159         System.out.println();
160       }
161     }
162   }
```

5.5.8 Übungsaufgaben

Aufgabe 5.5

Erweitern Sie das Programm so, dass auch um Geld gespielt werden kann. Hierbei gelten folgende Regeln:
Bevor die erste Karte ausgeteilt wird, kann der Spieler seinen Einsatz machen. Gewinnt er gleich bei der zweiten Karte (Black Jack), so erhält er die Hälfte seines Einsatzes als Gewinn (also 3 GE[4] Rückzahlung bei 2 GE Einsatz). Gewinnt er im späteren Verlauf des Spiels, erhält er seinen Einsatz als Gewinn (sprich: 4 GE Rückzahlung bei 2 GE Einsatz).
Verliert der Spieler das Spiel, so verliert er seinen Einsatz. Bei Egalité erhält er seinen Einsatz ohne Gewinn zurück.
Zu Anfang des Spiels erhält der Spieler einen gewissen Kontostand (etwa 100 GE). Das Spiel ist beendet, wenn der Spieler aussteigt oder Pleite geht, also sein Kontostand auf 0 GE gesunken ist.
Nach all der grauen Theorie soll nun wieder einmal ein Kapitel mit etwas konkreteren Beispielen folgen. Wir werden anhand von drei Beispielen den Umgang mit Klassen erproben und hierbei mehr oder minder komplexe Probleme in Java lösen.

5.6 Streng geheim

5.6.1 Vorwissen aus dem Buch

- Kapitel 7 (Die objektorientierte Philosophie)
- Kapitel 8 (Der grundlegende Umgang mit Klassen) sowie
- Kapitel 9 (Vererbung und Polymorphismus).

[4] GE ist eine Geldeinheit, also z. B. DM, Dollar oder Euro.

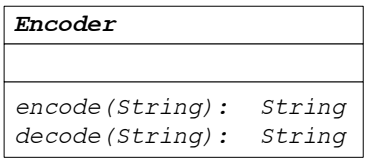


Abbildung 5.1: Die abstrakte Klasse `Encoder`

5.6.2 Aufgabenstellung

Stellen Sie sich vor, Sie arbeiten für einen kleinen, aber exklusiven Club von Geheimagenten. Sie trinken Ihren Martini geschüttelt (nicht gerührt) und sind auf der ganzen Welt „geschäftlich“ unterwegs.
In den letzten Jahren hat sich die Arbeitswelt eines Geheimagenten leider drastisch verändert. Vorbei sind die Zeiten, an denen Sie geheime Nachrichten in toten Briefkästen unter einsamen Parkbänken fanden. Vorbei auch die Zeiten, in denen Sie so wundervolle Erkennungssätze wie „in einem warmen Sommer flogen die Schwalben stets tief“ auswendig daherbeten mussten. Heutzutage verschicken Sie Ihre Geheimbotschaften per E-Mail und geben sich lediglich per Retina-Scan, PIN-Nummer und digitaler Signatur zu erkennen.
Leider haben Sie in letzter Zeit Probleme mit der Sicherheit Ihrer Post festgestellt. Schon mehrere Nachrichten wurden abgefangen, entschlüsselt und gegen Sie verwendet. Die explodierende Cocktailkirsche fanden Sie ja noch ganz lustig, aber als man Ihren Goldhamster neulich bei seinem täglichen Spaziergang von einem Kampfhund verfolgen ließ, war das Maß voll!
Aus diesem Grund haben Sie beschlossen, Ihr eigenes Sicherheitssystem zu entwickeln. Sie wollen ein Programm schreiben, das eine Nachricht ver- und entschlüsseln kann. Das Programm soll hierbei so flexibel sein, dass Sie die konkrete Form der Verschlüsselung jederzeit austauschen können.

5.6.3 Analyse des Problems

Wir beginnen damit, die Aktion des Ver- und Entschlüsselns in einer Klasse zu modellieren. Wie bei der Klasse `Waehrung` beginnen wir auch hier wieder mit einer abstrakten Klasse, die lediglich die Schnittstelle für ihre Subklassen vorgibt. Abbildung 5.1 zeigt den Entwurf unserer Klasse, die wir `Encoder` nennen wollen. Unsere Klasse verfügt über zwei Methoden, `encode` und `decode` genannt, die einen beliebigen `String` ver- und entschlüsseln können. Um dies zu demonstrieren, werden wir in späteren Tests die folgende Methode `demo` verwenden:

```
/** Liest eine Textzeile ein, ver- und entschluesselt
 diese */
public static void demo(Encoder enc) {
  // Lies die zu verschluesselnde Zeile ein
  String line=IOTools.readLine("Zu verschluesselnde Zeile: ");
  // Verschluessele die Zeile
```

```
        String encoded=enc.encode(line);
        System.out.println("Verschluesselt: " + encoded);
        // Entschluessele die Zeile
        String decoded=enc.decode(encoded);
        System.out.println("Entschluesselt: " + decoded);
        // Test: entsprechen sich Original und Kopie ?
        if (line.equals(decoded)) // Sind die beiden Strings gleich?
          System.out.println("VERSCHLUESSELUNG ERFOLGREICH!");
        else
          System.out.println("PROGRAMMFEHLER!");
      }
```

Unsere Methode liest eine Textzeile von der Tastatur ein, die es zu verschlüsseln gilt. Zur Verschlüsselung verwendet sie ein `Encoder`-Objekt, das ihr als Argument beim Methodenaufruf übergeben wurde. Mit Hilfe der Methode `encode` des Objektes kann sie den Text dann verschlüsseln und auf dem Bildschirm ausgeben.
Anschließend soll getestet werden, ob die Verschlüsselung erfolgreich war. Zu diesem Zweck wird der `String` mit Hilfe der Methode `decode` wieder rückübersetzt und auf dem Bildschirm ausgegeben. Anschließend verwendet das Programm die Methode `equals`, um den entschlüsselten `String` mit dem Original zu vergleichen. Stimmen beide Texte überein, so war die Verschlüsselung erfolgreich.
Wir werden uns nun der Aufgabe widmen, verschiedene Verschlüsselungsalgorithmen in Java zu realisieren. Da dieses Buch natürlich keine Kenntnisse in Kryptographie voraussetzen kann, bleiben wir bei relativ einfachen und verständlichen Methoden. Werfen wir jedoch zuerst noch einen Blick auf unsere abstrakte Superklasse `Encoder`:

```
1  /** Diese Klasse symbolisiert eine beliebige
2      Verschluesselung */
3  public abstract class Encoder {
4
5    /** Verschluesselt einen String */
6    public abstract String encode(String s);
7
8    /** Entschluesselt einen String anhand eines
9        gegebenen Schluessels */
10   public abstract String decode(String s);
11
12 }
```

5.6.4 Verschlüsselung durch Aufblähen

Beginnen wir mit einer einfachen Methode, wie Sie sie vielleicht schon einmal in alten Detektivromanen oder Kindergeschichten gelesen haben. Wir verschlüsseln einen Text, indem wir jeweils zwei Buchstaben einer Nachricht miteinander vertauschen und in deren Mitte einen zufälligen anderen Buchstaben einfügen. Da unsere Nachricht hierbei durch die Zufallsbuchstaben „aufgebläht“ wird, nennen wir unsere Realisierung in Java einfach `Inflater`:

```
/** Verschluesselt einen String, indem jeweils
    zwei Zeichen paarweise vertauscht werden und
    zwischen diese ein weiteres zufaelliges Zeichen
    einfuegt wird. */
public class Inflater extends Encoder {

  /** Verschluesselt einen String */
  public String encode(String s) {
    // Wandle den String in ein char-Array um
    // (toCharArray ist Methode der Klasse String)
    char[] c = s.toCharArray();
    // Initialisiere den String res,
    // der das Ergebnis enthalten soll
    String res="";
    // Wende den Algorithmus immer auf zwei Zeichen an
    for (int i=0;i<c.length-1;i=i+2) {
      char c1=c[i];   // das erste Zeichen
      char c2=c[i+1]; // das zweite Zeichen
      // Bestimme ein drittes, zufaelliges Zeichen
      char c3=(char)('a'+26*Math.random());
      // tausche c2,c1 und fuege c3 dazwischen
      res=res+c2+c3+c1;
    }
    // Falls die Laenge des Feldes ungerade war,
    // haben wir ein Zeichen uebersehen
    if (c.length%2!=0) {
      // Dieses Zeichen muessen wir noch hinzufuegen
      res=res+c[c.length-1];
    }
    // Gib das Ergebnis zurueck
    return res;
  }

  /** Entschluesselt einen String */
  public String decode(String s) {
    // Wandle den String in ein char-Array um
    char[] c=s.toCharArray();
    // Initialisiere den String res,
    // der das Ergebnis enthalten soll
    String res="";
    // Wende den Algorithmus immer auf drei Zeichen an
    for (int i=0;i<c.length-2;i=i+3) {
      // zuerst das Zeichen c1, das ja an Stelle 3 steht
      res=res+c[i+2];
      // nun das    Zeichen c2
      res=res+c[i];
      // das Zeichen c3 faellt weg!
    }
    // Teste, ob ein Zeichen uebersehen wurde
    if (c.length % 3 != 0) {
      res=res + c[c.length - 1];
    }
    // Gib das Ergebnis zurueck
    return res;
  }
```

```
56
57  }
```

Gehen wir nun die wichtigsten Merkmale unserer neuen Klasse durch. Unsere Klasse besitzt keinerlei Instanzvariablen, da unser Algorithmus derlei Dinge nicht benötigt. Da wir also keine speziellen Werte initialisieren müssen, brauchen wir uns auch um Konstruktoren keine Gedanken zu machen – der vom Compiler eingefügte Standardkonstruktor reicht vollkommen aus!
Wir machen es uns also lediglich zur Aufgabe, die abstrakten Methoden `decode` und `encode` unserer Superklasse zu überschreiben. Hierbei beginnen wir mit der Methode `encode` in Zeile 8.
Da wir innerhalb der Methode nicht mit der kompletten Zeichenkette, sondern mit einzelnen Buchstaben arbeiten müssen, können wir mit dem Datentyp `String` leider wenig anfangen. Glücklicherweise besitzt die Klasse `String` jedoch eine Instanzmethode namens `toCharArray`. Diese Methode wandelt das Stringobjekt in ein Feld von **`char`**-Variablen um, das wir im Programm durch die Variable `c` referenzieren (Zeile 11).
Nun können wir statt mit einem String mit einem Feld von einzelnen Zeichen arbeiten. Zuerst erzeugen wir einen leeren `String res`, in dem wir unser verschlüsseltes Ergebnis erzeugen (Zeile 14). Anschließend gehen wir in einer Schleife (Zeile 16) durch die einzelnen Komponenten des Feldes – also die Buchstaben. Da wir jeweils ein Buchstabenpaar betrachten, erhöhen wir unseren Zähler `i` in Zweierschritten.
Innerhalb unserer Schleife bezeichnen wir die Buchstaben des aktuell betrachteten Paares mit `c1` und `c2` (Zeile 17 und 18). Ein drittes Zeichen `c3`, das wir später zwischen die beiden Zeichen einfügen wollen, bestimmen wir in Zeile 20 rein zufällig. Nachdem wir diese drei Zeichen festgelegt haben, müssen wir sie nur noch in vertauschter Reihenfolge an unseren Ergebnisstring anhängen (Zeile 22).
Nach dieser relativ simplen Schleife wären wir eigentlich bereits fertig – wenn wir nicht einen wichtigen Sonderfall übersehen hätten. Angenommen, wir wollten das Wort „Hallo“ verschlüsseln, das aus exakt fünf Zeichen besteht. In diesem Fall hätten wir die Buchstabenpaare „Ha“ und „ll“, die in der Schleife bearbeitet werden können. Nichtsdestotrotz darf das letzte Zeichen („o“) nicht unter den Tisch fallen!
Zu diesem Zweck prüfen wir in Zeile 26, ob die Länge unserer Zeichen ungerade war. Trifft dieser Fall zu, so müssen wir an unser Ergebnis `res` das letzte verbliebene Zeichen anhängen (Zeile 28). Erst anschließend können wir das Resultat zurückgeben.
In der Dekodierungsphase (`decode`) müssen wir den Vorgang der Verschlüsselung nun wieder rückgängig machen. Zu diesem Zweck wandeln wir den übergebenen `String` mit Hilfe der Methode `toCharArray` wieder in ein Feld von einzelnen Zeichen um (Zeile 37). In einer anschließenden Schleife betrachten wir jeweils ein *Tripel* von Zeichen (also drei Stück). Wir fügen erst das dritte und dann das erste dieser Zeichen an unseren Ergebnisstring an (Zeile 44 und 46). Das mittlere Zeichen, das wir bei der Verschlüsselung zufällig eingefügt haben, fällt hier-

bei unter den Tisch.
Natürlich müssen wir auch in dieser Methode überprüfen, ob wir wegen eines Sonderfalls ein Zeichen übersehen haben. Da wir in diesem Fall Zahlentripel statt -paare betrachten, testen wir hierzu einfach, ob die Länge unseres Feldes durch 3 teilbar ist (Zeile 50).
Natürlich ist diese Methode der Verschlüsselung nicht besonders effizient. Durch simples Ausprobieren kann selbst ein Kind innerhalb kürzester Zeit hinter den Trick kommen, mit dem wir unsere Nachricht zu schützen versuchen. Wir werden deshalb im nächsten Abschnitt eine etwas interessantere Form der Verschlüsselung realisieren.

5.6.5 XOR-Verschlüsselung

Eines der einfachsten Verfahren aus der Verschlüsselungstechnik ist die so genannte XOR-Verschlüsselung. Hierbei werden die einzelnen zu verschlüsselnden Zeichen als binäre Zahlenreihen aufgefasst. Jedes Zeichen wird mit einem so genannten **Schlüssel** verknüpft. Hierbei handelt es sich eben um eine weitere binäre Zahlenreihe, die dem Benutzer bzw. der Benutzerin bekannt ist. Zeichen und Schlüssel werden über die binäre Operation **exklusives Oder** (englisch: „exclusive or“ oder einfach XOR) verknüpft. In Java ist dies der Dach-Operator ^.
Werden alle Zeichen (als binäre Zahlen aufgefasst) mit dem Schlüssel verknüpft, so ergibt sich eine Folge von neuen Zahlen, die auf den ersten Blick mit den originalen Zahlen nichts mehr zu tun haben. Erst eine weitere Verknüpfung mit dem Schlüssel ergibt wieder die originale Zahlen- bzw. Zeichenfolge.
Die folgende Klasse realisiert die XOR-Verschlüsselung:

```
1  /** Diese Klasse realisiert die so genannte XOR-Verschluesselung */
2  public class XorEncoder extends Encoder {
3
4    /** Hier wird der geheime Schluessel abgespeichert */
5    private int key;
6
7    /** Konstruktor. Dem Objekt wird der
8        geheime Schluessel uebergeben */
9    public XorEncoder(int key) {
10     this.key = key;
11   }
12
13   /** Verschluesselt einen String anhand
14       eines gegebenen Schluessels */
15   public String encode(String s) {
16     // Wandle den String in ein char-Array um
17     // (toCharArray ist Methode der Klasse String)
18     char[] c = s.toCharArray();
19     // Wende auf die einzelnen Zeichen die XOR-Verschluesselung an
20     for (int i=0;i<c.length;i++)
21       c[i]=(char)(c[i]^key);
22     // Gib das verschluesselte Feld als Array zurueck
23     return new String(c);
24   }
```

```
25
26    /** Entschluesselt einen String anhand eines
27        gegebenen Schluessels */
28    public String decode(String s) {
29      // Bei der einfachen XOR-Verschluesselung
30      // sind Ver- und Entschluesselung identisch.
31      // Wir koennen also die Methode encode verwenden
32      return encode(s);
33    }
34
35  }
```

Wir wollen nun die Realisierung der Klasse in einzelnen Schritten durchgehen:

1. Um den von uns verwendeten Schlüssel speichern zu können, haben wir eine private Instanzvariable `key` definiert (Zeile 5). Die Variable ist hierbei vom Typ **`int`**.

2. Im anschließend definierten Konstruktor (Zeile 9 bis 11) müssen wir den Schlüssel auf einen bestimmten Wert setzen. Hierzu übergeben wir einen Schlüssel als Argument, den wir in Zeile 10 in unsere Instanzvariable schreiben.

3. Die eigentliche Verschlüsselung ist nun noch einfacher als in unserer `Inflater`-Klasse. Wir wandeln unseren String wieder in ein Feld von Zeichen um (Zeile 18), das wir in einer Schleife durchlaufen (Zeile 20). Innerhalb der Schleife verknüpfen wir jeden einzelnen Feldeintrag durch ein exklusives Oder mit dem Schlüssel (Zeile 21). An Ende wandeln wir das resultierende Feld lediglich wieder in einen String um und geben diesen als Ergebnis zurück (Zeile 23). Hierbei nutzen wir den Umstand, dass die Klasse `String` einen Konstruktor besitzt, der eben dies leistet.

4. Die Entschlüsselung einer Zeichenkette geschieht wieder durch eine Verknüpfung mit dem Schlüssel. Wir müssen also lediglich innerhalb der Methode `decode` einen Aufruf der Methode `encode` realisieren (Zeile 32).

Die Verschlüsselung durch exklusives Oder ist ein wenig sicherer als der Inflater-Algorithmus. Für einen professionellen Hacker stellt er natürlich kein Problem dar, da die Zahl der möglichen Schlüssel sehr gering ist (insbesondere, wenn er zur Entschlüsselung einen Computer einsetzt). Da dieses Buch aber keinen Lehrgang in Kryptographie darstellen soll, mag uns dieser Algorithmus genügen.

5.6.6 Ein einfacher Test

Wir wollen jetzt unsere Methode `demo` verwenden, um die Funktionalität unserer Verschlüsselungsklassen zu testen. Da sich beide Klassen von der allgemeinen `Encoder`-Klasse ableiten (vgl. Abbildung 5.2), ist uns dies auch problemlos möglich. Wir entwerfen ein einfaches Testprogramm, das unseren Zwecken genügt:

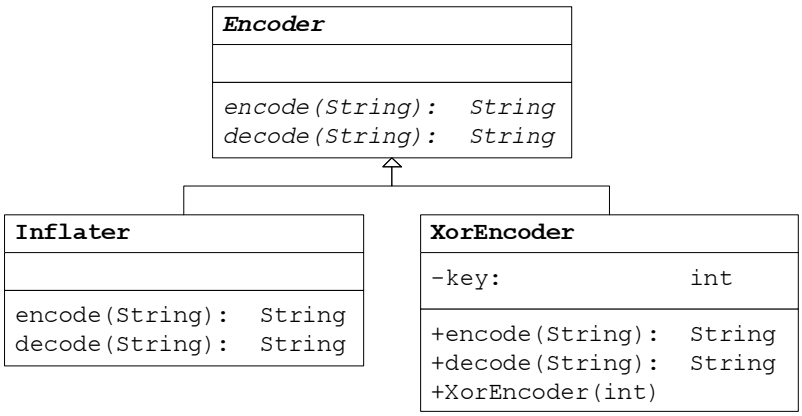


Abbildung 5.2: Konkrete Realisierungen der Klasse `Encoder`

```
1  import Prog1Tools.*;
2
3  /** Ein einfaches Demonstrationsprogramm */
4  public class CodingDemo {
5
6    /** Liest eine Textzeile ein, ver- und entschluesselt
7     diese */
8    public static void demo(Encoder enc) {
9      // Lies die zu verschluesselnde Zeile ein
10     String line=IOTools.readLine("Zu verschluesselnde Zeile: ");
11     // Verschluessele die Zeile
12     String encoded=enc.encode(line);
13     System.out.println("Verschluesselt: " + encoded);
14     // Entschluessele die Zeile
15     String decoded=enc.decode(encoded);
16     System.out.println("Entschluesselt: " + decoded);
17     // Test: entsprechen sich Original und Kopie ?
18     if (line.equals(decoded)) // Sind die beiden Strings gleich?
19       System.out.println("VERSCHLUESSELUNG ERFOLGREICH!");
20     else
21       System.out.println("PROGRAMMFEHLER!");
22   }
23
24   /** Die Main-Methode liest einen Schluessel ein, baut ein
25    Verschluesselungsobjekt auf und ruft die demo-Routine auf */
26   public static void main(String[] args) {
27     // Wir suchen uns eine zu verwendende Codierung aus
28     System.out.println("Codierungs-Demo                    \n" +
29                        "==================================\n" +
30                        "1 = Demo mit \"Inflater\"-Algorithmus\n" +
31                        "2 = Demo mit XOR-Verschluesselung  \n" +
32                        "==================================\n" +
33                        "\n"
34                       );
35     int auswahl = IOTools.readInteger("Ihre Wahl:");
```

```
36      // Nun durchlaufen wir das eigentliche Demo-Programm
37      switch(auswahl) {
38        case 1:
39          // Der Inflater kommt ohne einen Schluessel aus
40          demo(new Inflater());
41          break;
42        case 2:
43          // In diesem Fall brauchen wir noch einen Schluessel
44          int key=IOTools.readInteger("Bitte Schluessel eingeben: ");
45          Encoder enc = new XorEncoder(key);
46          demo(enc);
47          break;
48        default:
49          System.out.println("Ungueltige Auswahl!");
50      }
51    }
52  }
```

Die Methode `demo` ist uns bereits bekannt und muss an dieser Stelle nicht weiter erläutert werden: sie liest einen Text von der Tastatur ein, verschlüsselt und entschlüsselt ihn wieder. Aufgabe der Methode `main` ist es lediglich, den Benutzer bzw. die Benutzerin zwischen den verschiedenen Verschlüsselungsalgorithmen wählen zu lassen. Im Fall der Verschlüsselung durch Aufblähen wird hierzu ein `Inflater`-Objekt erzeugt (Zeile 40), im Falle der XOR-Verschlüsselung verwenden wir eine Instanz der Klasse `XorEncoder` (Zeile 45 und 46). In letzterem Fall lesen wir zuvor noch den zu verwendenden Schlüssel von der Tastatur ein (Zeile 44).
Wir übersetzen unser Programm und lassen es mit den verschiedenen Verschlüsselungsalgorithmen ablaufen. Wir beginnen mit der Klasse `Inflater`:

Konsole

```
Codierungs-Demo
===================================
1 = Demo mit "Inflater"-Algorithmus
2 = Demo mit XOR-Verschluesselung
===================================

Ihre Wahl:1
Zu verschluesselnde Zeile: Programmieren macht Spass
Verschluesselt: rsPgaoazrmumeyietr dnaomhoc btpvSsxas
Entschluesselt: Programmieren macht Spass
VERSCHLUESSELUNG ERFOLGREICH!
```

Wie wir sehen, war die Verschlüsselung unserer Zeile erfolgreich. Das gilt auch für die Verschlüsselung mit exklusivem Oder:

Konsole

```
Codierungs-Demo
===================================
1 = Demo mit "Inflater"-Algorithmus
2 = Demo mit XOR-Verschluesselung
```

```
==================================

Ihre Wahl:2
Bitte Schluessel eingeben: 1
Zu verschluesselnde Zeile: Programmieren macht Spass
Verschluesselt: Qsnfs`llhdsdo!l`biu!Rq`rr
Entschluesselt: Programmieren macht Spass
VERSCHLUESSELUNG ERFOLGREICH!
```

Machen Sie sich selbst ein Bild davon, welche der beiden Verschlüsselungen mit bloßem Auge leichter zu „knacken“ ist.

5.6.7 Übungsaufgaben

Aufgabe 5.6

Schreiben Sie eine Klasse `XorInflater`, die die Verschlüsselungen durch Aufblähen und mit exklusivem Oder miteinander kombiniert: die Nachricht soll zuerst durch den `Inflater`-Algorithmus aufgebläht und anschließend mit exklusivem Oder verschlüsselt werden. Hierbei soll der `XorInflater` natürlich auch eine Subklasse von `Encoder` sein. Versuchen Sie, die bereits existierenden Klassen so gut wie möglich wiederzuverwerten (siehe Abschnitt 4.3 in dieser Erweiterung für Inspiration).
Erweitern Sie die `main`-Methode der Klasse `CodingDemo` so, dass eine Verschlüsselung durch den `XorInflater` möglich ist.

5.7 Game of Life

5.7.1 Vorwissen aus dem Buch

- Kapitel 7 (Die objektorientierte Philosophie)
- Kapitel 8 (Der grundlegende Umgang mit Klassen) sowie
- Kapitel 9 (Vererbung und Polymorphismus).

5.7.2 Aufgabenstellung

Wir züchten in einer Petrischale eine Kolonie von Zellen. Unsere Zellen seien der Einfachheit halber wie auf einem Schachbrett angeordnet. Zwei Zellen, die senkrecht, waagerecht oder diagonal aneinander grenzen, nennt man *benachbart*. Eine Zelle hat also bis zu acht Nachbarn.
Wir unterscheiden in unserer Petrischale zwischen lebenden und toten Zellen. Lebende Zellen werden durch einen Kreis symbolisiert (vgl. Abbildung 5.3). Die Anzahl der Nachbarn einer Zelle, die lebendig sind, bezeichnen wir als die *Zahl der lebenden Nachbarn*.

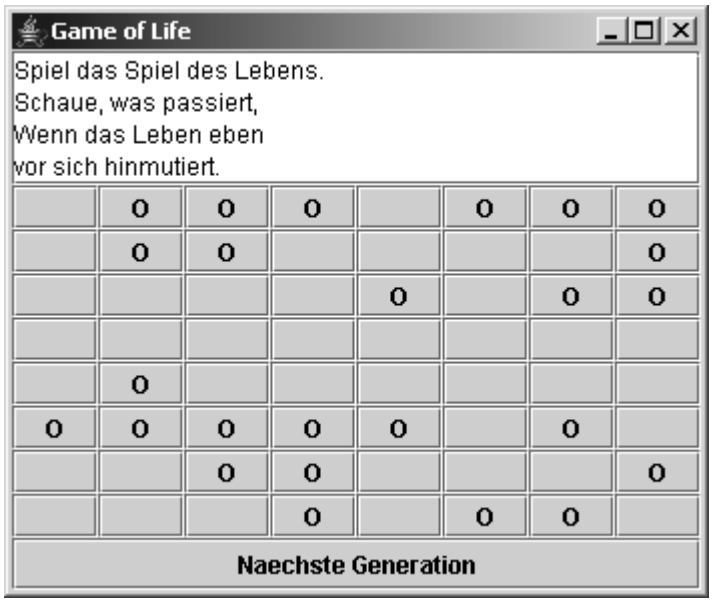


Abbildung 5.3: Game of Life

In jeder neuen Generation entscheidet sich für die einzelnen Zellen neu, ob sie lebendig oder tot sind. Entscheidungskriterium hierfür ist eben die Zahl der lebenden benachbarten Zellen:

- Eine Zelle wird (unabhängig von ihrem derzeitigen Zustand) in der nächsten Generation *tot* sein, wenn sie in der jetzigen Generation weniger als zwei oder mehr als drei lebende Nachbarn besitzt.
- Eine Zelle mit genau zwei lebenden Nachbarn ändert ihren Zustand nicht.
- Eine Zelle mit genau drei lebenden Nachbarn wird sich in der nächsten Generation im Zustand *lebendig* befinden.

Hierbei finden diese Zustandsänderungen alle auf einen Schlag statt, das heißt, der Zustand einer Zelle der neuen Generation wird nur von Zellen aus der alten Generation beeinflusst.
Unsere Aufgabe ist es nun, die Entwicklung der Zellen auf dem Computer zu simulieren, so wie es in Abbildung 5.3 schon vorweggenommen wurde. Hierbei soll der Spieler (sprich der „Zellforscher") interaktiv in das Geschehen eingreifen können, d. h. er soll auch in der Lage sein, eine der Zellen manuell vom Status „lebend" auf „tot" zu setzen (und umgekehrt).

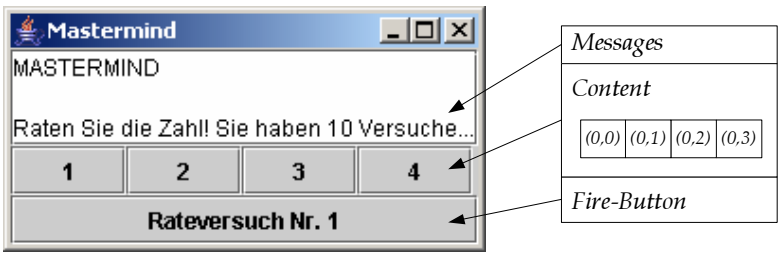


Abbildung 5.4: Die grafische Oberfläche und das zugehörige Modell

5.7.3 Die Klassen `GameModel` und `GameEngine`

Auch wenn wir uns in späteren Kapiteln mit grafischer Programmierung befassen, ist es jetzt noch ein wenig zu früh. Für den Moment gehen wir deshalb einfach davon aus, dass es genug andere Menschen gibt, die dieses schon vor uns getan haben – und verwenden eine ihrer vorgefertigten Komponenten.[5]
In unserem Fall befindet sich der vordefinierte Grafikteil im Paket `Prog1Tools` und setzt sich aus zwei Strukturen zusammen:

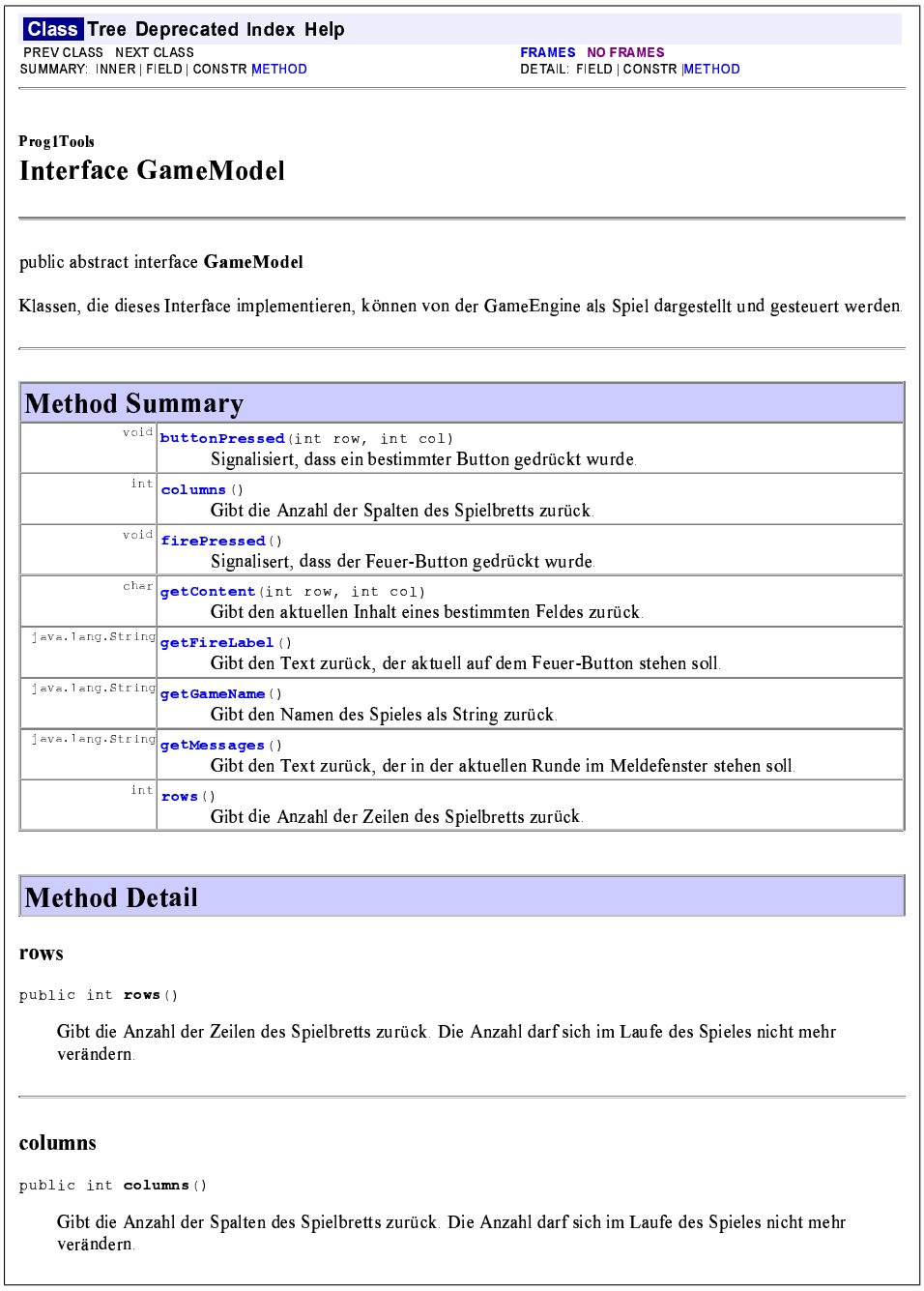
1. Ein Interface namens `GameModel` erlaubt es uns, beinahe beliebige Brettspiele (oder Spiele, die in ihrem Aufbau einem Brettspiel ähneln) durch ein und dieselbe Oberfläche darstellbar zu machen. Die verschiedenen Methoden, die auf den Seiten 118 und 119 im JavaDoc-Format spezifiziert sind, werden in diesem Kapitel noch näher erläutert.

2. Die Klasse `GameEngine` übernimmt die komplette Steuerung und den grafischen Aufbau unseres Spieles. Alles, was wir tun müssen, ist, ein Objekt der `GameEngine` zu erzeugen und dem Konstruktor eine Instanz des `GameModel` zu übergeben:

```
new GameModel(konkretesModell);
```

Wie können wir uns den Aufbau unserer Spieleoberfläche am besten vorstellen? Abbildung 5.4 skizziert das Datenmodell, das unserer Oberfläche zugrundeliegt.

[5] Dieser Vorgang der **Wiederverwertung** von bereits definierten Klassen macht eine der Stärken der objektorientierten Programmierung aus. Probleme wie die Programmierung eines Spieles lassen sich in thematisch zusammenhängende Blöcke (wie etwa die Programmierung der Grafik oder der Entwurf der Spielelogik) aufteilen, die dann unabhängig voneinander von verschiedenen Entwicklern bearbeitet werden. Hat man sich vorher auf eine genaue Schnittstelle geeinigt, arbeiten all diese Teile später reibungslos miteinander zusammen.



Class Tree Deprecated Index Help
PREV CLASS NEXT CLASS
SUMMARY: INNER | FIELD | CONSTR | METHOD
FRAMES NO FRAMES
DETAIL: FIELD | CONSTR | METHOD

Prog1Tools

Interface GameModel

public abstract interface **GameModel**

Klassen, die dieses Interface implementieren, können von der GameEngine als Spiel dargestellt und gesteuert werden.

Method Summary

void	**buttonPressed**(int row, int col) Signalisiert, dass ein bestimmter Button gedrückt wurde.
int	**columns**() Gibt die Anzahl der Spalten des Spielbretts zurück.
void	**firePressed**() Signalisert, dass der Feuer-Button gedrückt wurde.
char	**getContent**(int row, int col) Gibt den aktuellen Inhalt eines bestimmten Feldes zurück.
java.lang.String	**getFireLabel**() Gibt den Text zurück, der aktuell auf dem Feuer-Button stehen soll.
java.lang.String	**getGameName**() Gibt den Namen des Spieles als String zurück.
java.lang.String	**getMessages**() Gibt den Text zurück, der in der aktuellen Runde im Meldefenster stehen soll.
int	**rows**() Gibt die Anzahl der Zeilen des Spielbretts zurück.

Method Detail

rows

```
public int rows()
```

Gibt die Anzahl der Zeilen des Spielbretts zurück. Die Anzahl darf sich im Laufe des Spieles nicht mehr verändern.

columns

```
public int columns()
```

Gibt die Anzahl der Spalten des Spielbretts zurück. Die Anzahl darf sich im Laufe des Spieles nicht mehr verändern.

Abbildung 5.5: Dokumentation der Klasse `GameModel` (Seite 1)

getFireLabel

```
public java.lang.String getFireLabel()
```

Gibt den Text zurück, der aktuell auf dem Feuer-Button stehen soll.

---

getMessages

```
public java.lang.String getMessages()
```

Gibt den Text zurück, der in der aktuellen Runde im Meldefenster stehen soll.

---

getGameName

```
public java.lang.String getGameName()
```

Gibt den Namen des Spieles als String zurück.

---

getContent

```
public char getContent(int row,
                       int col)
```

Gibt den aktuellen Inhalt eines bestimmten Feldes zurück.
Parameters:
`row` - die Zeile, von 0 an gezählt
`col` - die Spalte, von 0 an gezählt

---

buttonPressed

```
public void buttonPressed(int row,
                          int col)
```

Signalisiert, dass ein bestimmter Button gedrückt wurde.
Parameters:
`row` - die Zeile, von 0 an gezählt
`col` - die Spalte, von 0 an gezählt

---

firePressed

```
public void firePressed()
```

Signalisert, dass der Feuer-Button gedrückt wurde.

Abbildung 5.6: Dokumentation der Klasse `GameModel` (Seite 2)

- Für die Ausgabe von Nachrichten an den Spieler bzw. die Spielerin gibt es ein spezielles Textfenster. Die `GameEngine` holt sich aus dem Datenmodell die darzustellenden Texte, indem sie die Methode `getMessages` des `GameModel`-Objektes aufruft. Wie dieser Text anschließend auf dem Bildschirm dargestellt wird, braucht uns als Entwickler nicht zu kümmern. Wir geben den Text lediglich als einen `String` zurück.
- Das eigentliche Spielbrett wird im Datenmodell als „Content" bezeichnet. Wie bei einem Schachbrett oder einer Tabelle setzt es sich aus einer festen Anzahl von Zeilen (englisch: rows) und Spalten (englisch: columns) zusammen. Die Anzahl der Zeilen bzw. Spalten ist fest und kann aus dem Modell durch die Methoden `rows` bzw. `columns` erfragt werden.

  Jedes Spielfeld, das durch das Modell angesprochen werden kann, wird anhand seiner Spalten- und Zeilennummer angesprochen (vgl. Abbildung 5.4). Es gibt zwei Aktionen, die im `GameModel` mit einem Feld geschehen können: der Inhalt des Spielfeldes kann abgefragt werden (`getContent`) und der Benutzer bzw. die Benutzerin kann auf ein einzelnes Spielfeld mit der Maus klicken (`buttonPressed`). Die durch das Klicken ausgelösten Aktionen (etwa, dass sich der Inhalt des Feldes verändert), werden innerhalb der Modellklasse vollzogen. Anschließend werden diese Änderungen von der `GameEngine` automatisch auf dem Bildschirm dargestellt. Der Inhalt eines Content-Feldes ist hierbei ein einzelnes Character-Zeichen (**`char`**) – in unserem Mastermind-Beispiel also etwa der Zustand einer einzelnen Zelle.
- Außerdem existiert noch der so genannte Feuer-Knopf (fire-button), der mit einem beliebigen Text belegt werden kann. Der Feuer-Knopf ist insbesondere für rundenbasierte Spiele (wie etwa Schach, bei dem man jeweils einen Zug machen kann) von Interesse. Die Beschriftung des Knopfes wird durch die Methode `getFireLabel` erfragt. Das Drücken des Knopfes wird durch die Methode `firePressed` symbolisiert.

Wie wir sehen, können wir also mit Hilfe unseres Interfaces ein Modell unseres Mastermind-Spieles erstellen. Die `GameEngine` kann dann dieses Modell verwenden, um das Spiel auf dem Bildschirm grafisch darzustellen.

5.7.4 Designphase

Bevor wir auf „Teufel komm 'raus" losprogrammieren, sollten wir anhand eines schlüssigen Entwurfes versuchen, die zu simulierende Situation in ein objektorientiertes Modell zu gießen. Wir werden den groben Entwurf am Klassenmodell (Abbildung 5.7) vornehmen und anschließend in einigen Fällen näher konkretisieren.
Es ist relativ klar, dass wir für die grafische Darstellung unseres Spieles die Klasse `GameEngine` verwenden wollen. Diese braucht allerdings ein Modell unseres Spieles (also ein `GameModel`), um dieses auf dem Bildschirm darstellen zu

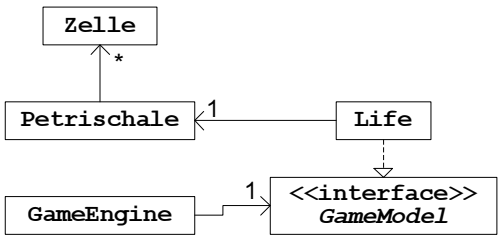


Abbildung 5.7: Game of Life – Grobentwurf

können.[6] Unsere Klasse `Life` implementiert deshalb das Interface[7] `GameModel`, das eben die Schnittstelle zwischen dem eigentlichen Spiel und der **GUI** (= graphical user interface, also quasi die Oberfläche auf dem Bildschirm) realisiert.
Wir wollen uns im weiteren Entwurf um die Klasse `Life` nicht weiter kümmern; diese ist durch die Schnittstellenvorgabe des `GameModel` sowieso bereits weitgehend festgelegt. Interessanter ist vielmehr die Frage, wie wir unsere Petrischale in den Computer bekommen.
Wir wollen prinzipiell zwischen zwei Klassen unterscheiden: einer Klasse `Zelle`, die das Modell einer einzelnen Zelle repräsentiert, und einer Klasse `Petrischale`, die für den Zustand der kompletten Petrischale in einer Generation steht. Wir wollen uns nun um die Schnittstelle dieser Klassen nach außen kümmern (vgl. Abbildung 5.8).
Die Klasse `Zelle` soll den Zustand einer einzelnen Zelle (also lebendig oder tot) speichern. Dieser Zustand, einmal festgelegt, soll unveränderlich feststehen (das heißt, ein Zugriff von außen soll nicht ermöglicht werden). Wir modellieren deshalb zwar eine Methode `istLebendig`, mit der wir die Vitalität erfragen können, lassen aber eine entsprechende set-Methode weg.
Da unsere Klasse über keine weiteren Zustände verfügt, lassen sich auf den ersten Blick keine weiteren Instanzmethoden identifizieren, die auf einem Zellobjekt irgendeinen Sinn machen würden. Wir beschäftigen uns deshalb mit der Frage, wie wir jenen einen Zustand (lebendig oder tot) bei der Initialisierung am besten setzen können – wie sollen also die Konstruktoren aussehen?
Ein einfacher Konstruktor, der den Zustand als **`boolean`**-Wert (**`true`** steht für „lebendig“) erhält, sollte auf jeden Fall vorhanden sein. Wir sehen aber noch einen zweiten Konstruktor vor, der statt einem zwei Argumente erhält:

1. Unser neuer Konstruktor soll verwendet werden, um eine neue Generation von Zellen anhand einer alten Generation zu erzeugen. Gemäß den Spielregeln benötigen wir hierfür den Zustand der entsprechenden `Zelle` der alten Generation

[6] Achten Sie auf die Verbindung zwischen `GameModel` und `GameEngine` in Abbildung 5.7. Es handelt sich hierbei nicht um einen Vererbungspfeil, sondern um eine Beziehung zwischen den Objekten. Eine `GameEngine` „hat ein“ `GameModel`.
[7] Beachten Sie: Der Vererbungspfeil zu Interfaces sieht etwas anders aus als der zu Superklassen.

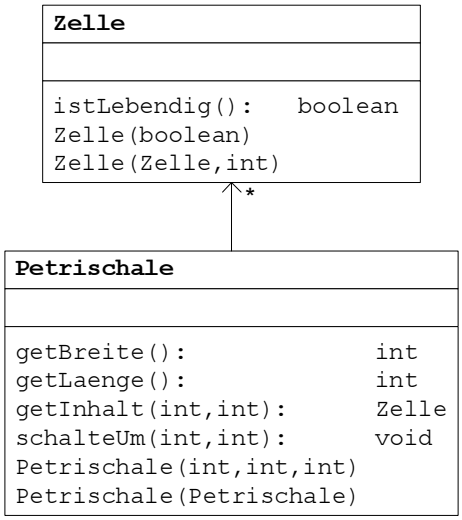


Abbildung 5.8: Die Klassen `Zelle` und `Petrischale`

2. und die Anzahl der lebenden Nachbarn unserer Zelle (ein **`int`**-Wert).

Mit diesen Parametern verfügen wir über alle notwendigen Daten, um den Zustand der neuen Zelle berechnen zu können. Wir werden uns im nächsten Abschnitt damit beschäftigen, eine konkrete Implementierung zu liefern.
Kommen wir nun zu unserer Petrischale. Wir legen die Petrischale „rechteckig" an, d. h. wir haben eine gewisse Anzahl von Zeilen (Länge der Schale) mit einer bestimmten Zahl von Spalten (Breite der Schale). Länge und Breite unserer Schale geben wir in den Methoden `getLaenge` und `getBreite` zurück.
Innerhalb unseres Objektes können wir beispielsweise ein Feld verwenden, um den tatsächlichen Inhalt unserer Petrischale abzuspeichern. Wir wollen uns hierauf im Moment noch nicht festlegen, da diese Entscheidung die Schnittstelle nach außen nicht beeinflusst und deshalb für den Entwurf nicht weiter wichtig ist. Wir modellieren jedoch eine Methode `getInhalt`, mit der wir in der Lage sind, eine einzelne Zelle aus unserer Petrischale auszulesen. Um hierbei die Möglichkeit von auftretenden Fehlern (was passiert zum Beispiel beim Aufruf von `getInhalt(-1,-1)`?) zu vermeiden, setzen wir das Ergebnis von `getInhalt` *immer auf eine tote Zelle*, sofern innerhalb der Klasse nicht eine lebendige Zelle gespeichert ist.
Für das spätere Spiel müssen wir ferner in der Lage sein, den Zustand einzelner Zellen in unserem Spielverband zu manipulieren. Der Spieler soll durch Mausklick in der Lage sein, eine bestimmte Zelle in der Schale von lebendig auf tot zu setzen (und umgekehrt). Wir sehen zu diesem Zweck eine Methode `schalteUm` vor, die eben diese Zustandsveränderung bewirkt.
Jetzt befassen wir uns noch mit den Möglichkeiten, eine neue Petrischale zu erzeu-

gen. Da wäre zunächst ein Konstruktor, der ein völlig neues Objekt ohne Wissen über vorherige Generationen erzeugt. Dieser Konstruktor muss lediglich wissen, wie lang und breit die Schale sein soll. Ferner übergeben wir ihm einen dritten ganzzahligen Wert, der die Zahl der lebendigen Zellen in der Petrischale festlegt. Der Konstruktor soll die übergebene Zahl von lebendigen Zellen zufällig in der Schale verteilen. Auf diese Weise erklären sich also die drei Parameter in unserem Konstruktor `Petrischale(int,int,int)` in Abbildung 5.8.
Jetzt fehlt in unserem Entwurf nur noch die Möglichkeit, eine neue Generation anhand der alten Generation zu erstellen. Zu diesem Zweck legen wir uns auf einen zweiten Konstruktor `Petrischale(Petrischale)` fest. Diesem Konstruktor wird die alte Generation in Form einer `Petrischale` übergeben. Es obliegt dann ihm, aus ihr die neue Generation zu erzeugen.

5.7.5 Die Klasse `Zelle`

Wir kommen nun zur Umsetzung unseres Entwurfs und beginnen mit der (noch sehr einfachen) Klasse `Zelle`. Unsere Zelle besitzt einen der beiden Zustände lebendig oder tot. Wir müssen diesen Zustand in irgendeiner Form abspeichern und fügen unserem Code deshalb eine private Instanzvariable namens `lebendig` hinzu:

```
/** Der Zustand der Zelle */
private boolean lebendig;
```

Die Variable besitzt einen **`boolean`**-Wert. Dieser ist genau dann **`true`**, wenn unsere Zelle lebendig ist. Unsere Methode `istLebendig` ist somit eine get-Methode für diesen Wert:

```
/** Prueft, ob die Zelle am Leben ist */
public boolean istLebendig() {
  return lebendig;
}
```

Ebenso einfach formuliert ist der erste unserer beiden Konstruktoren. Der übergebene Parameter, der eine Aussage über die Lebendigkeit unserer Zelle macht, muss lediglich in der Instanzvariablen abgelegt werden:

```
/** Konstruktor */
public Zelle(boolean istLebendig) {
  lebendig = istLebendig;
}
```

Der eigentlich interessante Part beginnt in unserem zweiten Konstruktor:

```
/** Konstruktor */
public Zelle(Zelle alt, int zahlDerLebendenNachbarn) {
```

Hier ist es unsere Aufgabe, aus unserer alten Zelle und der Zahl ihrer lebenden Nachbarn eine neue Zelle zu erzeugen. Gemäß den Spielregeln können wir folgende Regeln aufstellen:

- Ist die Zahl der lebenden Nachbarn genau 2, so setzen wir den Zustand der neuen Zelle auf den Zustand der alten Zelle.
- Ist die Zahl der lebenden Nachbarn genau 3, so setzen wir den Zustand der neuen Zelle auf lebend, also **`true`**.
- In jedem anderen Fall ist unsere Zelle tot; wir setzen ihren Zustand also auf **`false`**.

Drei Fälle, die sich allesamt auf die Zahl der lebenden Nachbarn stützen – das klingt schon sehr nach einer **`switch`**-Anweisung. Tatsächlich werden Sie feststellen, wie einfach sich diese Regeln in einem solchen Block realisieren lassen:

```
/** Konstruktor */
public Zelle(Zelle alt, int zahlDerLebendenNachbarn) {
  switch(zahlDerLebendenNachbarn) {
    case 2:
      lebendig = alt.lebendig;
      break;
    case 3:
      lebendig = true;
      break;
    default:
      lebendig = false;
      break;
  }
}
```

In den ersten beiden Fällen haben wir es mit einer konkreten Zahl (2 oder 3) zu tun, und wir können also direkt jeweils eine bestimmte **`case`**-Marke verwenden. Den letzten Fall (alle außer 2 und 3) realisieren wir einfach durch den **`default`**-Block.[8]
Im Folgenden sehen Sie noch einmal den kompletten Quellcode unserer Klasse. Vergleichen Sie ihn mit dem Entwurf in Abbildung 5.8. Sie sehen, dass wir uns genau an die vorgegebene Schnittstelle gehalten haben.

```
1  /** Eine einzelne Zelle */
2  public class Zelle {
3
4    /** Der Zustand der Zelle */
5    private boolean lebendig;
6
7    /** Prueft, ob die Zelle am Leben ist */
8    public boolean istLebendig() {
9      return lebendig;
10   }
11
12   /** Konstruktor */
13   public Zelle(boolean istLebendig) {
14     lebendig = istLebendig;
```

[8] Diesen Teil hätten wir übrigens auch weglassen können, da unsere Instanzvariable bei der Erzeugung automatisch mit **`false`** initialisiert wird. Es ist aber übersichtlicher, jeden möglichen Fall zu betrachten.

```
15    }
16
17    /** Konstruktor */
18    public Zelle(Zelle alt, int zahlDerLebendenNachbarn) {
19      switch(zahlDerLebendenNachbarn) {
20        case 2:
21          lebendig = alt.lebendig;
22          break;
23        case 3:
24          lebendig = true;
25          break;
26        default:
27          lebendig = false;
28          break;
29      }
30    }
31
32  }
```

5.7.6 Die Klasse `Petrischale`

Kommen wir nun zu unserer `Petrischale`. Eigentlich ist auch diese Klasse nicht schwer zu realisieren. Da der Code jedoch relativ lang ist, verteilen wir die Umsetzung auf mehrere zusammenhängende Einheiten.

5.7.6.1 Interne Struktur und einfacher Datenzugriff

Beginnen wir mit der Art und Weise, wie wir die Zellen in unserer Petrischale hinterlegen wollen. Der einfachste Weg an dieser Stelle (und momentan der einzige, da wir andere Datenspeicher noch nicht kennen) ist die Verwendung eines Feldes.

```
/** Ein Feld von Zellen */
private Zelle[][] inhalt;
```

Wir verwenden also ein zweidimensionales Feld namens `inhalt`, in dem wir unsere Zellen hinterlegen. Hierbei soll die Breite der Zahl der Einträge der ersten Dimension, die Länge der Zahl der Einträge der zweiten Dimension entsprechen. Aus diesem Entwurf ergibt sich somit auch automatisch die Definition von Breite und Länge unserer Petrischale:

```
/** Gib die Breite der Petrischale zurueck */
public int getBreite() {
  return inhalt.length;
}

/** Gib die Laenge der Petrischale zurueck */
public int getLaenge() {
  return inhalt[0].length;
}
```

Kommen wir nun zu den Möglichkeiten, Zugriff auf unser Feld von Zellen zu erlangen. Wir beginnen mit der Methode `getInhalt`, mit der wir den Inhalt un-

serer Petrischale auslesen:

```
/** Gibt die Zelle an einer bestimmten Position zurueck.
 Liegt der Index ausserhalb des darstellbaren Bereiches,
 wird eine tote Zelle zurueckgegeben.
 */
public Zelle getInhalt(int x, int y) {
  if (x < 0 || x >= inhalt.length ||
      y < 0 || y >= inhalt[0].length)
    return new Zelle(false);
  return inhalt[x][y];
}
```

Für den Fall, dass wir uns innerhalb des Bereiches befinden, der in unserem Feld hinterlegt ist, geben wir lediglich den Inhalt unseres Feldes an der Stelle `[x][y]` zurück. Andernfalls erzeugen wir mit Hilfe des **`new`**-Operators eine neue (tote) Zelle und geben diese als Ergebnis zurück.
Ebenso einfach wie der lesende Zugriff gestaltet sich auch der Schreibzugriff in unserer Methode `schalteUm`. Hier haben wir keine Kontrolle der Feldgrenzen gefordert; wir können uns obige Fallunterscheidung also ersparen. Wir setzen lediglich den Inhalt des Feldes an der Stelle `[x][y]` auf einen anderen Wert:

```
/** Setzt die Zelle an der Position (x,y) vom Zustand
 lebendig auf tot (oder umgekehrt).
 */
public void schalteUm(int x,int y) {
  inhalt[x][y] = new Zelle( ! inhalt[x][y].istLebendig() );
}
```

Wir verwenden hierzu wieder einmal den **`new`**-Operator, um ein neues Zellenobjekt zu erzeugen. Der Inhalt des Zellenobjektes soll genau das Gegenteil vom alten Zellzustand sein. Wir benutzen die logische Negation (das Ausrufezeichen), um aus **`true false`** bzw. aus **`false true`** zu machen. Den alten Zustand der Zelle erfahren wir mit Hilfe der Methode `istLebendig`.

5.7.6.2 Erster Konstruktor: Zufällige Belegung der Zellen

Wir wollen nun den Konstruktor spezifizieren, der unser Feld von Zellen mit zufälligen Werten füllt. Um den Aufwand bei der Implementierung möglichst gering zu halten, nehmen wir uns vor, so weit wie möglich bereits vordefinierte Methoden zu verwenden. Je mehr vorprogrammierte Elemente wir wieder verwerten können, desto weniger neue Fehler können wir in unser Programm einbauen.
Zuerst definieren wir uns eine private Hilfsmethode namens `mischen`, die ein Feld von Objekten zufällig durchmischt. Wir haben diese Methode schon im Praxisbeispiel um das BlackJack-Kartenspiel kennen gelernt und können ihre Formulierung Wort für Wort aus Abschnitt 5.5.4 übernehmen (mit der Ausnahme, dass wir jetzt ein Feld von Objekten durchmischen):

```
/** Mische ein Feld von Objekten */
private static void mischen(Object[] feld) {
  for (int i=0;i<feld.length;i++) {
```

```
      int j=(int)(feld.length*Math.random());
      Object dummy=feld[i];
      feld[i]=feld[j];
      feld[j]=dummy;
    }
  }
```

Kommen wir aber nun zu unserem eigentlichen Konstruktor. Dieser soll ein Feld von `breite` Zeilen und `hoehe` Spalten mit insgesamt `zahlDerLebenden` lebendigen Zellen füllen.

```
public Petrischale(int breite, int laenge, int zahlDerLebenden) {
```

Die erste Frage, die sich stellt, ist die Frage nach der Verteilung dieser Zellen. Wie soll man die Zellen am besten auf der Petrischale platzieren?
Aus der Erfahrung der Autoren[9] hat sich gezeigt, dass es den meisten nicht schwer fällt, ein Feld von Zellen mit *ungefähr* der gewünschten Zahl der lebenden Zellen zu füllen. Sie suchen sich hierzu einfach jeweils per Zufall ein Feld aus und setzen dies auf den neuen Zustand.
Diese Methode hat allerdings einen versteckten Nachteil: Was passiert, wenn das ausgesuchte Feld bereits auf lebend gesetzt wurde? In diesem Fall überschreiben wir eines unserer Felder doppelt, d. h. die Änderung des Inhalts wirkt sich nicht auf die Gesamtmenge der lebenden Zellen aus. Je mehr Felder wir zu füllen haben, desto größer ist somit die Wahrscheinlichkeit, dass obiger Fall auftritt. Ein beliebtes Testkriterium für Prüfer und Tutoren ist es deshalb, ein Feld von $10 \cdot 10$ Zellen mit 100 lebenden Zellen auffüllen zu lassen. Ist eine wie auch immer geartete Lücke entstanden, hat der Programmierer einen Fehler gemacht.
Andere haben aus diesem Problem gelernt und ihre Programme entsprechend angepasst. Die häufigste Vorgehensweise ist, in diesem Fall einfach eine Überprüfung des Feldinhaltes vorzuschalten. Ist das Feld bereits besetzt, so suche man sich eine neue Zufallszahl. Sie können sich vielleicht vorstellen, wie lange der Computer meistens sucht, wenn man etwa von 100 Feldern bereits 99 mit „lebendig“ besetzt hat.
Wir wollen aus diesem Grund einen etwas anderen Ansatz für das Problem suchen. Nehmen wir zuerst einmal an, es würde sich nicht um ein zweidimensionales, sondern nur um ein eindimensionales Feld handeln:

```
Zelle[] zellen = new Zelle[breite * laenge];
```

Wir wollen dieses Feld nun mit lebenden und toten Zellen füllen. Hierbei sollen die Zellen von `0` bis `zahlDerLebenden-1` mit lebenden, die restlichen mit toten Zellen gefüllt werden. Wir könnten uns zu diesem Zweck – ähnlich wie bei der Methode `mischen` – eine Methode `fuellen` definieren, die ein Feld von Objekten mit Werten füllt:

```
public void fuellen(Object[] feld,
```

[9] Eine vergleichbare Aufgabe wurde innerhalb von vier Jahren in acht aufeinanderfolgenden C++-Kursen gestellt. Hierbei zeigte sich, dass in jeder der Studierendengruppen die Verteilung einer der am häufigsten falsch programmierten Teile des Game of Life waren.

```
                    int von,
                    int bis,
                    Object fuelleHiermit) {
    for (int i = von; i < bis; i++) {
      feld[i] = fuelleHiermit;
    }
  }
```

Glücklicherweise ist die Definition einer solchen Methode nicht notwendig – ähnlich wie die Methode `System.arraycopy` ist auch eine Methode `java.util.Arrays.fill` bereits vordefiniert, die genau nach der obigen Syntax arbeitet. Das Auffüllen unseres Feldes beschränkt sich also auf zwei simple Methodenaufrufe:

```
java.util.Arrays.fill(zellen,
                      0,
                      zahlDerLebenden,
                      new Zelle(true));
java.util.Arrays.fill(zellen,
                      zahlDerLebenden,
                      zellen.length,
                      new Zelle(false));
```

Nun haben wir unser Feld `zellen` also mit lebenden und toten Zellen in genau der richtigen Menge aufgefüllt. Beachten Sie hierbei, dass wir zu diesem Zweck nur zwei tatsächlich vorhandene Objekte verwendet haben. Jede unserer Komponenten verweist entweder auf ein- und dasselbe lebendige oder auf das tote Zellobjekt. Würden wir also etwa das lebendige Zellobjekt an der Stelle `zellen[0][0]` manipulieren und auf tot setzen, so würde unsere Zelle automatisch kein einziges lebendiges Objekt mehr enthalten. Glücklicherweise kann uns dies jedoch nicht passieren, da wir für den inneren Zustand einer `Zelle` keine set-Methode definiert haben.[10]
Wir haben nun also ein eindimensionales Feld mit genau der richtigen Anzahl lebender und toter Zellen – jedoch leider noch in „geordneter" Form. Um diese Ordnung jedoch durch eine zufällige Reihenfolge zu ersetzen, können wir erneut auf Altbekanntes und Vordefiniertes zurückgreifen: Wir *mischen* das Feld einfach gut durch!

```
mischen(zellen);
```

So weit zum eindimensionalen Fall. Unser Feld wurde erzeugt, mit den richtigen Werten initialisiert und anschließend gemischt. Wie verfahren wir jedoch für unser zweidimensionales Feld?
Die Antwort auf diese Frage klingt wieder einmal simpler, als sie ist: *auf die gleiche Art und Weise.* Wie es der Zufall will, haben wir bei der Initialisierung des Feldes `zellen` bereits eine Länge von `breite` mal `laenge` angenommen, d. h. wir haben bereits ein Feld von ausreichend vielen lebenden und toten Zellen in

[10] Machen Sie sich an dieser Stelle noch einmal klar, warum dies ohne das Prinzip des **data hiding** nicht möglich gewesen wäre.

zufälliger Reihenfolge. Wir müssen diese Zellen also nur noch in unser zweidimensionales Array hineinkopieren:

```
for (int i = 0; i < breite; i++) {
  System.arraycopy(zellen,i*laenge,inhalt[i],0,laenge);
}
```

Achten Sie darauf, dass wir auch hier wieder einen vordefinierten Befehl (`System.arraycopy`) verwenden und uns auf diese Weise Programmierarbeit ersparen. Wenn wir den Konstruktor nun in seiner Gesamtheit betrachten, so werden wir feststellen, wie einfach und übersichtlich er strukturiert ist – obwohl er sich mit einem komplexen Problem befasst:

```
/** Konstruktor */
public Petrischale(int breite, int laenge, int zahlDerLebenden) {
  // Zuerst erzeugen wir ein vorbelegtes Feld von Zellen
  Zelle[] zellen = new Zelle[breite * laenge];
  java.util.Arrays.fill(zellen,
                        0,
                        zahlDerLebenden,
                        new Zelle(true));
  java.util.Arrays.fill(zellen,
                        zahlDerLebenden,
                        zellen.length,
                        new Zelle(false));
  mischen(zellen);
  // Nun tragen wir diesen Inhalt in unsere Zellen ein
  inhalt = new Zelle[breite][laenge];
  for (int i = 0; i < breite; i++) {
    System.arraycopy(zellen,i*laenge,inhalt[i],0,laenge);
  }
}
```

5.7.6.3 Zweiter Konstruktor: Die neue Generation

Kommen wir nun zur Berechnung der neuen Generation. Nehmen wir für den Anfang einmal an, wir hätten eine Hilfsmethode namens `zahlDerNachbarn`, die uns für jede beliebige Zelle aus unserer Petrischale die Zahl der lebenden Nachbarn nennen kann. Dann ist die Initialisierung unseres Feldes in wenigen Zeilen getan:

```
/** Konstruktor. Erzeugt aus einer alten Petrischale die
 neue Generation.
 */
public Petrischale(Petrischale alt) {
  inhalt = new Zelle[alt.getBreite()][alt.getLaenge()];
  for (int i = 0; i < inhalt.length; i++)
    for (int j = 0; j < inhalt[i].length; j++)
      inhalt[i][j] = new Zelle(alt.getInhalt(i,j),
                               alt.zahlDerNachbarn(i,j));
}
```

Was haben wir in unserem Programm getan? Zuerst haben wir das Feld `inhalt` gemäß der Breite und der Länge des alten Feldes initialisiert. Anschließend haben

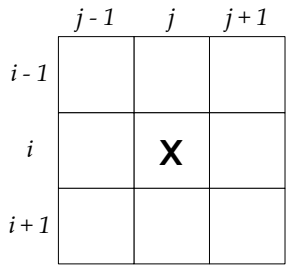


Abbildung 5.9: Nachbarn einer Zelle

wir in zwei geschachtelten Schleifen für jede Komponente des neuen Feldes eine neue Instanz der Klasse `Zelle` erzeugt. Dem hierbei verwendeten Konstruktor haben wir die Zelle der alten Generation sowie die Zahl der lebenden Nachbarn übergeben. Der Zustand der Zelle wird gemäß den Spielregeln automatisch gesetzt:

```
inhalt[i][j] = new Zelle(alt.getInhalt(i,j),
                         alt.zahlDerNachbarn(i,j));
```

Um unseren Konstruktor lauffähig zu machen, müssen wir uns nur noch Gedanken um die Formulierung der Methode `zahlDerNachbarn` machen:

```
/** Gib die Zahl der lebenden Nachbarn einer Zelle zurueck */
private int zahlDerNachbarn(int x,int y) {
```

Wie gehen wir hier am besten vor? Zuerst machen wir uns klar, dass eine Zelle insgesamt acht Nachbarn besitzt (vgl. Abbildung 5.9). Wir können diese Nachbarn in einer geschachtelten Schleife durchgehen und in einem Zähler vermerken, wie viele von ihnen lebendig sind:

```
int res = 0;
// Summiere die Zelle und die acht umgebenden Felder
for (int i = x-1; i <= x+1; i++) {
  for (int j = y-1; j <= y+1; j++) {
    res += (getInhalt(i,j).istLebendig()) ? 1 : 0;
  }
}
```

Achten Sie hierbei auf den Umstand, dass wir die Methode `getInhalt` zum Zugriff auf das Feld verwenden, obwohl wir innerhalb der Klasse direkten Zugriff auf die Daten haben. Diesen Trick verwenden wir, um die Behandlung lästiger Sonderfälle zu erledigen. Wenn wir uns beispielsweise in der linken oberen Ecke befinden (`x==0`, `y==0`), so versucht unsere Schleife beispielsweise, den Feldinhalt an der Stelle `[-1][-1]` auszulesen. Unsere get-Methode fängt derartige Probleme jedoch automatisch ab und liefert eine tote Zelle zurück.
Wir haben nun über alle Nachbarn iteriert und hierbei in der Variablen `res` ihre Gesamtsumme hinterlegt. Allerdings haben wir hierbei einen kleinen Fehler gemacht: wir haben die eigentliche Zelle selbst (im Bild mit einem **X** markiert)

ebenfalls mitgezählt (an der Stelle `i==x` und `j==y`). Diesen Fehler werden wir nachträglich korrigieren:

```
if (getInhalt(x,y).istLebendig())
  res--;
```

Nun haben wir unsere Methode komplett und können das in `res` gespeicherte Ergebnis zurückgeben. Unser zweiter Konstruktor ist somit endgültig lauffähig.

5.7.6.4 Die komplette Klasse im Überblick

Werfen wir noch einmal einen Blick auf unsere komplette Klasse. Vergleichen Sie die einzelnen (öffentlichen) Methoden mit unserem Schnittstellendesign in Abbildung 5.8. Sie werden feststellen, dass die konkrete Realisierung die im UML-Diagramm definierte Schnittstelle voll und ganz erfüllt. Ein anderer Programmierer, der mit der gleichen Schnittstelle gearbeitet hat, müsste somit eine Implementierung liefern, die in der Funktionalität mit der unseren übereinstimmt.

```
1  /** Eine Kolonie von Zellen */
2  public class Petrischale {
3
4    /** Ein Feld von Zellen */
5    private Zelle[][] inhalt;
6
7    /** Mische ein Feld von Objekten */
8    private static void mischen(Object[] feld) {
9      for (int i=0;i<feld.length;i++) {
10       int j=(int)(feld.length*Math.random());
11       Object dummy=feld[i];
12       feld[i]=feld[j];
13       feld[j]=dummy;
14     }
15   }
16
17   /** Konstruktor */
18   public Petrischale(int breite, int laenge, int zahlDerLebenden) {
19     // Zuerst erzeugen wir ein vorbelegtes Feld von Zellen
20     Zelle[] zellen = new Zelle[breite * laenge];
21     java.util.Arrays.fill(zellen,
22                           0,
23                           zahlDerLebenden,
24                           new Zelle(true));
25     java.util.Arrays.fill(zellen,
26                           zahlDerLebenden,
27                           zellen.length,
28                           new Zelle(false));
29     mischen(zellen);
30     // Nun tragen wir diesen Inhalt in unsere Zellen ein
31     inhalt = new Zelle[breite][laenge];
32     for (int i = 0; i < breite; i++) {
33       System.arraycopy(zellen,i*laenge,inhalt[i],0,laenge);
34     }
35   }
36
```

```
37    /** Gib die Breite der Petrischale zurueck */
38    public int getBreite() {
39      return inhalt.length;
40    }
41
42    /** Gib die Laenge der Petrischale zurueck */
43    public int getLaenge() {
44      return inhalt[0].length;
45    }
46
47    /** Gibt die Zelle an einer bestimmten Position zurueck.
48     Liegt der Index ausserhalb des darstellbaren Bereiches,
49     wird eine tote Zelle zurueckgegeben.
50     */
51    public Zelle getInhalt(int x, int y) {
52      if (x < 0 || x >= inhalt.length ||
53          y < 0 || y >= inhalt[0].length)
54        return new Zelle(false);
55      return inhalt[x][y];
56    }
57
58    /** Gib die Zahl der lebenden Nachbarn einer Zelle zurueck */
59    private int zahlDerNachbarn(int x,int y) {
60      int res = 0;
61      // Summiere die Zelle und die acht umgebenden Felder
62      for (int i = x-1; i <= x+1; i++) {
63        for (int j = y-1; j <= y+1; j++) {
64          res += (getInhalt(i,j).istLebendig()) ? 1 : 0;
65        }
66      }
67      // Ziehe die eigentliche Zelle wieder ab
68      if (getInhalt(x,y).istLebendig())
69        res--;
70      // Gib das Resultat zurueck
71      return res;
72    }
73
74    /** Konstruktor. Erzeugt aus einer alten Petrischale die
75     neue Generation.
76     */
77    public Petrischale(Petrischale alt) {
78      inhalt = new Zelle[alt.getBreite()][alt.getLaenge()];
79      for (int i = 0; i < inhalt.length; i++)
80        for (int j = 0; j < inhalt[i].length; j++)
81          inhalt[i][j] = new Zelle(alt.getInhalt(i,j),
82                                   alt.zahlDerNachbarn(i,j));
83    }
84
85    /** Setzt die Zelle an der Position (x,y) vom Zustand
86     lebendig auf tot (oder umgekehrt).
87     */
88    public void schalteUm(int x,int y) {
89      inhalt[x][y] = new Zelle( ! inhalt[x][y].istLebendig() );
90    }
91
```

```
92  }
```

5.7.7 Die Klasse `Life`

Kommen wir nun zu unserer Klasse `Life`, die eine Implementierung des `GameModel` darstellt und somit eine Anbindung an die Grafik der `GameEngine` realisiert:

```
import Prog1Tools.*;

/** Dieses GameModel realisiert das Game of Life */
public class Life implements GameModel {
```

Wir beginnen damit, die momentan dargestellte Generation in einer privaten Instanzvariablen zu hinterlegen. Unser Konstruktor wird diese `Petrischale` lediglich mit Hilfe des **`new`**-Operators instantiieren:

```
  /** Hier wird die aktuelle Petrischale gespeichert */
  private Petrischale zellen;

  /** Konstruktor. Uebergeben werden die Laenge und
   die Breite des Spielfeldes sowie die Zahl der
   Zellen, die zu Anfang leben sollen.
   */
  public Life(int breite, int hoehe, int lebendig) {
    zellen = new Petrischale(breite,hoehe,lebendig);
  }
```

Mit dieser einen Petrischale haben wir alles vordefiniert, was wir an Daten zur Erfüllung des Interfaces benötigen. So können wir für die Anzahl der Zeilen und Spalten etwa die Breite und Höhe unserer Petrischale verwenden:

```
  /** Anzahl der Zeilen */
  public int rows() {
    return zellen.getBreite();
  }

  /** Anzahl der Spalten */
  public int columns() {
    return zellen.getLaenge();
  }
```

Da wir in unserem Spiel des Lebens keine besonderen Nachrichten auszugeben haben, verwenden wir die Methode `getMessages` einfach für eine mehr oder weniger sinnvolle Spielanleitung:

```
  /** Message - Text */
  public String getMessages() {
    return
      "Spiel das Spiel des Lebens.\n" +
      "Schaue, was passiert,\n" +
      "Wenn das Leben eben\n" +
      "vor sich hinmutiert.";
  }
```

Auch der Name unseres Spieles bleibt natürlich konstant. Das Gleiche gilt für die Aufschrift auf unserem Feuerknopf:

```
/** Gibt den Namen des Spieles als String zurueck */
public String getGameName() {
  return "Game of Life";
}

/** Feuer-Knopf */
public String getFireLabel() {
  return "Naechste Generation";
}
```

Kommen wir nun zum Inhalt unserer einzelnen Zellen. Hier fragen wir mit Hilfe der Methode `istLebendig` ab, ob unsere Zelle als lebendig betrachtet werden kann. Wenn ja, liefern wir das Zeichen `O` als auszugebenden Wert zurück. Andernfalls verwenden wir ein Leerzeichen, das den entsprechenden Knopf also leer lässt:

```
/** Zustand der aktuellen Zelle */
public char getContent(int row, int col) {
  return
    (zellen.getInhalt(row,col).istLebendig()) ?
    'O' : ' ';
}
```

Last but not least müssen wir natürlich noch auf Aktionen des Benutzers bzw. der Benutzerin reagieren. Drückt er bzw. sie auf eine bestimmte Zelle (`buttonPressed`), so möchte er den Zustand der entsprechenden Zelle neu setzen. Wir können hierzu die in `Petrischale` definierte Methode `schalteUm` verwenden. Wird der Feuerknopf aktiviert (`firePressed`), so errechnen wir aus der alten Generation eine neue, indem wir den entsprechenden Konstruktor der `Petrischale` verwenden. Unsere Instanzvariable verweist in Zukunft nun auf die neue Generation:

```
/** Schalte eine bestimmte Zelle um */
public void buttonPressed(int row,int col) {
  zellen.schalteUm(row,col);
}

/** Berechne die naechste Generation */
public void firePressed() {
  zellen = new Petrischale(zellen);
}
```

Unsere eigentliche Klasse `Life` ist somit fertig; wir können das Spiel des Lebens spielen. In einer abschließenden `main`-Methode tun wir genau dies, wobei wir die notwendigen Daten (Anzahl der Zeilen, Spalten und lebendigen Zellen) eingeben lassen:

```
/** Hauptprogramm */
public static void main(String[] args) {
  new GameEngine(
    new Life(IOTools.readInteger("Anz. der Zeilen  :"),
```

```
                IOTools.readInteger("Anz. der Spalten :"),
                IOTools.readInteger("Anz. der Lebenden:")));
    }
```

5.7.8 Fazit

Das Spiel des Lebens ist eine anspruchsvolle Aufgabe, die dem Programmierer ein gewisses Maß an Voraussicht und ein gutes Gespür für objektorientierten Entwurf abverlangt. Hat man sich jedoch einmal auf ein bestimmtes Design festgelegt und realisiert man die einzelnen Objekte Schritt für Schritt, so stellen selbst dermaßen schwierig erscheinende Probleme für das geübte Auge kein unüberwindliches Hindernis dar.
In diesem Abschnitt haben Sie gelernt, dass es zwischen Objekten mehr Beziehungen geben kann als nur die einfache Vererbung. Obwohl die Klasse `Life` ein Interface implementiert hat, war doch die ausschlaggebende Beziehung in diesem Beispiel nicht die „ist-ein“, sondern die „hat-ein“-Beziehung. Vererbung ist ein probates Mittel in vielen Situationen; sie ist jedoch nicht in jeder Situation anwendbar.
Ferner haben wir auf den letzten Seiten erfahren, wie wertvoll die Wiederverwertung von bereits bekannten Problemlösungen sein kann. Was man bereits einmal erfolgreich eingesetzt hat, lässt sich auch auf andere Probleme übertragen. Standardprobleme (wie etwa das Kopieren oder Sortieren von Feldern bzw. das Mischen) besitzen oftmals Standardlösungen, die Sie in allgemein verbreiteten Softwarebibliotheken (oftmals direkt mit Java ausgeliefert) entdecken können. Versuchen Sie nicht, das Rad jedes Mal aufs Neue zu erfinden!

5.7.9 Übungsaufgaben

Aufgabe 5.7

Erinnern Sie sich an das Achtdamenproblem (Seite 87). Passen Sie das Programm so an, dass es

- alle Lösungen statt einer Lösung berechnet und
- diese Lösungen mit Hilfe der `GameEngine` auf dem Bildschirm darstellt. Sie können den Feuerknopf verwenden, um zwischen den verschiedenen Lösungen hin- und herzuschalten.

Aufgabe 5.8

Die folgende Übungsaufgabe stellt die Aufgabe des „Game of Life“ dar, wie sie üblicherweise in einem Stadium gestellt wird, in dem die Studierenden noch keine Erfahrung mit objektorientierter Programmierung haben. Versuchen Sie wie diese Studierenden, die Aufgabe nur mit statischen Methoden und Feldern zu lösen. Welche Vorgehensweise finden Sie einfacher: mit oder ohne Objektorientierung?

Das „Spiel des Lebens“ soll für eine Matrix von $n \times m$ gleichartigen Zellen programmiert werden, ein sog. *Gewebe*. Eine Zelle kann sich in genau einem der Zustände `lebendig` (=**`true`**) oder `tot` (=**`false`**) befinden. Als Nachbarn einer Zelle bezeichnet man alle Zellen, die links, rechts, oberhalb, unterhalb oder diagonal versetzt der Zelle liegen. (Vorsicht: Am Rand existieren nicht alle diese Zellen.) Eine Zelle im Innern hat also genau acht, Randzellen haben fünf und Eckzellen genau drei Nachbarzellen.
Die Zellen sind zeilenweise wie folgt indiziert:

$$\begin{array}{cccc} (1,1) & (1,2) & \dots & (1,m) \\ (2,1) & (2,2) & \dots & (2,m) \\ \vdots & \vdots & & \vdots \\ (n,1) & (n,2) & \dots & (n,m) \end{array}$$

Beispielsweise hat die Eckzelle $(1,1)$ also die Nachbarn $(1,2)$, $(2,1)$ und $(2,2)$.
Ausgehend von einer Anfangsgeneration wird eine neue Zellgeneration nach folgenden Regeln erzeugt:

- Eine Zelle wird (unabhängig von ihrem derzeitigen Zustand) in der nächsten Generation `tot` sein, wenn sie in der jetzigen Generation weniger als zwei oder mehr als drei lebende Nachbarn besitzt.
- Eine Zelle mit genau zwei lebenden Nachbarn ändert ihren Zustand nicht.
- Eine Zelle mit genau drei lebenden Nachbarn wird sich in der nächsten Generation im Zustand `lebendig` befinden.

Hinweis: Man darf erst dann die alte Generation ändern, wenn alle Entscheidungen für die neue Generation getroffen wurden.

Bearbeiten Sie folgende Aufgaben:

a) Man schreibe eine Methode `erzeugeGewebe`, die die Dimensionen des Gewebes als Argumente erhält und die Referenz auf ein entsprechendes **`boolean`**-Feld zurückliefert; der Zustand einer einzelnen Zelle soll zufällig initialisiert werden – hierzu können Sie folgendes Konstrukt verwenden:

```
boolean zustand =
  ( ((int)(Math.random()*10))%2 == 0 ) ? true : false;
```

b) Man schreibe eine Methode `leseGewebe` mit der gleichen Signatur wie die obige Methode, nur mit dem Unterschied, dass die Zustände der Zelle jetzt mit Hilfe der `IOTools` von der Tastatur eingelesen werden.

c) Man schreibe eine Methode `druckeGewebe`, die ein übergebenes Gewebe auf der Standardausgabe darstellt: Eine `lebendige` Zelle soll dabei durch „`*`“ repräsentiert werden, für `tote` Zellen ist als Platzhalter ein Leerzeichen einzufügen; die Gewebematrix ist oben und unten durch “`-`”, links und rechts durch “`|`” einzurahmen. Beispiel für $n = 5, m = 4$:

Konsole

```
------
|*  *|
| ** |
| * *|
|** *|
|  * |
------
```

d) Man schreibe eine Methode `getAnzNachbarn`, die innerhalb eines als Argument übergebenen Gewebes die Anzahl der Nachbarn einer bestimmten Zelle berechnet.

e) Man schreibe eine Methode `nextGeneration`, die zu einem übergebenen Gewebe eine entsprechend der obigen Regeln erzeugte Nachfolgegeneration zurückgibt.

f) Fragen Sie in der `main`-Methode zunächst die Dimension des Gewebes ab - je nach Benutzerwunsch soll sein Anfangszustand dann von Tastatur eingelesen werden oder zufällig erzeugt werden. Geben Sie daraufhin die nächsten fünf Generationen aus.

Aufgabe 5.9

Überarbeiten Sie Ihre Lösung aus der letzten Aufgabe so, dass sie mit Hilfe der `GameEngine` darstellbar ist.

5.8 Rechnen mit rationalen Werten

5.8.1 Vorwissen aus dem Buch

- Kapitel 8 (Der grundlegende Umgang mit Klassen) sowie
- Kapitel 12 (Einige wichtige Hilfsklassen).

Wir haben gelernt, dass die Klassen `BigInteger` und `BigDecimal` uns die Möglichkeit bieten, mit beliebig langen Ganzzahlen bzw. Dezimalzahlen zu arbeiten. Dies ist eine sehr praktische Sache – in manchen Fällen aber immer noch nicht ausreichend. Es gibt Zahlen, die sich in einem Gleitkommasystem einfach nicht mit endlich vielen Ziffern darstellen lassen. Hierzu zählen neben den irrationalen Zahlen wie $\sqrt{2}$ viele rationale Werte. So hat etwa (wie wir später im Beispiel sehen werden) die Zahl 0.1 keine endliche Darstellung im Binärsystem. Im Dezimalsystem wäre $\frac{1}{3}$ ein vergleichbarer Fall.
In diesem Abschnitt werden wir eine Klasse schreiben, die uns das exakte Rechnen mit beliebigen Bruchzahlen ermöglicht. Wir stellen hierbei eine Bruchzahl durch zwei Zahlen `zaehler` und `nenner` dar, wobei `nenner` nicht negativ oder

null sein darf. Ferner wollen wir in unserer Darstellung `zaehler` und `nenner` so weit wie möglich kürzen. Dies lässt sich dadurch erreichen, dass wir beide Werte durch ihren größten gemeinsamen Teiler dividieren:

$$\frac{\text{zaehler}}{\text{nenner}} = \frac{\text{zaehler} \div \text{ggt(zaehler,nenner)}}{\text{nenner} \div \text{ggt(zaehler,nenner)}}$$

Anschließend definieren wir die vier Grundrechenarten Addition, Subtraktion, Multiplikation und Division gemäß den Regeln

$$\frac{a}{b} + \frac{c}{d} = \frac{a \cdot d + c \cdot b}{b \cdot d} \tag{5.1}$$

$$\frac{a}{b} - \frac{c}{d} = \frac{a}{b} + \frac{-c}{d} \tag{5.2}$$

$$\frac{a}{b} \cdot \frac{c}{d} = \frac{a \cdot c}{b \cdot d} \tag{5.3}$$

$$\frac{a}{b} \div \frac{c}{d} = \frac{a \cdot d}{b \cdot c} \tag{5.4}$$

5.8.2 Variablen und Konstruktoren

Unsere Klasse `Bruchzahl` besitzt zwei Instanzvariablen vom Typ `BigInteger`, die Zähler und Nenner des Bruchs repräsentieren:

```
public class Bruchzahl {

  /** Der Zaehler der Bruchzahl */
  private BigInteger zaehler;

  /** Der Nenner der Bruchzahl */
  private BigInteger nenner;
```

Analog zu den Wrapper-Klassen konzipieren wir die Klasse `Bruchzahl` so, dass ein einmal instantiiertes Objekt nicht mehr veränderbar ist. Wir definieren einen Konstruktor, dem wir Zähler und Nenner als Argumente übergeben. Den Fall, dass ein Benutzer der Klasse versehentlich den Nenner auf Null setzen will, fangen wir ab:

```
  /** Konstruktor, dem Zaehler und Nenner als BigInteger
    * uebergeben werden.
    * @param zaehler der Zaehler der Bruchzahl
    * @param nenner der Nenner der Bruchzahl
    **/
  public Bruchzahl(BigInteger zaehler,BigInteger nenner) {
    if (nenner.equals(BigInteger.valueOf(0)))
    {
      throw new ArithmeticException("Nenner darf nicht 0 sein.");
    }
    this.zaehler = zaehler;
    this.nenner = nenner;
    // Sonderfall: wir behandeln die Zahl 0
```

```
    if (zaehler.equals(BigInteger.valueOf(0)))
      nenner = BigInteger.valueOf(1);
    // Normalfall: die Zahl ist nicht null
    else {
      // Berechne den groessten gemeinsamen Teiler
      BigInteger ggt = zaehler.gcd(nenner);
      // Teile Zaehler und Nenner durch den ggt
      this.zaehler = zaehler.divide(ggt);
      this.nenner = nenner.divide(ggt);
    }
    // Sorge dafuer, dass das Vorzeichen im Zaehler steckt
    if (this.nenner.signum() < 0)
    {
      this.nenner = nenner.negate();
      this.zaehler = zaehler.negate();
    }
  }
```

Nachdem wir die Werte `zaehler` und `nenner` unserer Instanz gesetzt haben, müssen wir die Werte noch normalisieren – sprich, wir haben dafür zu sorgen, dass der Nenner unserer `Bruchzahl` ein positiver Wert ist und dass `zaehler` und `nenner` gegebenenfalls gekürzt werden. Beachten Sie, dass für die hierzu notwendige Berechnung des größten gemeinsamen Teilers bereits eine Methode in der Klasse `BigInteger` zur Verfügung steht.
Natürlich ist es für die Benutzung der Klasse sehr unbequem, die verschiedenen Zahlenformate manuell in das gewünschte Eingabeformat umrechnen zu müssen. Für ganzzahlige Werte stellen wir deshalb entsprechende Konstruktoren zur Verfügung. Sie sollen dem Programmierer das Leben vereinfachen:

```
  /** Konstruktor, dem Zaehler und Nenner
    * als long-Werte uebergeben werden.
    **/
  public Bruchzahl(long zaehler,long nenner) {
    this(BigInteger.valueOf(zaehler),BigInteger.valueOf(nenner));
  }

  /** Konstruktor, der einen long-Wert als Eingabegroesse nimmt.
    * @param zahl eine Gleitkommazahl
    **/
  public Bruchzahl(long zahl) {
    this(zahl,1);
  }
```

Können wir etwas Ähnliches auch für Gleitkommazahlen bewerkstelligen? Beginnen wir mit einer `BigDecimal`-Zahl. Wenn d die Zahl, n die Anzahl der Nachkommastellen und z die Ziffern der Zahl (das Komma haben wir also herausgestrichen) sind, so gilt:

$$d = \frac{z}{10^n}.$$

Für die Berechnung von n und z stellt uns `BigDecimal` die Methoden `scale()` und `unscaledValue()` zur Verfügung:

```
  /** Konstruktor, der eine Gleitkommazahl als Eingabegroesse
```

```
   * nimmt.
   * @param zahl eine Gleitkommazahl als BigDecimal
   **/
public Bruchzahl(BigDecimal zahl) {
  this(zahl.unscaledValue(),
    BigInteger.valueOf(10).pow(zahl.scale()));
}
```

Unter Verwendung dieses neuen Konstruktors stehen uns nun alle Umwandlungsmöglichkeiten der Klasse `BigDecimal` zur Verfügung:

```
/** Konstruktor, der eine Gleitkommazahl als Eingabegroesse
  * nimmt.
  * @param zahl eine Gleitkommazahl in Textrepraesentation,
  *  z.B. "0.123"
  **/
public Bruchzahl(String zahl) {
  this(new BigDecimal(zahl));
}

/** Konstruktor, der einen double-Wert als Eingabegroesse nimmt.
  * @param zahl eine Gleitkommazahl
  **/
public Bruchzahl(double zahl) {
  this(new BigDecimal(zahl));
}
```

Beachten Sie übrigens, dass die Konstruktor-Aufrufe **new** `Bruchzahl("0.1")` und **new** `Bruchzahl(0.1)` unterschiedliche Ergebnisse liefern. Während der erste Aufruf wie erwartet die Zahl $\frac{1}{10}$ repräsentiert, erzeugt der zweite Aufruf den Bruch

$$\frac{3602879701896397}{36028797018963968}.$$

Dies ist zwar sehr nahe an $\frac{1}{10}$, trifft aber doch nicht ganz unsere Erwartungen. Der Grund ist der bereits oben erwähnte Umstand, dass 0.1 im Binärsystem nicht mit endlich vielen Stellen dargestellt werden kann. Beim Übersetzen des Programms setzt Java Gleitkommakonstanten deshalb auf jenen Wert, der der dezimalen Darstellung am nächsten liegt. Bei der Ausgabe von **float**- oder **double**-Werten findet der umgekehrte Prozess statt. Man rechnet also schon von Anfang an mit ungenauen Werten.

5.8.3 `toString`, `equals` und `hashCode`

Wie im ersten Teil bereits erwähnt, gibt es drei Methoden, die man bei Klassen oftmals überschreibt:

- die Methode `toString()`, die eine textuelle Darstellung eines Objektes liefert,
- die Methode `equals()`, die zwei Objekte auf Gleichheit prüft, und

- die Methode `hashCode()`, die konsistent mit der Methode `equals()` gehalten werden muss.[11]

Wir wollen in unserer `Bruchzahl`-Klasse diese drei Methoden ebenfalls überschreiben:

```
/** Liefert eine String-Darstellung dieser Zahl */
public String toString() {
  return (nenner.equals(BigInteger.valueOf(1))) ?
    zaehler.toString() :
    (zaehler.toString() + '/' + nenner.toString());
}

/** Vergleicht zwei Objekte auf Gleichheit
  * @param o das zu vergleichende Objekt
  * @return true genau dann, wenn das andere Objekt
  *  auch eine Bruchzahl vom gleichen Inhalt ist.
  **/
public boolean equals(Object o) {
  if (o == null)
    return false;
  if (this == o)
    return true;
  if (getClass() != o.getClass())
    return false;
  Bruchzahl b = (Bruchzahl) o;
  return zaehler.equals(b.zaehler) && nenner.equals(b.nenner);
}

/** Liefert einen Hashcode fuer dieses Objekt zurueck.
  * @return der Hashcode, berechnet aus Zaehler und Nenner
  **/
public int hashCode() {
  return zaehler.hashCode() * nenner.hashCode();
}
```

Die neue `toString`-Methode gibt Zähler und Nenner unserer Bruchzahl aus. Zwei `Bruchzahl`-Objekte sind gemäß `equals()` genau dann gleich, wenn `zaehler` und `nenner` übereinstimmen.

5.8.4 Die vier Grundrechenarten

Nachdem nun das Grundgerüst unserer `Bruchzahl`-Klasse steht, können wir uns an die Implementierung der Rechenoperationen machen. Die Formeln, wie auf Seite 138 beschrieben, beruhen auf Operationen, die für die Klasse `BigInteger` bereits definiert sind. Wir verwenden diese Operationen sowie unseren `Bruchzahl`-Konstruktor, um das Rechenergebnis als neues `Bruchzahl`-Objekt zurückzugeben. Für die Subtraktion zweier Bruchzahlen definieren wir

[11] Zwei Objekte, die gemäß `equals` gleich sind, müssen denselben `hashCode` zurückliefern. Andernfalls funktionieren einige `Collection`-Implementierungen, wie etwa `HashSet` oder `HashMap`, nicht.

eine Hilfsmethode namens `negate`. Da auch der Wechsel des Vorzeichens einer Zahl durchaus eine oft verwendete Operation ist, machen wir diese Methode ebenfalls öffentlich zugänglich:

```
/** Addiert zwei Bruchzahlen
  * @param zahl der zweite Summand
  * @return this + zahl
  **/
public Bruchzahl add(Bruchzahl zahl) {
  return new Bruchzahl(
    zaehler.multiply(zahl.nenner).add(nenner.multiply(zahl.zaehler)),
    nenner.multiply(zahl.nenner)
  );
}

/** Negiert eine Bruchzahl
  * @return -this
  **/
public Bruchzahl negate() {
  return new Bruchzahl(zaehler.negate(),nenner);
}

/** Subtrahiert zwei Bruchzahlen
  * @param zahl der Subtrahend
  * @return this - zahl
  **/
public Bruchzahl subtract(Bruchzahl zahl) {
  return add(zahl.negate());
}

/** Multipliziert zwei Bruchzahlen
  * @param zahl der zweite Faktor
  * @return this * zahl
  **/
public Bruchzahl multiply(Bruchzahl zahl) {
  return new Bruchzahl(
    zaehler.multiply(zahl.zaehler),
    nenner.multiply(zahl.nenner)
  );
}

/** Dividiert zwei Bruchzahlen
  * @param zahl der Divisor
  * @return this / zahl
  **/
public Bruchzahl divide(Bruchzahl zahl) {
  return new Bruchzahl(
    zaehler.multiply(zahl.nenner),
    nenner.multiply(zahl.zaehler)
  );
}
```

Damit ist unsere `Bruchklasse` komplett. Wir fügen zu guter Letzt lediglich noch eine `main`-Methode an, in der wir einige Funktionen unserer Klasse demonstrieren können:

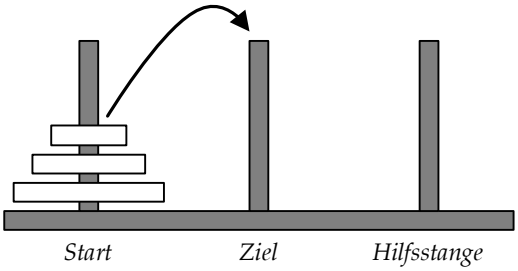


Abbildung 5.10: Die Türme von Hanoi

```
/** Ein kleiner Test unserer Implementierung */
public static void main(String[] args) {
 // Gib einige umgewandelte Zahlen aus
 System.out.println(new Bruchzahl(1,3));
 System.out.println(new Bruchzahl(2,-6));
 System.out.println(new Bruchzahl(3));
 System.out.println(new Bruchzahl(0.1)); // Achtung!
 System.out.println(new Bruchzahl("0.1"));
 System.out.println(new Bruchzahl(0));
 // Fuehre einige Rechenoperationen aus
 System.out.println(new Bruchzahl(1,3).add(new Bruchzahl(1,6)));
 System.out.println(new Bruchzahl(1,3).subtract(new Bruchzahl(1,6)));
 System.out.println(new Bruchzahl(2,3).multiply(new Bruchzahl(1,6)));
 System.out.println(new Bruchzahl(1,3).divide(new Bruchzahl(1,6)));
 // Vergleiche zwei Objekte auf Gleichheit
 System.out.println(new Bruchzahl(1,-3).equals(new Bruchzahl(-2,6)));
}
```

Führen wir diese Methode aus, erhalten wir folgende Ausgabe:

Konsole

```
1/3
-2/6
3
3602879701896397/36028797018963968
1/10
0
1/2
1/6
1/9
2
true
```

5.9 Die Türme von Hanoi

Bei den Türmen von Hanoi handelt es sich um ein klassisches Problem, das in viele Programmierkurse Einzug gehalten hat. Entsprechend soll es auch in diesem

Kurs nicht fehlen.
Gegeben sind drei Stangen. Auf der einen Stange ist eine Anzahl von Scheiben aufgereiht, nach Größe sortiert (siehe Abbildung 5.10). Aufgabenstellung sei es, die Scheiben von der ersten auf die zweite Stange zu versetzen.
Um das Ganze nicht allzu einfach zu machen, sind einige Grundregeln für das Verschieben gegeben:

- Es darf pro Zug immer nur eine Scheibe bewegt werden.
- Eine größere Scheibe darf niemals auf eine kleinere Scheibe gelegt werden.

Wir wollen ein Programm erstellen, das die durchzuführenden Züge für eine beliebige Anzahl von Scheiben berechnet und auf dem Bildschirm ausgibt.

5.9.1 Vorwissen aus dem Buch

- Kapitel 8 (Der grundlegende Umgang mit Klassen) sowie
- Kapitel 12 (Einige wichtige Hilfsklassen).

5.9.2 Designphase

So wie das Achtdamen-Problem handelt es sich bei den Türmen um ein klassisches Beispiel für einen rekursiven Ansatz. Wo wir jedoch zuvor mit einer kompliziert anmutenden (und daher fehleranfälligen) Indizierung auf einem Feld gearbeitet haben, wollen wir dieses Mal ein solides Objektmodell für uns arbeiten lassen.
Abbildung 5.11 zeigt ein Modell, das in seiner Struktur der Wirklichkeit aus Abbildung 5.10 nachempfunden ist. Eine Klasse `Hanoi` repräsentiert das Spielbrett und damit die zu lösende Aufgabe. Einem Konstruktor wird die Anzahl der Scheiben übergeben; eine Instanzvariable `zaehler` registriert die Anzahl der benötigten Züge. Mit Hilfe der `toString`-Methode können wir den Zustand des Spielbrettes zu einem bestimmten Zeitpunkt ausgeben.
Eine Instanz der Klasse `Hanoi` hat eine Assoziation zu drei `Stange`-Objekten. Jedes dieser Objekte repräsentiert eine der drei Stangen des Spiels. Mit Hilfe von `getHoehe()` lässt sich die Anzahl der Scheiben erfragen, die sich auf der Stange momentan befinden. Der Aufruf von `entferne()` entfernt die oberste `Scheibe` von der Stange. Die Methode `fuegeEin()` reiht eine `Scheibe` auf der `Stange` auf, wobei sichergestellt werden muss, dass niemals eine größere auf eine kleinere `Scheibe` gelegt wird. Der Erfolg des Einfügens wird durch den Booleschen Rückgabewert **`true`** signalisiert.
Um die Breite der einzelnen Scheiben miteinander vergleichen zu können, müssen Instanzen der Klasse `Scheibe` Informationen über ihre Breite liefern können. Auch dieser Punkt ist im Design berücksichtigt.

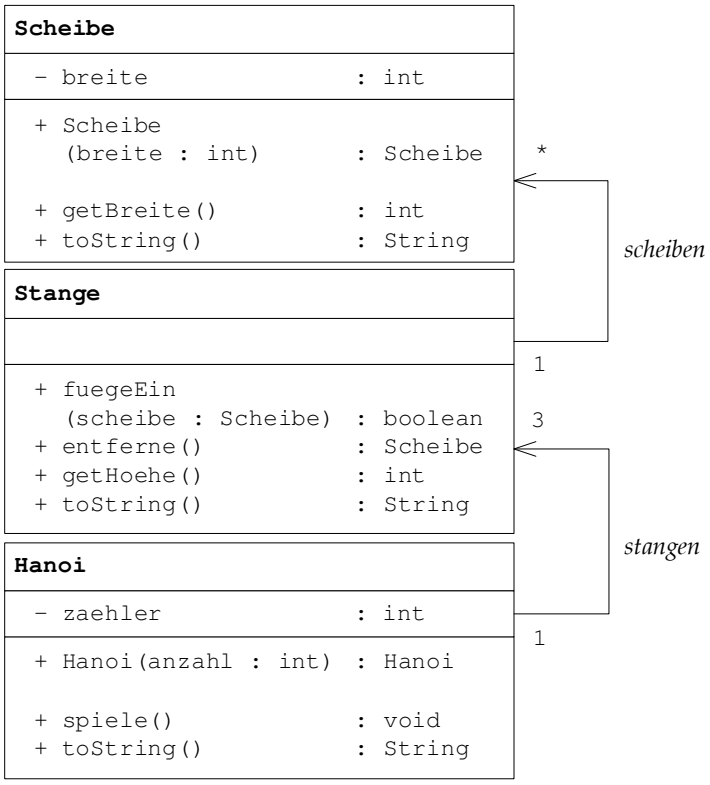


Abbildung 5.11: Türme von Hanoi – Klassendesign

5.9.3 Die Klasse `Scheibe`

Beginnen wir mit der einfachsten der drei Klassen. Die Klasse `Scheibe` vermerkt die Breite mit Hilfe von Instanzvariablen und ist in der Lage, diesen Inhalt in formatierter Weise als `String` zurückzuliefern:

```
1  import java.text.DecimalFormat;
2
3  /** Diese Klasse repraesentiert eine einzelne Scheibe,
4    * die waehrend des Tuerme-von-Hanoi-Spieles hin- und
5    * hergeschoben wird.
6    **/
7  public class Scheibe {
8
9    /** Formatierungs-Objekt fuer die Breite */
10   private final static DecimalFormat FORMAT = new DecimalFormat("00");
11
12   /** Wie breit ist die Scheibe ? */
13   private int breite;
14
```

```
15    /** Konstruktor.
16      * @param breite die Breite der Scheibe.
17      **/
18    public Scheibe(int breite) {
19      this.breite = breite;
20    }
21
22    /** Liefert die Breite der Scheibe zurueck
23      * @return die Breite der Scheibe
24      **/
25    public int getBreite() {
26      return breite;
27    }
28    /** Gibt eine String-Repraesentation des Objektes zurueck.
29      * @return eine textuelle Darstellung der Stange
30      **/
31    public String toString() {
32      return "(" + FORMAT.format(breite) + ")";
33    }
34  }
```

Für die formatierte Ausgabe in der `toString`-Methode verwenden wir ein `DecimalFormat`-Objekt. Wir geben die Zahl immer zweistellig aus und garantieren somit, dass jede `toString`-Repräsentation eines Scheiben-Objektes gleich breit ist.[12]

5.9.4 Die Klasse `Stange`

Kommen wir nun zu einer einzelnen `Stange`. Unsere Stangen müssen eine ständig wechselnde Anzahl von `Scheibe`-Objekten in der Reihenfolge beherbergen, in der sie aufgereiht wurden. Wir verwenden zur Speicherung eine Liste.

```
/** Diese Klasse repraesentiert eine Stange, auf
  * der einzelne Scheiben aufgereiht werden koennen.
  **/
public class Stange {

  /** Die einzelnen Scheiben werden in einer Liste
    * abgelegt.
    **/
  private List<Scheibe> scheiben = new ArrayList<Scheibe>();
```

Die Anzahl der gespeicherten Stangen zu erfahren, ist somit lediglich ein Aufruf der `size`-Methode unserer zugrunde liegenden `Collection`:

```
  /** Gib die Hoehe des Turms aus
    * @return die Anzahl der Scheiben, die auf der Stange
    * aufgereiht sind.
    **/
  public int getHoehe() {
```

[12] Wir gehen davon aus, dass wir keine Scheibe breiter als 99 haben. Schon mit 16 Scheiben würde unsere Lösung über 65000 Lösungsschritte erfordern, so dass eine Bildschirmausgabe nicht mehr sinnvoll wäre.

```
    return scheiben.size();
  }
```

Auch das Entfernen von Stangen ist mittels der `remove`-Methode leicht zu realisieren:

```
  /** Entfernt das oberste Element vom Turm.
    * @return das oberste Element, null, falls
    * die Stange leer ist.
    **/
  public Scheibe entferne() {
    if (scheiben.isEmpty()) {
      return null;
    }
    return (Scheibe)scheiben.remove(0);
  }
```

Wie sieht es jedoch mit dem Einfügen aus? Hier müssen wir die Breite unserer `Scheibe` mit der Breite des obersten Elements vergleichen. Das oberste Element sei in unserer Liste immer das Element an der Stelle 0, und wir können es mit Hilfe der `get`-Methode auslesen. Die Einfügeoperation selbst übernimmt für uns die Methode `set()`:

```
  /** Fuegt der Stange genau dann eine einzelne Scheibe
    * hinzu, wenn nicht schon eine kleinere Scheibe auf
    * der Stange sitzt.
    * @param scheibe die hinzugefuegte Scheibe
    * @return true, wenn das Einfuegen erfolgreich war
    **/
  public boolean fuegeEin(Scheibe scheibe) {
    // Test: haben wir versucht, null einzufuegen?
    if (scheibe == null)
      return false;
    // Test: ist die Stange leer?
    if (!scheiben.isEmpty()) {
      // Hole die oberste Scheibe aus der Liste
      Scheibe oben = (Scheibe) scheiben.get(0);
      // Vergleiche die Breiten der Scheiben
      if (oben.getBreite() < scheibe.getBreite())
        return false;
    }
    // Fuehre die Operation durch
    scheiben.add(0,scheibe);
    return true;
  }
```

Kommen wir nun zur Methode `toString()`. Hier müssen wir durch den ganzen Inhalt unserer Liste iterieren (unter Verwendung eines `Iterator`) und die jeweiligen `toString()`-Ausgaben der `Scheibe`-Objekte miteinander verknüpfen. Wir verwenden einen `StringBuffer`, um während der Iteration die Zwischenergebnisse abzuspeichern:

```
  /** Gibt eine String-Repraesentation des Objektes zurueck.
    * @return eine textuelle Darstellung der Stange
    **/
```

```
  public String toString() {
    // Wir speichern das Ergebnis in einem StringBuffer zwischen
    StringBuffer res = new StringBuffer("||=");
    // ... und iterieren durch die Liste
    for (Iterator it = scheiben.iterator(); it.hasNext(); ) {
      res.insert(2,it.next());  // impliziter Aufruf von toString()
      res.insert(2,'=');
    }
    // Fertig :-)
    return res.toString();
  }
}
```

Das Endergebnis dieser Methode stellt eine Stange auf der Konsole beispielsweise wie folgt dar:

Konsole

```
||=(11)=(10)=(09)=(08)=(07)=
```

5.9.5 Die Klasse `Hanoi`, erster Teil

Nun bleibt nur noch die Implementierung der Klasse `Hanoi` – Spielbrett und Spieler in einem. Wir wollen uns zuerst um das Spielbrett an sich kümmern; der Methode `spiele()` ist ein eigener Abschnitt gewidmet.
Abbildung 5.11 zeigt, dass Instanzen dieser Klasse neben der `zaehler`-Variablen auch eine Assoziation zu genau drei `Stange`-Objekten besitzen. Es stellt sich die Frage, wie wir diese Assoziation realisieren. Für die Beziehung zwischen `Stange` und `Scheibe` haben wir eine Liste verwendet. Sollten wir dies hier auch tun?
Obwohl die Verwendung einer `Collection` durchaus möglich wäre, fällt die Wahl in diesem Falle auf den Einsatz eines Arrays:

```
/** Dieses Programm loest das Tuerme-von-Hanoi-Problem
  * und gibt die Loesung auf dem Bildschirm aus.
  **/

public class Hanoi {

  /** Ein Zaehler fuer die Zuege */
  private int zaehler;

  /** Die drei Tuerme */
  private Stange[] stangen = {
    new Stange(),new Stange(),new Stange()
  };
```

Der Grund ist die spezielle Beziehung zwischen Spielbrett und Stangen. Ein Spielbrett hat *genau* drei Stangen. Es werden keine Stangen hinzugefügt oder weggenommen. Es werden auch keine Stangen ausgetauscht. Die Flexibilität unserer `Collection`-Klassen wird in diesem Fall also nicht benötigt.
Im nächsten Schritt bauen wir einen Konstruktor. Hier verwenden wir die Methode `fuegeEin`, um die erste Stange mit den zu verschiebenden Scheiben zu füllen:

```
/** Konstruktor
  * @param anzahl die Anzahl der Scheiben
  **/
public Hanoi(int anzahl) {
  for (int i = anzahl; i > 0; i--)
    stangen[0].fuegeEin(new Scheibe(i));
}
```

Ferner definieren wir eine `toString`-Methode, mit der wir den Zustand des Bretts auf dem Bildschirm ausgeben werden. Beachten Sie, wie die `toString`-Methode der `Stange`-Objekte aufgerufen wird, die die `toString`-Methode der `Scheibe`-Objekte aufruft...

```
/** Liefert eine String-Darstellung des
  * aktuellen Zustandes.
  * @return die drei Stangen in textueller Darstellung
  **/
public String toString() {
  StringBuffer res = new StringBuffer();
  res.append("Start:       ");
  res.append(stangen[0].toString());
  res.append('\n');
  res.append("Ziel:        ");
  res.append(stangen[1].toString());
  res.append('\n');
  res.append("Hilfsstange: ");
  res.append(stangen[2].toString());
  return res.toString();
}
```

Zu guter Letzt definieren wir noch die `main`-Methode. Hier verwenden wir die Methode `parseInt()` der `Integer`-Hilfsklasse, um die Anzahl der Scheiben von der Kommandozeile einzulesen. Anschließend instantiieren wir das Spielbrett und rufen den Algorithmus auf:

```
/** main-Methode: Erhaelt die Anzahl der Scheiben als
  * Kommandozeilenparameter und startet den Algorithmus.
  **/
public static void main(String[] args) {
  if (args.length != 1) {
    System.out.println("Korrekter Aufruf: Hanoi <Anzahl Scheiben>");
    return;
  }
  int anzahl = Integer.parseInt(args[0]);
  new Hanoi(anzahl).spiele();
}
```

5.9.6 Der Algorithmus

So wie das Achtdamenproblem sind die Türme von Hanoi ein klassisches Beispiel für die Lösung von Problemen mittels rekursiver Methoden. Unser Versuch, die Scheiben vom Start zum Ziel zu transportieren, spielt sich nach folgendem Schema ab:

- Wenn wir nur eine Scheibe verschieben müssen, so tun wir das, andernfalls
- verschieben wir alle bis auf die größte Scheibe auf den Hilfsturm,
- versetzen die größte Scheibe zu ihrem Ziel und
- versetzen danach die anderen Scheiben vom Hilfsturm zu ihrem Ziel.

Hierbei betrachten wir jeweils die `Stange` als Hilfsturm, der für den jeweiligen Rekursionsschritt weder Start noch Ziel ist. In Java-Code sieht diese Schiebeoperation (für eine Scheibe) wie folgt aus:

```
boolean ok = stangen[ziel].fuegeEin(stangen[start].entferne());
```

Der Inhalt der Variable `ok` muss hierbei immer **`true`** sein – sonst haben wir einen Programmierfehler begangen. Seit Version 1.4 bietet Java eine elegante Lösung für die Überprüfung derartiger Annahmen: die so genannten **Assertions**. Fügen wir die Zeile

```
assert ok;
```

nach unserer Verschiebeoperation ein, überprüft Java während des Ablaufs unseres Programms automatisch den Zustand dieser Variablen. Setzt man den Wert auf **`false`**, wird ein so genannter `AssertionError` geworfen. Wir verringern somit die Wahrscheinlichkeit, dass ein Programmierfehler unentdeckt bleibt.
Falls Sie eine ältere Version von Java verwenden oder aus anderen Gründen auf die Assertion verzichten wollen (etwa aus Gründen der Abwärtskompatibilität), so können Sie die Zeile bedenkenlos aus dem Code löschen. Ansonsten gilt es zu beachten, dass Sie beim Übersetzen mit `javac` einen zusätzlichen Parameter angeben müssen, der die Assertions aktiviert:

Konsole

```
javac -source 1.4 Hanoi.java
```

Doch zurück zu unserem Algorithmus. Werfen wir einen Blick auf die vollständige rekursive Methode:

```
/** Der eigentliche (rekursive) Verschiebe-Algorithmus.
  * @param kleinste die kleinste zu verschiebende Scheibe
  * @param groesste die groesste zu verschiebende Scheibe
  * @param start index der Stange, von der wir starten
  * @param ziel index der Stange, auf die wir schieben
  */
private void rekursion(int kleinste,int groesste,int start,int ziel){
  // Falls kleinste == groesste ist, sind wir am Ende der
  // Rekursion (und koennen direkt verschieben)
  if ( kleinste == groesste ) {
    // Die eigentliche Verschiebung
    boolean ok = stangen[ziel].fuegeEin(stangen[start].entferne());
    assert ok;
    // Gib den Zustand der Tuerme aus
    System.out.println("Zug : " + (++zaehler));
    System.out.println(this);
```

```
        System.out.println();
      }
      // Andernfalls muessen wir etwas mehr tun
      else {
        // Schiebe die kleineren Scheiben auf einen anderen Turm
        rekursion(kleinste, groesste - 1, start, 3 - start - ziel);
        // Dann verschiebe die groesste Scheibe
        rekursion(groesste,groesste,start,ziel);
        // Danach schiebe die ganzen kleineren ebenfalls aufs Ziel
        rekursion(kleinste, groesste - 1, 3 - start - ziel, ziel);
      }
    }
```

Beim rekursiven Aufruf unserer Methode verwenden wir einen kleinen Trick, um die jeweils zu verwendende Hilfsstange zu berechnen. Da wir unsere Stangen über den Index innerhalb eines Feldes ansprechen und die Summe der drei Indizes $0+1+2 = 3$ ist, können wir den jeweils verbleibenden Index einfach errechnen, indem wir Start- und Zielindex von der Gesamtsumme 3 abziehen:

```
        rekursion(kleinste, groesste - 1, start, 3 - start - ziel);
```

Natürlich ist unsere rekursive Hilfsmethode nur zum internen Gebrauch bestimmt. Wie wir aber nun sehen, ist die Methode `spiele()` nicht viel mehr als der Aufruf unserer Hilfsmethode mit den richtigen Parametern:

```
    /** Fuehre den "Tuerme von Hanoi"-Algorithmus durch */
    public void spiele() {
      if (zaehler > 0)
        return;
      System.out.println("Ausgangszustand:");
      System.out.println(this);
      System.out.println();
      rekursion(1,stangen[0].getHoehe(),0,1);
    }
```

Somit ist unser Programm komplett. Der folgende Ausdruck zeigt das Ergebnis für drei Scheiben:

Konsole

```
Ausgangszustand:
Start:       ||=(03)=(02)=(01)=
Ziel:        ||=
Hilfsstange: ||=

Zug : 1
Start:       ||=(03)=(02)=
Ziel:        ||=(01)=
Hilfsstange: ||=

Zug : 2
Start:       ||=(03)=
Ziel:        ||=(01)=
Hilfsstange: ||=(02)=
```

```
Zug : 3
Start:       ||=(03)=
Ziel:        ||=
Hilfsstange: ||=(02)=(01)=

Zug : 4
Start:       ||=
Ziel:        ||=(03)=
Hilfsstange: ||=(02)=(01)=

Zug : 5
Start:       ||=(01)=
Ziel:        ||=(03)=
Hilfsstange: ||=(02)=

Zug : 6
Start:       ||=(01)=
Ziel:        ||=(03)=(02)=
Hilfsstange: ||=

Zug : 7
Start:       ||=
Ziel:        ||=(03)=(02)=(01)=
Hilfsstange: ||=
```

5.10 Body-Mass-Index

Der Body-Mass-Index ist eine sehr einfache Kenngröße bezüglich der Frage, ob eine Person über- oder untergewichtig ist. Er berechnet sich gemäß der Formel

$$\text{BMI} = \frac{\text{Gewicht in kg}}{(\text{Körpergröße in m})^2}$$

Tabelle 5.2 zeigt, wie aus dem Body-Mass-Index geschlossen wird, ob eine Person über- oder untergewichtig ist. Es ist zu beachten, dass der BMI für Heranwachsende unter 18 Jahren normalerweise keine Aussagekraft hat.
In diesem Abschnitt wollen wir ein einfach zu bedienendes Programm schreiben, das die Benutzerin bzw. den Benutzer ihren bzw. seinen BMI berechnen lässt und bei zu hohem oder niedrigem Wert eine Warnung anzeigt.

5.10.1 Vorwissen aus dem Buch

Diese Übungsaufgabe behandelt eine Vielzahl neuer Konzepte aus der Programmierung grafischer Oberflächen zusammen:

18 bis 34 Jahre			
BMI: unter 19 Untergewicht	BMI: 19 bis 24 Gesunder Bereich	BMI: 25 bis 30 Übergewicht	BMI: über 30 Fettleibigkeit
35 Jahre und mehr			
BMI: unter 19 Untergewicht	BMI: 19 bis 26 Gesunder Bereich	BMI: 27 bis 30 Übergewicht	BMI: über 30 Fettleibigkeit

Tabelle 5.2: BMI-Verteilung

- Kapitel 13 (Aufbau grafischer Oberflächen in Frames)
- Kapitel 14 (Swing-Komponenten) sowie
- Kapitel 15 (Ereignisverarbeitung).

Falls dies ein bischen viel für eine vereinzelte Aufgabe, mag es eventuell Sinn machen, erst mit Ergänzungskapitel 6 auf Seite 163 zu starten.

5.10.2 Design und Layout

Auch wenn man eigentlich nicht auf das Resultat spicken sollte, bevor man es geschrieben hat, wollen wir hier eine Ausnahme machen. Bitte werfen Sie einen Blick auf Abbildung 5.12.
Wie man sieht, ist das Layout ein wenig an einer Arztwaage orientiert. Anstelle von Eingabefeldern für Gewicht, Alter und Körpergröße verwenden wir drei Schieberegler (`javax.swing.JSlider`). Links von den Schiebereglern bezeichnen wir die einzustellende Größe, rechts zeigen wir den gerade gesetzten Wert. Unter diesen Reglern stellen wir das Ergebnis und eine eventuelle Warnung (bei gefährlich hohem oder niedrigem BMI) dar.
Unsere grafische Oberfläche setzt sich also nur aus einer Anzahl von `Label`- und `JSlider`-Instanzen zusammen, die wir in unserer Klasse als Instanzvariablen definieren:

```
JSlider alter = new JSlider(18,100,25);
JSlider gewicht = new JSlider(40,200,75);
JSlider groesse = new JSlider(120,210,175);
JLabel label1 = new JLabel("Alter (in Jahren): ");
JLabel label2 = new JLabel("Gewicht (in kg): ");
JLabel label3 = new JLabel("Groesse (in cm): ");
JLabel alterAktuell = new JLabel();
JLabel gewichtAktuell = new JLabel();
JLabel groesseAktuell = new JLabel();
JLabel bmi = new JLabel(" ");
JLabel warnung = new JLabel(" ");
```

Bitte beachten Sie, dass wir den `JSlider`-Objekten den Wertebereich (kleinster und größter Wert, den der Schieberegler annehmen kann) im Konstruktor übergeben.
Auch können wir gewisse Grundeinstellungen an den Labeln, etwa bezüglich Farbgebung oder textueller Ausrichtung, bereits vornehmen:

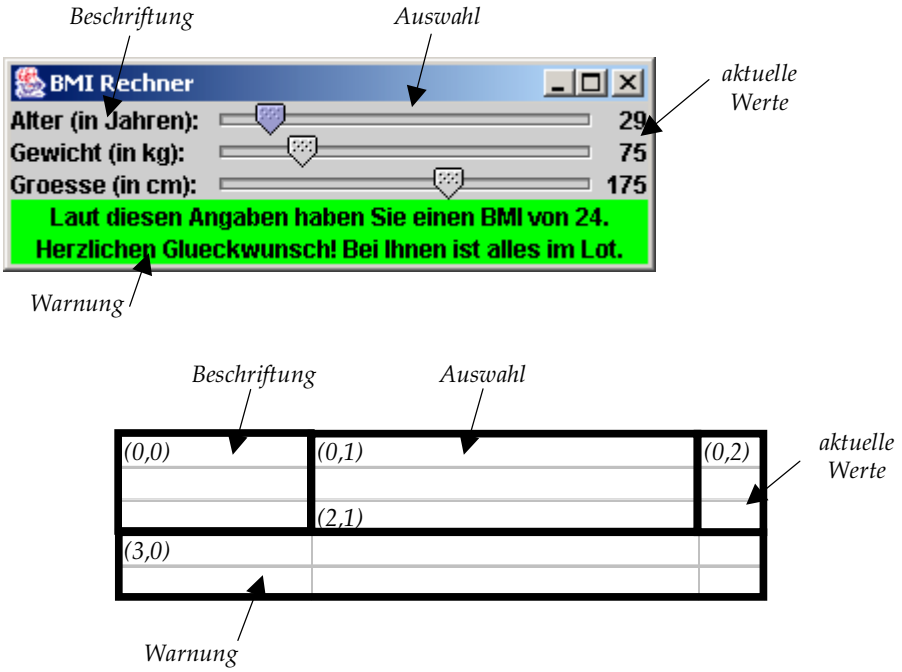


Abbildung 5.12: Das GridBagLayout am Beispiel des BMI-Rechners

```
// Einige Labels wollen sollen farbig sein -
// hierzu muessen wir den opaque-"Schalter" setzen
warnung.setOpaque(true);
bmi.setOpaque(true);
// Der Text in diesen Labeln soll ausserdem zentriert sein
warnung.setHorizontalAlignment(SwingConstants.CENTER);
bmi.setHorizontalAlignment(SwingConstants.CENTER);
// Fuer unsere aktuell gesetzten Werte sollen die Zahlen nach
// rechts ausgerichtet werden.
alterAktuell.setHorizontalAlignment(SwingConstants.RIGHT);
gewichtAktuell.setHorizontalAlignment(SwingConstants.RIGHT);
groesseAktuell.setHorizontalAlignment(SwingConstants.RIGHT);
```

Wie aber bekommen wir das komplexe Design aus Abbildung 5.12 am besten realisiert?
Wenn wir die schematische Aufzeichnung unter dem eigentlichen Bildschirmfoto betrachten, erkennen wir gewisse Ähnlichkeiten mit einer Tabelle. Wenn man einmal vom Warnungsbereich absieht, sind alle `JLabel`- und `JSlider`-Instanzen in einer einfachen Tabelle untergebracht. Alle Objekte einer Spalte haben die gleiche Breite, allerdings können unterschiedliche Spalten unterschiedlich breit sein (sonst wäre es ein Leichtes, das `GridLayout` zu verwenden). Wir benötigen also ein Layout, das

- Objekte in einer „Tabelle“ darstellt,
- die Breite der Spalten auf die Breite des Inhaltes abstimmt,
- unterschiedliche Breiten für unterschiedliche Spalten zulässt und
- ein Objekt (die Warnungen) auf die Breite von mehreren Spalten ziehen kann.

Dies und noch viel mehr kann die Klasse `java.awt.GridBagLayout`. Das `GridBagLayout` ist in seinen Möglichkeiten so vielfältig (und deshalb in seiner Verwendung so komplex), dass für jedes mit dem Layout eingefügte Objekt eine Vielzahl von Optionen und Parameter gesetzt werden müssen. In welche Zeile und Spalte soll das Objekt? Soll es so hoch bzw. breit sein wie die anderen Objekte in derselben Zeile bzw. Spalte? Wenn nicht, wie soll es innerhalb der Zelle ausgerichtet werden? Wie viel Abstand zu seinen Nachbarn soll das Objekt haben?
Damit sich der Programmierer nicht um alle möglichen Optionen scheren muss, haben die Entwickler von Java alle Optionen in ein eigenes Objekt ausgelagert – die Klasse `GridBagConstraints`. Eine Instanz der Klasse `GridBagConstraints` ist bereits mit einer Menge von Standardwerten voreingestellt. Wir müssen nur noch verändern, was vom Standard abweicht. In unserem Fall sind dies die folgenden Instanzvariablen:[13]

- Die Variablen `gridx` und `gridy`, die die Spalten- bzw. Zeilennummer des darzustellenden Objektes angeben. Es wird immer von Null an gezählt.
- Die Variable `gridwidth`, die die Anzahl der Spalten angibt, die ein darzustellendes Objekt einnimmt.
- Die Variable `fill`, die wir auf `GridBagConstraints.HORIZONTAL` setzen. Dies besagt, dass das darzustellende Objekt immer so weit gestreckt wird, dass alle Objekte in derselben Spalte gleich weit sind.

Diese Schalter reichen vollkommen aus, um für unseren Entwurf alle nötigen Einstellungen zu setzen. Für die Vielzahl weiterer Einstellungen sei auf die Dokumentation der entsprechenden Klasse verwiesen.
Um ein Objekt mit Hilfe des `GridBagLayout` darzustellen, gehen wir wie folgt vor:

- Zuerst erzeugen wir ein `GridBagConstraints`-Objekt, in dem wir alle notwendigen Optionen setzen.
- Danach verwenden wir eine Methode namens `setConstraints` im GridBagLayout, um die Einstellungen für die Komponente zu registrieren.
- Schließlich fügen wir die Komponente wie gewohnt mit `add` in den umschließenden Container ein.

[13] Der OO-gestählte Leser wird hier gleich bemerken, dass das Setzen von Instanzvariablen ohne get- und set-Methoden nicht gerade ideal im Sinne der Datenkapselung ist. Die Autoren geben diesen Lesern voll und ganz recht.

Übrigens sind `GridBagConstraints`-Objekte wiederverwertbar. Nachdem die `setConstraints`-Methode aufgerufen wurde, können die Einstellungen für das nächste darzustellende Objekt einfach überschrieben werden.
Nun aber zu unserem Code. Wir verwenden eine Schleife, um die verschiedenen Zeilen unseres Layouts zu setzen:

```
GridBagLayout gridbag = new GridBagLayout();
setLayout(gridbag);
GridBagConstraints constraints = new GridBagConstraints();
// Das Layout soll keine "Luecken" hinterlassen,
// sondern alle Komponenten einer Spalte so weit strecken,
// dass sie gleich weit sind
constraints.fill = GridBagConstraints.HORIZONTAL;
// Setze die einzelnen Eingabefelder in einer Schleife
JSlider[] sliders = {alter,gewicht,groesse};
JLabel[] labels = {label1,label2,label3};
JLabel[] aktuell = {alterAktuell,gewichtAktuell,groesseAktuell};
for (int i = 0; i < 3; i++) {
  // Setze die Zeile, in der wir uns befinden
  constraints.gridy = i;
  // Setze den Label
  constraints.gridx = 0;
  gridbag.setConstraints(labels[i],constraints);
  this.add(labels[i]);
  // Setze den Slider
  constraints.gridx = 1;
  gridbag.setConstraints(sliders[i],constraints);
  this.add(sliders[i]);
  // Setze die Darstellung der gesetzten Werte
  constraints.gridx = 2;
  gridbag.setConstraints(aktuell[i],constraints);
  this.add(aktuell[i]);
}
```

Zu guter Letzt setzen wir noch die Label für unsere Warnungen. Wie bereits erwähnt, verwenden wir den Schalter `gridwidth`, um unsere Label über alle drei Spalten zu strecken:

```
constraints.gridx = 0;
constraints.gridy = 3;
constraints.gridwidth = 3;
gridbag.setConstraints(bmi,constraints);
this.add(bmi);
constraints.gridy = 4;
gridbag.setConstraints(warnung,constraints);
this.add(warnung);
```

Somit ist unser Layout fertig. Wir müssen das Fenster also nur noch mit der entsprechenden Anwendungslogik versehen.

5.10.3 Events und Anwendungslogik

Im Falle des BMI-Rechners ist ziemlich klar, was unser Programm eigentlich bewerkstelligen soll. Sobald einer der Schieberegler bewegt wird, soll es den Body-

Mass-Index errechnen und mit dem Inhalt von Tabelle 5.2 vergleichen. Abhängig vom Ergebnis sollen der BMI und eine eventuelle Warnung ausgegeben werden. Bevor wir die eigentliche Logik realisieren, wollen wir die Werte aus der Tabelle in unser Programm übernehmen. Wir definieren ein zweidimensionales Feld:

```
private final static int[][] GRENZEN =
  {{19,25,30},{19,27,30}};
```

Jede Zeile des Feldes stellt die Grenzen aus der Tabelle für eine entsprechende Altersstufe dar. Die Indices entsprechen denen der wie folgt definierten Warnungstexte und Farben:

```
// Die Nachricht fuer die entsprechende Gewichtsgrenze
private final static String[] WARNUNGEN =
{ "Laut BMI sind Sie untergewichtig. Bitte essen Sie mehr!",
  "Herzlichen Glueckwunsch! Bei Ihnen ist alles im Lot.",
  "Laut BMI sind Sie uebergewichtig. Bitte tun Sie etwas!",
  "Suchen Sie bitte einen Arzt auf!!!"
};

// Die Farben, in denen die Warnungen dargestellt werden sollen
private final static Color[] FARBEN =
  {Color.RED,Color.GREEN,Color.YELLOW,Color.RED};
```

Kommen wir nun zu unserer Anwendungslogik. Wir lesen die gesetzten Werte aus unseren Schiebereglern mit der Instanzmethode `getValue` aus und aktualisieren hiermit unsere Darstellung. Anschließend berechnen wir den Body-Mass-Index und stellen diesen im Label `bmi` dar. Zu guter Letzt bestimmen wir anhand der Tabelle die darzustellende Warnung und Farbe und leiten sie an `warnung` weiter:

```
/** Diese Methode wird aufgerufen,
  * wann immer ein Slider bewegt wird
  **/
private void setBMI() {
  // Aktualisiere die gesetzten Werte in den Labeln
  alterAktuell.setText(String.valueOf(alter.getValue()));
  gewichtAktuell.setText(String.valueOf(gewicht.getValue()));
  groesseAktuell.setText(String.valueOf(groesse.getValue()));
  // Berechne den BMI und setze das Label
  int berechnet = (int) Math.round(gewicht.getValue() / Math.pow(
    groesse.getValue() / 100.0, 2));
  bmi.setText("Laut diesen Angaben haben Sie einen BMI von "
    + berechnet + ".");
  // Nun finde die Warnung, die zu diesem BMI passt
  int kategorie = alter.getValue() < 35 ? 0 : 1;
  int index = 0;
  for (; index < 3; index++)
    if (berechnet < GRENZEN[kategorie][index])
      break;
  // Setze den entsprechenden Text und die Farbe
  warnung.setText(WARNUNGEN[index]);
  warnung.setBackground(FARBEN[index]);
  bmi.setBackground(FARBEN[index]);
}
```

Unsere Anwendungslogik ist somit in der Methode `setBMI` definiert. Alles, was uns noch verbleibt, ist, sie immer dann aufzurufen, wenn einer unserer Schieberegler bewegt wird:

```
ChangeListener listener = new ChangeListener() {
  public void stateChanged(ChangeEvent e) {
    setBMI();
  }
};
alter.addChangeListener(listener);
gewicht.addChangeListener(listener);
groesse.addChangeListener(listener);
```

Beim Verändern eines `JSlider` wird ein so genanntes `ChangeEvent` geworfen, das wir mit unserem `ChangeListener` auffangen. Beachten Sie, dass wir auf den Inhalt des Events hier überhaupt keinen Wert gelegt haben. Wie so oft ist auch hier der Umstand, *dass* das Event ausgelöst wurde, der einzig interessante Punkt.

5.10.4 Das gesamte Programm im Überblick

Zu guter Letzt wollen wir noch einmal einen Blick auf das komplette Programm werfen. Schenken Sie insbesondere der Aufteilung in verschiedene Methoden Beachtung. In einer Zeit, in der GUI-Programmierung immer häufiger durch visuelle Editoren statt durch Handarbeit erledigt wird, sollten wir uns eine Trennung von Darstellungs- und Anwendungslogik selbst innerhalb ein und derselben Klasse angewöhnen.

```
1  import javax.swing.*;
2  import java.awt.*;
3  import javax.swing.event.*;
4
5  /** Einfache Berechnung des Body-Mass-Index  */
6  public class BMI extends JPanel {
7
8    // Verwendete grafische Komponenten
9    JSlider alter = new JSlider(18,100,25);
10   JSlider gewicht = new JSlider(40,200,75);
11   JSlider groesse = new JSlider(120,210,175);
12   JLabel label1 = new JLabel("Alter (in Jahren): ");
13   JLabel label2 = new JLabel("Gewicht (in kg): ");
14   JLabel label3 = new JLabel("Groesse (in cm): ");
15   JLabel alterAktuell = new JLabel();
16   JLabel gewichtAktuell = new JLabel();
17   JLabel groesseAktuell = new JLabel();
18   JLabel bmi = new JLabel(" ");
19   JLabel warnung = new JLabel(" ");
20
21   // Die Grenzen fuer Unter- und Uebergewicht
22   private final static int[][] GRENZEN =
23     {{19,25,30},{19,27,30}};
24
25   // Die Nachricht fuer die entsprechende Gewichtsgrenze
26   private final static String[] WARNUNGEN =
```

```
27    { "Laut BMI sind Sie untergewichtig. Bitte essen Sie mehr!",
28      "Herzlichen Glueckwunsch! Bei Ihnen ist alles im Lot.",
29      "Laut BMI sind Sie uebergewichtig. Bitte tun Sie etwas!",
30      "Suchen Sie bitte einen Arzt auf!!!"
31    };
32
33    // Die Farben, in denen die Warnungen dargestellt werden sollen
34    private final static Color[] FARBEN =
35      {Color.RED,Color.GREEN,Color.YELLOW,Color.RED};
36
37    /** Hilfsmethode fuer den Konstruktor: Setze das Design */
38    private void setDesign() {
39      // Einige Labels wollen sollen farbig sein -
40      // hierzu muessen wir den opaque-"Schalter" setzen
41      warnung.setOpaque(true);
42      bmi.setOpaque(true);
43      // Der Text in diesen Labeln soll ausserdem zentriert sein
44      warnung.setHorizontalAlignment(SwingConstants.CENTER);
45      bmi.setHorizontalAlignment(SwingConstants.CENTER);
46      // Fuer unsere aktuell gesetzten Werte sollen die Zahlen nach
47      // rechts ausgerichtet werden.
48      alterAktuell.setHorizontalAlignment(SwingConstants.RIGHT);
49      gewichtAktuell.setHorizontalAlignment(SwingConstants.RIGHT);
50      groesseAktuell.setHorizontalAlignment(SwingConstants.RIGHT);
51      // Verwende das GridBagLayout
52      GridBagLayout gridbag = new GridBagLayout();
53      setLayout(gridbag);
54      GridBagConstraints constraints = new GridBagConstraints();
55      // Das Layout soll keine "Luecken" hinterlassen,
56      // sondern alle Komponenten einer Spalte so weit strecken,
57      // dass sie gleich weit sind
58      constraints.fill = GridBagConstraints.HORIZONTAL;
59      // Setze die einzelnen Eingabefelder in einer Schleife
60      JSlider[] sliders = {alter,gewicht,groesse};
61      JLabel[] labels = {label1,label2,label3};
62      JLabel[] aktuell = {alterAktuell,gewichtAktuell,groesseAktuell};
63      for (int i = 0; i < 3; i++) {
64        // Setze die Zeile, in der wir uns befinden
65        constraints.gridy = i;
66        // Setze den Label
67        constraints.gridx = 0;
68        gridbag.setConstraints(labels[i],constraints);
69        this.add(labels[i]);
70        // Setze den Slider
71        constraints.gridx = 1;
72        gridbag.setConstraints(sliders[i],constraints);
73        this.add(sliders[i]);
74        // Setze die Darstellung der gesetzten Werte
75        constraints.gridx = 2;
76        gridbag.setConstraints(aktuell[i],constraints);
77        this.add(aktuell[i]);
78      }
79      // Zu guter Letzt setze die Ergebnislabel - und mache
80      // sie so breit wie das Fenster
81      constraints.gridx = 0;
```

```
82      constraints.gridy = 3;
83      constraints.gridwidth = 3;
84      gridbag.setConstraints(bmi,constraints);
85      this.add(bmi);
86      constraints.gridy = 4;
87      gridbag.setConstraints(warnung,constraints);
88      this.add(warnung);
89    }
90
91    /** Hilfsmethode fuer den Konstruktor: Setze benoetigte Listener */
92    private void setListeners() {
93      // Bei jedem Wechsel soll die setBMI-Methode aufgerufen werden.
94      ChangeListener listener = new ChangeListener() {
95        public void stateChanged(ChangeEvent e) {
96          setBMI();
97        }
98      };
99      alter.addChangeListener(listener);
100     gewicht.addChangeListener(listener);
101     groesse.addChangeListener(listener);
102   }
103
104   /** Diese Methode wird aufgerufen,
105     * wann immer ein Slider bewegt wird
106     **/
107   private void setBMI() {
108     // Aktualisiere die gesetzten Werte in den Labeln
109     alterAktuell.setText(String.valueOf(alter.getValue()));
110     gewichtAktuell.setText(String.valueOf(gewicht.getValue()));
111     groesseAktuell.setText(String.valueOf(groesse.getValue()));
112     // Berechne den BMI und setze das Label
113     int berechnet = (int) Math.round(gewicht.getValue() / Math.pow(
114       groesse.getValue() / 100.0, 2));
115     bmi.setText("Laut diesen Angaben haben Sie einen BMI von "
116       + berechnet + ".");
117     // Nun finde die Warnung, die zu diesem BMI passt
118     int kategorie = alter.getValue() < 35 ? 0 : 1;
119     int index = 0;
120     for (; index < 3; index++)
121       if (berechnet < GRENZEN[kategorie][index])
122         break;
123     // Setze den entsprechenden Text und die Farbe
124     warnung.setText(WARNUNGEN[index]);
125     warnung.setBackground(FARBEN[index]);
126     bmi.setBackground(FARBEN[index]);
127   }
128
129   /** Konstruktor */
130   public BMI() {
131     setDesign();
132     setListeners();
133     setBMI();
134   }
135
136   /** Main-Methode */
```

```
137     public static void main(String[] args) {
138       JFrame frame = new JFrame();
139       frame.setTitle("BMI Rechner");
140       frame.setContentPane(new BMI());
141       frame.pack();
142       frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
143       frame.setVisible(true);
144     }
145   }
```

Ergänzung 6

Praxisbeispiele: Wem die Stunde schlägt

6.1 Aller Anfang ist leicht

Kapitel 5 befasste sich mit einzelnen, „aus dem Leben“ gegriffenen Aufgaben, und wie wir sie mit den Konstrukten aus der Programmiersprache Java lösen können. In der Praxis ist es natürlich recht selten, dass wir mit einem derart isolierten Problem zu kämpfen haben. Unsere Programme sind oftmals komplex, umspannen eine Vielzahl von Anforderungen, und benötigen mehr Code als sich auf den Seiten eines Lehrbuches sinnvoll drucken lässt. Es wird „Zeit“, dem Rechnung zu tragen: aus diesem Grund werden wir uns in diesem Kapitel weniger mit vereinzelten Aufgaben als mit einem Projekt beschäftigen – ein größeres Programm, welches wir in eine Vielzahl kleinerer Einzelschritte zerlegen.
Um es vorwegzunehmen: wenn das Kapitel zu Ende ist, werden wir eine Applikation geschrieben haben, die

- eine Digital- oder Analoguhr grafisch auf dem Bildschirm darstellt,
- die Zeit im Sekundentakt aktualisiert und sogar in der Lage ist,
- sich sowohl lokal auf dem Rechner als auch über das Internet starten zu lassen und
- sich mit Hilfe eines Zeit-Servers im Netzwerk selbst zu stellen.

Da dieses Vorhaben recht groß ist, teilen wir unser Projekt in verschiedene Schritte auf. Jeder einzelne Schritt wird in einem funktionsfähigen Programm resultieren, das uns dem eigentlichen Ziel einen kleinen Schritt näher bringt. Dieser Ansatz, die so genannte **Iterative Entwicklung**, ist heutzutage in vielen kommerziellen Projekten üblich. Sie ist Bestandteil verschiedener Entwicklungsstile, beispielsweise des **Extreme Programming** oder des **Agile Development**.

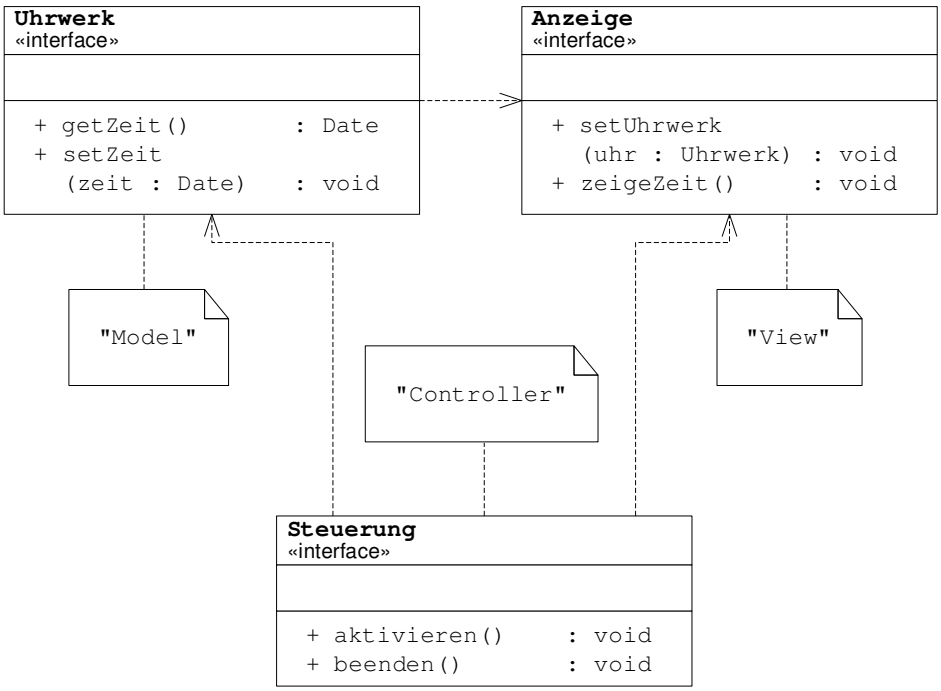


Abbildung 6.1: Aufbau einer Uhr – in Klassen gesehen

Beginnen wir also zunächst klein: in diesem Abschnitt werden wir ein Programm namens `WieSpaet` schreiben, das die Systemzeit erfragt und formatiert in der Form

Konsole
```
Es ist gerade 14:16 Uhr und 04 Sekunden.
```

auf dem Bildschirm ausgibt. Klingt einfach? Sollte es eigentlich auch sein. Dennoch werden wir vor Ende des Abschnittes insgesamt sage und schreibe *sechs* `.java`-Dateien formuliert haben, um diese simple Aufgabe zu lösen[1].

6.1.1 Designphase

Lösen wir uns einen Moment lang von der konkreten Aufgabe. Wie funktioniert eigentlich eine Digitaluhr?
Da wäre zum einen natürlich die LED-Anzeige. Diese Anzeige stellt dem Benutzer eine „grafische Oberfläche“ zur Verfügung, von der er die Zeit ablesen kann.

[1] Das Grundproblem wäre mit weniger Code lösbar. Es geht uns aber darum, die Grundlage für komplexere Uhrenprogramme zu schaffen. Für mehr Details sei auch auf die Ausführungen am Ende dieser Iteration in Abschnitt 6.1.4 auf Seite 168 verwiesen.

Die Anzeige weiß natürlich nicht von alleine, wie spät es ist. Diese Informationen erhält sie von einem Uhrwerk, dass innerhalb des Gehäuses die Sekunden zählt. Dieses Uhrwerk liefert die genaue Zeit zurück und kann (falls es sich nicht um eine Funkuhr handelt) vom Träger der Uhr auch gestellt werden. Zwischen Uhrwerk und Anzeige befindet sich eine wie auch immer geartete Logik, die die Abläufe zwischen Uhrwerk und Anzeige aktiv steuert.
In unserem Modell (Abbildung 6.1) bilden wir diese drei Bestandteile einer Uhr – `Uhrwerk` (Model), `Anzeige` (View) und `Steuerung` (Controller) – mit Hilfe von Interfaces nach, wir benutzen also das Model-View-Controller-Pattern (vergleiche auch Kapitel 4). In Java-Code sehen diese Interfaces wie folgt aus:

```
1  import java.util.Date;
2
3  /** Dieses Interface stellt das Zeitmessungsinstrument
4    * einer Uhr dar.
5    **/
6  public interface Uhrwerk {
7
8    /** Gibt die aktuelle Uhrzeit in Form eines Date-Objektes
9      * zurueck.
10     * @return die aktuelle Zeit
11     **/
12   public Date getZeit();
13
14   /** Stellt das Werk dieser Uhr auf eine bestimmte Zeit.
15     * Diese Methode ist optional und muss nicht immer
16     * implementiert sein.
17     * @param zeit die aktuelle Uhrzeit
18     * @exception UnsupportedOperationException falls diese Methode
19     *  nicht unterstuetzt wird
20     **/
21   public void setZeit(Date zeit) throws UnsupportedOperationException;
22 }
```

```
1  /** Dieses Interface repraesentiert die Darstellung der
2    * Uhrzeit gemaess einem gegebenen Uhrwerk.
3    **/
4  public interface Anzeige {
5
6    /** Nennt der Anzeige das Uhrwerk, das verwendet werden
7      * soll.
8      * @param uhr das zu verwendende Uhrwerk
9      **/
10   public void setUhrwerk(Uhrwerk uhr);
11
12   /** Weist die Anzeige an, die aktuelle Uhrzeit aus dem
13     * Uhrwerk auszulesen und darzustellen.
14     **/
15   public void zeigeZeit();
16
17 }
```

```
1  /** Implementierungen dieses Interfaces koordinieren die
```

```
2     * Interaktion zwischen dem Uhrwerk und der Zeitanzeige.
3     * Waehrend die anderen Interfaces passive Klassen
4     * darstellen, ist die Steuerung das aktive Glied zwischen
5     * "Model" und "View"
6     **/
7   public interface Steuerung {
8
9     /** Diese Methode signalisiert der Steuerung, dass
10      * sie mit ihrer Arbeit beginnen soll.
11      **/
12    public void aktivieren();
13
14    /** Diese Methode signalisiert der Steuerung, dass
15      * sie ihre Arbeit jetzt beenden kann.
16      **/
17    public void beenden();
18
19  }
```

Die konkreten Implementierungen dieser Klassen sind austauschbar. Ob wir (wie in dieser Stufe) die Zeit lediglich auf der Konsole ausgeben oder (wie im weiteren Verlauf des Buchs) eine grafische Anzeige vorziehen, ist für das Konzept eher zweitrangig. Auf diese Weise kann unser Programm wachsen, indem wir lediglich einzelne Bestandteile der konkreten Ausprägungen eines Interface verändern.
Übrigens noch ein Hinweis für die Freunde von Entwurfsmustern: diese Art von Design hat im Vokabular von Software-Entwicklern einen festen Namen, den Sie bereits kennen. Ersetzen Sie `Uhrwerk` durch *Model*, Anzeige durch *View* und `Steuerung` durch *Controller*, so haben wir einen klassischen Fall des bereits zuvor erwähnten MVC-Patterns.

6.1.2 Modell und View

Beginnen wir mit der Klasse `Systemzeit`, einer einfachen Realisierung des `Uhrwerk`-Interface. Diese Klasse verwendet die interne Uhr des Computers, um die Zeit zu messen. Da Java keine direkte Manipulation der Systemzeit erlaubt, ist ein Stellen der inneren Uhr nicht möglich:

```
1   import java.util.Date;
2
3   /** Diese Implementierung des Uhrwerk-Interfaces erhaelt ihre
4     * Zeitinformationen von der lokalen Uhr des Computers.
5     * Entsprechend ist es auch nicht moeglich, die Zeit dieses
6     * Uhrwerks vor- oder zurueckzustellen.
7     **/
8   public class Systemzeit implements Uhrwerk {
9
10    /** Gibt die aktuelle Uhrzeit in Form eines Date-Objektes
11      * zurueck.
12      * @return die aktuelle Zeit
13      **/
14    public Date getZeit() {
15      return new Date();
```

```
16    }
17
18    /** Stellt das Werk dieser Uhr auf eine bestimmte Zeit.
19      * Diese Methode ist optional und muss nicht immer
20      * implementiert sein.
21      * @param zeit die aktuelle Uhrzeit
22      * @exception UnsupportedOperationException falls diese Methode
23      *  nicht unterstuetzt wird
24      **/
25    public void setZeit(Date zeit) throws UnsupportedOperationException {
26      throw new UnsupportedOperationException();
27    }
28
29  }
```

Auch unser View, die Klasse `KonsolenAnzeige`, ist nicht wesentlich komplexer. Wir verwenden ein `SimpleDateFormat`-Objekt, um die Formatierung unserer Zeit auf dem Bildschirm zu erzielen:

```
private static SimpleDateFormat FORMAT = new SimpleDateFormat
  ("'Es ist gerade' HH:mm 'Uhr und' ss 'Sekunden.'");
```

Diese Klassenvariable verwenden wir in der Methode `zeigeZeit`, um die vom `Uhrwerk` erfragte Zeit auszugeben:

```
/** Weist die Anzeige an, die aktuelle Uhrzeit aus dem
  * Uhrwerk auszulesen und darzustellen.
  **/
public void zeigeZeit() {
  System.out.println(FORMAT.format(uhrwerk.getZeit()));
}
```

Das Uhrwerk selbst haben wir in einer Instanzvariablen gesichert, die wir mit der Methode `setUhrwerk()` manipulieren können:

```
/** Dieses Uhrwerk wird fuer die Zeitdarstellung verwendet. */
private Uhrwerk uhrwerk;

/** Nennt der Anzeige das Uhrwerk, das verwendet werden
  * soll.
  * @param uhr das zu verwendende Uhrwerk
  **/
public void setUhrwerk(Uhrwerk uhr) {
  uhrwerk = uhr;
}
```

6.1.3 Controller und Hauptprogramm

Kommen wir nun zur Klasse `WieSpaet`, die unser Steuerungsinterface realisiert. Auch diese Klasse ist alles andere als komplex – wir befinden uns ja auch in der allerersten Iteration unseres Mini-Projektes.

```
1  /** Eine einfache Uhr-Steuerung mit Main-Methode:
2    * zeige die aktuelle Systemzeit an und beende
```

```
3     * das Programm
4     **/
5   public class WieSpaet implements Steuerung {
6
7     /** Die verwendete Anzeige */
8     private Anzeige anzeige;
9
10    /** Konstruktor.
11      * @param uhrwerk das zu verwendende Uhrwerk
12      * @param anzeige die zu verwendende Anzeige
13      **/
14    public WieSpaet(Uhrwerk uhrwerk,Anzeige anzeige) {
15      this.anzeige = anzeige;
16      // Gehe sicher, dass das Uhrwerk gesetzt ist
17      anzeige.setUhrwerk(uhrwerk);
18    }
19
20    /** Diese Methode signalisiert der Steuerung, dass
21      * sie mit ihrer Arbeit beginnen soll.
22      **/
23    public void aktivieren() {
24      anzeige.zeigeZeit();
25    }
26
27    /** Diese Methode signalisiert der Steuerung, dass
28      * sie ihre Arbeit jetzt beenden kann.
29      **/
30    public void beenden() {
31      // Keine besonderen Aktionen notwendig
32    }
33
34    /** Main-Methode: zeigt die Zeit einmal an */
35    public static void main(String[] args) {
36      // Instantiiere ein einzelnes Steuerungsobjekt
37      Steuerung steuerung =
38        new WieSpaet(new Systemzeit(),new KonsolenAnzeige());
39      // Aktiviere die Steuerung und gebe somit die Zeit aus
40      steuerung.aktivieren();
41      // Beende die Steuerung und das Programm
42      steuerung.beenden();
43    }
44
45  }
```

6.1.4 Ausblick

Werfen wir einen Blick auf das folgende Programm:

```
1   import java.text.SimpleDateFormat;
2   import java.util.Date;
3   /** Gibt die aktuelle Uhrzeit auf dem Bildschirm aus. */
4   public class OhneSchnoerkel {
5     /** Main-Methode */
6     public static void main(String[] args) {
```

```
7       System.out.println(new SimpleDateFormat(
8         ("'Es ist gerade' HH:mm 'Uhr und' ss 'Sekunden.'"))
9         .format(new Date()));
10    }
11  }
```

Es bewirkt genau dasselbe, was wir gerade mit drei Interfaces und ebenso vielen Klassen mühsam erzielt haben. Wozu also der ganze Aufwand?
Diese Frage ist wie so oft auch diesmal nicht in einem Satz zu beantworten. Wir können natürlich auf unser „höheres Ziel" verweisen und auf die langfristige Perspektive unseres Modells bauen. Doch was hilft das dem Anwender, der *wirklich* nur die Zeit auf dem Bildschirm ausgegeben haben möchte?
Wenn wir diesen Fall aus kommerzieller Sicht betrachten, so ist unsere Implementierung natürlich ein Fiasko. Wir haben ein Vielfaches an Code und Zeit investiert und somit die Gewinnspanne unseres Projektes deutlich verringert. Wenn wir also wissen, dass die Anforderungen unseres Kunden feststehen, so ist der Ansatz „ohne Schnörkel" durchaus zu vertreten.
Doch nehmen wir einmal an, der Kunde ist mit seiner Anwendung so zufrieden, dass er einen Folgeauftrag vergibt. Statt die Uhrzeit nur zu sehen, möchte er sie mit dem Programm auch setzen können. Oder vielleicht möchte er auch herausfinden, wie die Systemzeit gerade auf einem *anderen* Computer im Firmennetzwerk gesetzt ist. Für keine dieser Anforderungen ist Raum im Ansatz des schnörkellosen Programms vorhanden. Jeder Folgeauftrag bedeutet also eine komplette Neuentwicklung; die Wiederverwertung von bereits verwendetem Code ist somit gleich null. Auf lange Sicht verliert das Unternehmen mit diesem Ansatz also mehr, als es gewinnt.
Natürlich wirken diese Fälle bei einem so kleinen Beispiel arg konstruiert, aber sie sind weniger weit von der Praxis entfernt, als man vielleicht annimmt. Viele kleine und mittelständische Softwarehäuser leben von Auftragsarbeit, konzentrieren sich auf einen Marktsektor und versuchen, zu *dem* Experten für diese Branche zu werden. Für diese Firmen ist es überlebenswichtig, Lösungen aus der Schublade und mit minimalem Entwicklungsaufwand generieren zu können. Um konkurrenzfähig zu bleiben, müssen sie oft Projektpreise unter den Entwicklungskosten anbieten und ihre Ausgaben auf mehrere Projekte umverteilen. Ihre Existenz hängt also davon ab, dass ein Großteil ihres Codes wiederverwertbar und in mehr als einem Projekt einsetzbar ist.

6.2 Iteration 2: Eine Digitalanzeige

Bevor wir unser ursprüngliches Problem von von Seite 163 weiter ausbauen, fassen wir die ersten Schritte noch einmal zusammen. Wir haben uns viel vorgenommen – die Realisierung einer Zeitansage (wahlweise analog oder digital) mit diversen mehr oder minder anspruchsvollen Features. Mit Hilfe eines iterativen Ansatzes wollen wir, beginnend von einer einfachen Zeitausgabe auf der Konsole, zu einer ausgereiften Anwendung gelangen. Unsere ersten Schritte waren:

- ein flexibler Entwurf (basierend auf dem MVC-Pattern), der uns eine schrittweise Entwicklung erlaubt (Design), und
- eine Konsolen-basierte Rumpfimplementierung (Iteration 1), auf der wir in den weiteren Praxiskapiteln aufbauen wollten.

Mit unserem Wissen über Swing-Programmierung können wir es nun wagen, an die grafische Umsetzung unseres Vorsatzes zu gehen. In diesem Abschnitt werden wir die Applikation so erweitern, dass

- die Zeit statt auf der Konsole in einem eigenen Fenster dargestellt wird,
- die Darstellung der Zeit wie eine Digitaluhr aussieht und
- das Fenster ein Menue besitzt, mit dem sich die Anwendung beenden lässt.

Sollte Ihnen der Umgang mit den Swing Klassen ncch nicht vertraut sein, ist jetzt wahrscheinlich ein guter Zeitpunkt, dies nachzulesen.

6.2.1 Jetzt wird's grafisch!

Beginnen wir mit der eigentlichen Digitalanzeige. Um diese in unserem Design verwenden zu können, muss unsere Klasse das Interface `Anzeige` implementieren. Wie aber bewerkstelligen wir die grafische Repräsentierung?
Die Darstellung einer Uhrzeit im digitalen Format ist nicht viel mehr als die Anzeige einer Zeile Text. Für die Darstellung einer Textzeile verwendet man der Einfachheit halber eine `Label`-Subklasse. In Kombination mit einer `DateFormat`-Instanz ist die formatierte Darstellung also eine Leichtigkeit:

```
1  import javax.swing.JLabel;
2  import java.awt.Font;
3  import java.awt.Color;
4  import java.text.SimpleDateFormat;
5
6  /** Digitalanzeige: Stellt die Uhrzeit in Form einer digitalen
7    * Anzeige dar.
8    **/
9  public class DigitalAnzeige extends JLabel implements Anzeige {
10
11   /** Dieses Format-Objekt wird fuer die textuelle Darstellung
12     * der Uhrzeit verwendet.
13     **/
14   private static SimpleDateFormat FORMAT = new SimpleDateFormat
15     ("HH:mm:ss");
16
17   /** Dieses Uhrwerk wird fuer die Zeitdarstellung verwendet. */
18   private Uhrwerk uhrwerk;
19
20   /** Konstruktor */
21   public DigitalAnzeige()
22   {
23     // Setze einen Standard-Text fuer das Label
24     setFont(new Font("Monospaced",Font.BOLD,30));
```

```
25      setText("00:00:00");
26      // Veraendere das Aussehen
27      setOpaque(true);
28      setBackground(Color.BLACK);
29      setForeground(Color.GREEN);
30    }
31
32    /** Nennt der Anzeige das Uhrwerk, das verwendet werden
33      * soll.
34      * @param uhr das zu verwendende Uhrwerk
35      **/
36    public void setUhrwerk(Uhrwerk uhr) {
37      uhrwerk = uhr;
38    }
39
40    /** Weist die Anzeige an, die aktuelle Uhrzeit aus dem
41      * Uhrwerk auszulesen und darzustellen.
42      **/
43    public void zeigeZeit() {
44      setText(FORMAT.format(uhrwerk.getZeit()));
45    }
46
47  }
```

Beachten Sie an diesem Programmcode folgende Besonderheiten:

- Anstatt ganze Pakete zu importieren, haben wir in den **`import`**-Anweisungen lediglich bestimmte Klassennamen angegeben:

  ```
  import javax.swing.JLabel;
  import java.awt.Font;
  import java.awt.Color;
  import java.text.SimpleDateFormat;
  ```

  Manche Softwareentwickler ziehen diese Schreibweise vor, da sie somit genau wissen, *welche* Klassen sie aus welchen Paketen beziehen. Ein weiterer Vorteil dieser Schreibweise ist, dass bei gleichen Klassennamen in unterschiedlichen Paketen (etwa `java.util.Date` und `java.sql.Date`) keine Doppeldeutigkeiten entstehen.

- Digitalanzeigen sind in den seltensten Fällen schwarz auf grauem Hintergrund. Wir setzen deshalb explizite Farben, um unserer Uhr den typischen Radiowecker-Look zu geben:

  ```
  setOpaque(true);
  setBackground(Color.BLACK);
  setForeground(Color.GREEN);
  ```

Wir werden diese Anzeige nun verwenden, um unsere grafische Uhr zusammenzubauen.

6.2.2 Eine neue `Steuerung`

Natürlich reicht die Definition einer neuen `Anzeige` nicht aus, um unsere Uhr auf magische Weise auf dem Bildschirm erscheinen zu lassen. Unsere momentane Steuerungsklasse ist noch sehr einfach und fuer die Verwendung eines grafischen Displays nicht ausgelegt. Wir werden dieses Versäumnis nun nachholen.
Unsere Klasse `SwingUhr` implementiert das Interface `Steuerung` und kann somit die zuvor entworfene Digitalanzeige kontrollieren. Im weiteren Verlauf der Iterationen werden wir die Klasse um diverse weitere Steuerungsroutinen erweitern (etwa den Wechsel zwischen digitaler und analoger Darstellung), so dass die Klasse zentraler Einstiegspunkt für unsere Anwendung werden kann. Unsere `SwingUhr` soll vielseitig einsetzbar sein – ob als eigenständige Applikation oder als Komponente in einem größeren Programm (Stichwort Wiederverwendbarkeit), sollte keine Rolle spielen. Wir leiten die Klasse deshalb von `JPanel` ab und erlauben dem Programmierer somit, diese Swing-Komponente in einer Vielzahl von Anwendungen mit anderen Programmbausteinen zu kombinieren:

```
public class SwingUhr extends JPanel implements Steuerung {

  /** Unsere DigitalAnzeige */
  private Anzeige digital;

  /** unser einfaches Systemzeit-Uhrwerk */
  private Uhrwerk systemZeit;

  /** Eine Referenz auf das gerade verwendete Anzeigen-Objekt */
  private JComponent aktuell;
```

Wie Sie sehen, definieren wir drei Instanzvariablen:

- Eine Variable `digital` verwaltet die `DigitalAnzeige`-Instanz, die wir intern für die Darstellung verwenden.
- Unter `systemZeit` speichern wir das `UhrWerk`, von dem wir unsere Zeitangaben erhalten.
- Im Laufe der Zeit werden wir zwischen verschiedenen `Anzeige`-Implementierungen (analog und digital) hin- und herschalten wollen. Unter `aktuell` halten wir deshalb immer eine Referenz auf jene Anzeige, die gerade aktuell verwendet wird. Wir gehen hierbei davon aus, dass sich jede von uns verwendete Darstellung von der Klasse `JComponent` ableitet.

Da die Verwendung unserer digitalen Darstellung zu einem gewissen Zeitpunkt von außen an- und abschaltbar sein soll, extrahieren wir die für das Layout notwendigen Anweisungen in eine eigene Methode namens `setDigital`:

```
  public void setDigital()
  {
    // Fall: Wir sind ohnehin schon auf der digitalen Anzeige
    if (aktuell != null && aktuell.equals(digital))
      return;
    // Andernfalls loeschen wir die aktuelle Komponente und ersetzen
```

```
    // sie durch die Digitalanzeige
    if (aktuell != null)
      remove(aktuell);
    aktuell = (JComponent) digital;
    add(aktuell);
  }
```

Diese Methode wird dann von unserem Konstruktor aufgerufen:

```
  public SwingUhr() {
    // Erzeuge die Instanzen fuer Uhrwerk und Anzeige
    digital = new DigitalAnzeige();
    systemZeit = new Systemzeit();
    // Gehe sicher, dass das Uhrwerk gesetzt ist
    digital.setUhrwerk(systemZeit);
    // Leite die DigitalAnzeige an die Oberflaeche weiter
    setDigital();
  }
```

Es verbleibt nun nur noch die Implementierung der restlichen `Steuerung`-Methoden. Momentan sind diese noch relativ frei von Code und Bedeutung. Dies wird sich aber im nächsten Teil dieses Kapitels wesentlich ändern.

```
  /** Diese Methode signalisiert der Steuerung, dass
    * sie mit ihrer Arbeit beginnen soll.
    **/
  public void aktivieren() {
    digital.zeigeZeit();
  }

  /** Diese Methode signalisiert der Steuerung, dass
    * sie ihre Arbeit jetzt beenden kann.
    **/
  public void beenden() {
    // Keine besonderen Aktionen notwendig
  }
```

6.2.3 Nicht aus dem Rahmen fallen!

Wir haben mit unserer `SwingUhr` eine grafische Komponente geschaffen, die sich in vielerlei Zusammenhängen einsetzen lässt. Wir wollen nun eine `JavaUhr` programmieren, die sich diesen Umstand zunutze macht.
Unsere Klasse wird sich von der Klasse `JFrame` ableiten. Entsprechend werden wir wie gewohnt eine `main`-Methode verfassen, die ein Fenster dieser Klasse instantiiert und auf dem Bildschirm darstellt:

```
  public static void main(String[] args) {
    // Erzeuge unseren JavaUhr-Frame
    JavaUhr uhr = new JavaUhr();
    uhr.setTitle("Java Uhr");
    uhr.pack();
    uhr.setVisible(true);
  }
```

Doch nun zu unserer eigentlichen Klasse. Wir verwenden eine private Instanz unserer `SwingUhr`, die wir `anzeige` nennen. Da es sich hierbei um eine aktive Komponente handelt,[2] wollen wir die Methode `aktivieren` bzw. `beenden` aufrufen, wenn unser Fenster geöffnet oder geschlossen wird. Wir automatisieren diesen Vorgang, indem wir die Methode `setVisible` überschreiben:

```
public void setVisible(boolean value) {
  if (value != isVisible()) {
    super.setVisible(value);
    if (value)
      anzeige.aktivieren();
    else
      anzeige.beenden();
  }
}
```

Das eigentliche Layout nehmen wir im Konstruktor der Klasse vor. Im ersten Schritt setzen wir den Inhalt unseres Fensters durch die Methode `setContentPane` auf eine `SwingUhr`-Instanz:[3]

```
anzeige = new SwingUhr();
setContentPane(anzeige);
```

Ferner wollen wir für unser Fenster auch ein Menü definieren. Für den Moment haben wir nur einen Menüpunkt: das Beenden des Programms. Dies wird sich aber in naher Zukunft ändern.

```
JMenuBar bar = new JMenuBar();
JMenu system = new JMenu("System");
JMenuItem beenden = new JMenuItem("Beenden");
bar.add(system);
system.add(beenden);
setJMenuBar(bar);
```

Wie soll sich aber unser Fenster verhalten, wenn man das Menü aufruft oder es zu schließen versucht? Wir definieren das Verhalten in einer Methode namens `beenden` und lassen diese in beiden Fällen aufrufen:

```
// Erzeuge einen Listener fuer das Beenden-Menue
beenden.addActionListener(new ActionListener() {
  public void actionPerformed(ActionEvent e) {
    beenden();
  }
});
// Verwende dieselbe Aktion auch, wenn wir das Fenster
// schliessen wollen
addWindowListener(new WindowAdapter() {
  public void windowClosing(WindowEvent e) {
    beenden();
```

[2] Sobald wir gelernt haben, *wie*, aktualisiert die Komponente ihre Anzeige automatisch im Sekundentakt

[3] Die Methode `setContentPane` ist das Gegenstück zu `getContentPane`, die wir bislang in den Beispielen verwendet haben. Während wir also bislang einen Container vom Fenster bezogen und hier das Layout eingefügt haben, übergeben wir in diesem Praxisbeispiel das Layout komplett.

```
    }
  });
```

Beachten Sie, dass wir für das Schließen des Fensters nicht die Methode `setDefaultCloseOperation` einsetzen. Wir definieren vielmehr einen eigenen `WindowListener`, indem wir aus dem `WindowAdapter` eine anonyme Klasse bilden.
Wie soll nun unsere `beenden`-Methode aussehen? Üblicherweise sollte das Programm seinen Benutzer vor dem Beenden fragen, ob er sich dessen wirklich sicher ist. Gerade dies scheint aber mit nicht geringem Aufwand verbunden. Wir müssen einen Dialog bauen, den Dialog mit `ActionListenern` versehen und das Ergebnis in die Methode `beenden` zurückfließen lassen. Geht das denn nicht einfacher?
Glücklicherweise lautet die Antwort hierauf „ja“. Das Paket `javax.swing` bietet uns eine Sammlung von Hilfsmethoden an, um häufig vorkommende Ja/Nein-Dialoge vom System bauen zu lassen. Diese statischen Methoden der Klasse `JoptionPane` können wir wie Aufrufe unserer guten alten `IOTools` behandeln: das Ergebnis wird von der Methode selbst zurückgeliefert. Der Benutzer hat das Beenden genau dann bestätigt, wenn das Ergebnis der Methode `showConfirmDialog` die Konstante `JOptionPane.YES_OPTION` zurückliefert:

```
private void beenden() {
  // Frage den Benutzer, ob er das Ernst meint!
  int bistDuSicher = JOptionPane.showConfirmDialog(this,
    "Programm wirklich beenden?");
  // Falls ja, beende das Programm
  if (bistDuSicher == JOptionPane.YES_OPTION) {
    setVisible(false);
    System.exit(0);
  }
}
```

Für die Freunde und Liebhaber des Entwurfsmuster-Konzeptes sei an dieser Stelle ein neuer Pattern-Name erwähnt. Eine Klasse, die wie `JOptionPane` eine Anzahl gleichartiger Produkte zur Verfügung stellt (in diesem Fall Ja/Nein-Dialoge), bezeichnet man als eine **Fabrik** (englisch: **factory**). Die Fabrik verbirgt vor dem Benutzer, wie das Produkt tatsächlich erzeugt wird, und gibt dem Programmierer lediglich das Endergebnis zur Weiterverarbeitung.

6.2.4 Zusammenfassung

Am Ende der zweiten Iteration haben wir erneut ein lauffähiges Programm: die Klasse `JavaUhr`. Unsere neue Klasse sieht aus wie eine Digitaluhr und hat bereits erste Ansätze von grafischer Darstellung und menüorientierter Steuerung. Zwar fehlt noch immer die Analoguhr, und unsere Uhr ist leider auch stehengeblieben,[4] doch haben wir ja auch noch einige Iterationen in diesem Buch vor uns.

[4] Die Uhrzeit wird noch nicht aktualisiert.

Abbildung 6.2: Einstellungs-Dialog für die Uhren-Anwendung

6.3 Iteration 3: die Qual der Wahl

In diesem Abschnitt werden wir unser Fenster um einen Einstellungs-Dialog erweitern (siehe Abbildung 6.2). Ziel der Iteration ist es, das Look and feel unserer Applikation durch das Menü wechseln zu können.

6.3.1 Design und Layout

Wir definieren eine Klasse namens `Einstellungen`, die sich von `JDialog` ableitet:

```
public class Einstellungen extends JDialog {
```

Natürlich soll unser Dialog im Laufe der Zeit mehr tun, als nur das Look and feel zu bestimmen. Wie in Abbildung 6.2 gezeigt, entwerfen wir unseren Dialog deshalb als eine Ansammlung von so genannten „Karteireitern“ bzw. „Tabs“. Jeder, der schon einmal einen Windows-Einstellungsdialog gesehen hat, ist mit dem Prinzip vertraut. Jeder Karteireiter ist fuer das Setzen gewisser Eigenschaften verantwortlich. Durch das Wechseln zwischen den verschiedenen Karteireitern kann man verschiedene Eigenschaften beeinflussen.
Swing unterstützt uns bei der Verwendung von Karteireitern durch die Klasse `JTabbedPane`, die wir als Hauptelement unseres Dialoges einsetzen. Im Süden unseres Designs (wir verwenden `BorderLayout`) fügen wir ein `Panel` mit drei `JButtons` hinzu. Diese Schalter stellen die Aktionen „OK“, „Abbrechen“ und „Anwenden“ dar:

```
/** Konstruktor.
  * @param frame das Fenster, zu dem der Dialog gehoert
  *  (Wert kann auch null sein)
  * @param modal besagt, ob der Dialog modal ist
  * @param bestandteile jene Einstellungen, die in diesem
```

```
    *  Dialog gesetzt werden koennen.
    **/
  public Einstellungen(Frame frame,boolean modal,
    Einstellung[] bestandteile) {

    // SCHRITT 1: GRUNDLEGENDES LAYOUT
    // ===============================
    super(frame,modal);
    setTitle("Einstellungen");
    Container content = getContentPane();
    content.setLayout(new BorderLayout());
    // Unser Dialog verwendet ein JTabbedPane fuer
    // die verschiedenen Einstellungsarten
    JTabbedPane optionen = new JTabbedPane();
    content.add(optionen,BorderLayout.CENTER);
    // Fuer die OK/ANWENDEN/ABBRECHEN - Buttons
    // setzen wir ein weiteres Panel
    JPanel buttons = new JPanel();
    content.add(buttons,BorderLayout.SOUTH);

    // SCHRITT 2: SETZE DIE BUTTON-LEISTE,
    // INKLUSIVE DER ACTION-LISTENER
    // ================================
    buttons.setLayout(new FlowLayout(FlowLayout.RIGHT));
    JButton ok = new JButton("Ok");
    JButton anwenden = new JButton("Anwenden");
    JButton abbrechen = new JButton("Abbrechen");
    buttons.add(ok);
    buttons.add(anwenden);
    buttons.add(abbrechen);
```

Einzelne Karteireiter werden als Panel definiert und mit der Methode `addTab` in unser `JTabbedPane`-Objekt eingefügt. Abhängig davon, welcher Schalter betätigt wird, werden die momentan definierten Einstellungen entweder erzeugt oder verworfen. Es stellt sich hierbei jedoch die Frage, wie wir diese Dinge allgemeingültig realisieren, ohne bereits etwas über die Definition der verschiedenen Karteireiter zu wissen.
Wie Sie vielleicht schon der Signatur des Konstruktors entnommen haben, lösen wir das Problem durch ein neues Interface:

```
public static interface Einstellung {

  /** Gibt die grafische Komponente zurueck, die im
    * Dialog dargestellt werden soll.
    **/
  public Component getComponent();

  /** Gibt den Titel zurueck, unter dem die Komponente
    * dargestellt werden soll.
    **/
  public String getLabel();

  /** Diese Methode wird aufgerufen, wenn die Einstellungen
    * aus dem Dialog uebernommen werden sollen.
    **/
```

```
    public void anwenden();
}
```

Dieses Interface repräsentiert eine einzelne Einstellung, die mit Hilfe des Dialoges gesetzt werden kann. Konkrete Realisierungen müssen die folgenden Informationen zurückliefern:

- Eine grafische Komponente, die im `JTabbedPane` dargestellt werden kann.
- Eine textuelle Kurzbeschreibung für den Namen des dargestellten Karteireiters.
- Die Ausführungslogik, wenn die gesetzten Einstellungen übernommen werden sollen.

Mit dieser Definition haben wir alle Informationen, um beliebige Einstellungen über den Dialog vornehmen zu können. Wir speichern das im Konstruktor übergebene Feld in einer Instanzvariablen und fügen die Komponenten mittels einer Schleife in das `JTabbedPane` ein:

```
this.bestandteile = (Einstellung[]) bestandteile.clone();
for (int i = 0; i < bestandteile.length; i++)
  optionen.addTab(bestandteile[i].getLabel(),
    bestandteile[i].getComponent());
```

Für das Setzen der verschiedenen Einstellungen definieren wir eine simple Methode, die die `anwenden`-Methode der `Einstellung`-Objekte nacheinander aufruft:

```
/** Diese Methode wird aufgerufen, wenn der Ok- oder
  * Anwende-Button gedrueckt wird.
  **/
private void setzeEinstellungen() {
  for (int i = 0; i < bestandteile.length; i++)
    bestandteile[i].anwenden();
}
```

Es obliegt nun den `ActionListener`-Implementierungen der verschiedenen Schalter, ob die Methode aufgerufen wird:

- „OK“ führt die Einstellungen aus und schließt den Dialog.
- „Anwenden“ führt die Einstellungen ebenfalls aus, lässt den Dialog aber offen.
- „Abbrechen“ verwirft die Einstellungen und schließt den Dialog.

Die Listener sind wie folgt durch anonyme Klassen realisiert:

```
ok.addActionListener(new ActionListener() {
  public void actionPerformed(ActionEvent e) {
    setzeEinstellungen();
    setVisible(false);
  }
});
anwenden.addActionListener(new ActionListener() {
  public void actionPerformed(ActionEvent e) {
```

```
        setzeEinstellungen();
      }
    });
    abbrechen.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent e) {
        setVisible(false);
      }
    });
```

6.3.2 Wechsel des Look and feel

Kommen wir nun zu jener Klasse, die für das Setzen des Look and feel zuständig ist. Die Implementierung ist recht geradlinig und die Kommentare in dem Listing sollten für sich selbst sprechen. Achten Sie jedoch auf folgende Besonderheiten:

- Zeile 8 und 9: Unsere Klasse implementiert nicht nur das Interface `Einstellung`, sie ist zugleich die darzustellende Komponente. Aus diesem Grund wird in Zeile 69 `this` zurückgegeben.
- Zeile 17 und 25: Das GTK-Layout existiert erst ab JDK 1.4.2. Bei älteren Java-Versionen wird eine `ClassNotFoundException` geworfen, die wir in Zeile 89 auffangen.
- Zeile 93 bis 96: Bereits dargestellte Fenster müssen mittels der Methode `updateComponentTreeUI` aktualisiert werden, damit das neue Look and feel übernommen wird. Die aufzufrischenden Fenster werden in der Methode `setZuAktualisieren` von außen übergeben.

```
1  import java.awt.*;
2  import javax.swing.*;
3
4  /** Diese Klasse ermoeglicht dem Benutzer (in Verbindung
5    * mit dem Einstellungs-Dialog) das Setzen des
6    * Look and feel.
7    **/
8  public class SetzeLookAndFeel extends JPanel
9    implements Einstellungen.Einstellung
10 {
11   /** Die verschiedenen Namen der Look-and-feels */
12   private final static String[] NAMES = {
13     "Metal (Standard)",
14     "Windows",
15     "Motif",
16     "GTK (nur Java 1.4.2 oder hoeher)"
17   };
18
19   /** Instanzen der verschiedenen Look-and-feels */
20   private final static String[] CLASSES = {
21     "javax.swing.plaf.metal.MetalLookAndFeel",
22     "com.sun.java.swing.plaf.windows.WindowsLookAndFeel",
23     "com.sun.java.swing.plaf.motif.MotifLookAndFeel",
24     "com.sun.java.swing.plaf.gtk.GTKLookAndFeel"
```

```
25    };
26
27    /** Diese Fenster muessen beim Wechsel des Look and feel
28      * aktualisiert werden.
29      **/
30    private Window[] zuAktualisieren = new Window[0];
31
32    /** Jeder RadioButton steht fuer ein Look and feel */
33    private JRadioButton[] buttons;
34
35    /** Konstruktor - hier wird das Layout gesetzt */
36    public SetzeLookAndFeel() {
37      // Schachtele mit einem inneren Panel
38      setLayout(new FlowLayout(FlowLayout.LEFT));
39      JPanel innerPanel = new JPanel();
40      add(innerPanel);
41      innerPanel.setLayout(new GridLayout(NAMES.length,1));
42      // Innerhalb des Panels setze nun
43      // die verschiedenen RadioButtons
44      buttons = new JRadioButton[NAMES.length];
45      ButtonGroup group = new ButtonGroup();
46      for (int i = 0; i < buttons.length; i++) {
47        buttons[i] = new JRadioButton(NAMES[i]);
48        group.add(buttons[i]);
49        innerPanel.add(buttons[i]);
50      }
51      // Zu Anfang ist der erste Button selektiert
52      buttons[0].setSelected(true);
53    }
54
55    /** Setze die zu aktualisierenden Fenster */
56    public void setZuAktualisieren(Window[] zuAktualisieren) {
57      if (zuAktualisieren != null)
58        this.zuAktualisieren = (Window[])
59          zuAktualisieren.clone();
60      else
61        this.zuAktualisieren = new Window[0];
62    }
63
64    /** Gibt die grafische Komponente zurueck, die im
65      * Dialog dargestellt werden soll.
66      **/
67    public Component getComponent() {
68      return this;
69    }
70
71    /** Gibt den Titel zurueck, unter dem die Komponente
72      * dargestellt werden soll.
73      **/
74    public String getLabel() {
75      return "Look&Feel";
76    }
77
78    /** Diese Methode wird aufgerufen, wenn die Einstellungen
79      * aus dem Dialog uebernommen werden sollen.
```

```
80      **/
81    public void anwenden() {
82      for (int i = 0; i < buttons.length; i++)
83        if (buttons[i].isSelected()) {
84          // Setze das Look and feel
85          try {
86            UIManager.setLookAndFeel(CLASSES[i]);
87          }
88          catch(Exception e) {
89            e.printStackTrace();
90          }
91          // Aktualisiere gezeichnete Komponenten
92          for (int j = 0; j < zuAktualisieren.length; j++) {
93            SwingUtilities.updateComponentTreeUI(zuAktualisieren[j]);
94            zuAktualisieren[j].pack();
95          }
96          // Somit sind wir fertig
97          return;
98        }
99    }
100
101 }
```

Mit dieser Definition ist unser Dialog fertig – wir müssen ihn also lediglich noch in unserem Hauptfenster erzeugen können. Zu diesem Zweck speichern wir eine Dialoginstanz in einer privaten Instanzvariablen der Klasse `JavaUhr` und definieren folgende Methode:

```
/** Diese Methode wird aufgerufen,
  * um den Einstellungs-Dialog anzuzeigen.
  **/
private void zeigeEinstellungen() {
  // Erzeuge nur neue Objekte, wenn der Dialog noch nicht existiert
  if (einstellungen == null) {
    // Initialisiere die Look-and-feel-Einstellung
    SetzeLookAndFeel lookAndFeel = new SetzeLookAndFeel();
    // Initialisiere den Dialog
    einstellungen =
    new Einstellungen(this,true,new Einstellungen.Einstellung[]{
      lookAndFeel
    });
    // Gegen Ende noch einige letzte Einstellungen
    lookAndFeel.setZuAktualisieren(new Window[] {
      this,einstellungen
    });
  }
  // Mache den Dialog sichtbar
  einstellungen.pack();
  einstellungen.setVisible(true);
}
```

Im Konstruktor unserer Klasse fügen wir nun lediglich ein weiteres `MenuItem` hinzu und sorgen mit einem entsprechenden Listener dafür, dass die Methode aufgerufen wird:

```
// Erzeuge ein Menue mit den verschiedenen Optionen
JMenuBar bar = new JMenuBar();
JMenu system = new JMenu("System");
JMenuItem einstellungen = new JMenuItem("Einstellungen");
JMenuItem beenden = new JMenuItem("Beenden");
bar.add(system);
system.add(einstellungen);
system.add(beenden);
setJMenuBar(bar);
// Erzeuge einen Listener fuer das Einstellungs--Menue
einstellungen.addActionListener(new ActionListener() {
  public void actionPerformed(ActionEvent e) {
    zeigeEinstellungen();
  }
});
```

6.4 Iteration 4: Zeiger und Zifferblatt

6.4.1 Erste Schritte

In diesem Abschnitt schlägt erneut eine große Stunde für unser Uhrenprojekt: wir versehen unsere Uhr mit einem analogen Zifferblatt.
Die Programmierung des gesamten Zifferblattes wird relativ komplex, wir fangen daher klein an. Unsere erste Version soll

- eine Teilimplementierung des Zifferblattes vornehmen und
- dieses anstelle der Digitalanzeige auf dem Bildschirm darstellen.

Gemäß unserem generellen Entwurf ist es klar, dass unser Zifferblatt das Interface `Anzeige` implementiert. Wir leiten unsere Klasse `AnalogAnzeige` ferner vom `JPanel` ab, das wir als digitale Leinwand verwenden:

```
public class AnalogAnzeige extends JPanel implements Anzeige {
```

Wir modifizieren unsere `SwingUhr`-Klasse nun so, dass sie die neue Klasse verwendet. Zuerst definieren wir eine Instanzvariable, die wir zur Speicherung des Anzeigenobjektes verwenden:

```
private Anzeige analog;
```

Im Konstruktor instantiieren wir das Objekt

```
analog = new AnalogAnzeige();
```

und weisen ihm das korrekte Uhrwerk zu:

```
analog.setUhrwerk(systemZeit);
```

Auch müssen wir in der Methode `aktivieren` nun bedenken, dass wir es mit unterschiedlichen Anzeigen zu tun haben können:

```
public void aktivieren() {
  ((Anzeige)aktuell).zeigeZeit(); //NEU
}
```

Ferner rufen wir eine Methode namens `setAnalog` auf, die wir wie folgt definieren:

```
public void setAnalog()
{
  // Fall: Wir sind sowieso schon auf der digitalen Anzeige
  if (aktuell != null && aktuell.equals(analog))
    return;
  // Andernfalls loeschen wir die aktuelle Komponente und ersetzen
  // sie durch die Digitalanzeige
  if (aktuell != null)
    remove(aktuell);
  aktuell = (JComponent) analog;
  add(aktuell);
}
```

Zurück zu unserer `AnalogAnzeige`. Wie auch in ihrem Vorgänger speichern wir das `Uhrwerk` in einer Instanzvariable. Wir definieren jedoch ferner noch weitere Instanzvariablen, in denen wir Stunden, Minuten und Sekunden der darzustellenden Uhrzeit hinterlegen:

```
/** Dieses Uhrwerk wird fuer die Zeitdarstellung verwendet. */
private Uhrwerk uhrwerk;

/** Die darzustellenden Stunden (0-11) */
private int stunden;

/** Die darzustellenden Minuten (0-59) */
private int minuten;

/** Die darzustellenden Sekunden (0-59) */
private int sekunden;

/** Ist es nachmittags (PM) ? */
private boolean zeigePM;
```

Die Variable `zeigePM` speichert, ob es sich um den Zeitbereich von 0 bis 12 oder den Zeitbereich von 12 bis 24 Uhr handelt. Im letztgenannten Fall werden wir einen kleinen Punkt auf der Analoguhr anzeigen.
Werfen wir nun einen Blick auf den Konstruktor unserer Klasse:

```
/** Konstruktor */
public AnalogAnzeige() {
  // Veraendere das Aussehen
  setBackground(Color.BLACK);
  setForeground(Color.GREEN);
  // Sorge dafuer, dass die Komponente
  // eine gewisse bevorzugte Groesse hat
  setPreferredSize(new Dimension(200,200));
}
```

Die meisten Anweisungen sind aus früheren Praxisbeispielen bereits bekannt. Die Methode `setPreferredSize` teilt der Komponente mit, wie groß sie bevorzugt dargestellt werden soll. Dieser Wunsch wird dann im Layout von der Methode `pack()` berücksichtigt.

Kommen wir nun zur Methode `zeigeZeit`, die die Darstellung unserer Uhrzeit aktualisiert. Mit Hilfe eines `GregorianCalendar`-Objektes extrahieren wir die notwendigen Zeitinformationen und aktualisieren unsere Instanzvariablen. Anschließend rufen wir `repaint()` auf und erneuern die Darstellung:

```
public void zeigeZeit() {
  // Extrahiere Stunden, Minuten und Sekunden
  Date date = uhrwerk.getZeit();
  Calendar calendar = new GregorianCalendar();
  calendar.setTime(date);
  stunden = calendar.get(Calendar.HOUR);
  minuten = calendar.get(Calendar.MINUTE);
  sekunden = calendar.get(Calendar.SECOND);
  zeigePM = calendar.get(Calendar.AM_PM) == Calendar.PM;
  // Dann rufe die repaint-Methode auf
  repaint();
}
```

Die eigentliche Darstellung wird also von der Methode `paint()` erledigt. Wir gehen in folgenden Schritten vor:

- Kompliziertere Grafiken werden normalerweise nicht auf dem Bildschirm selbst gezeichnet. Je nach Prozessorgeschwindigkeit kann der Benutzer den Zeichenvorgang sonst „mitverfolgen“: das Bild flackert. Man verwendet deshalb ein so genanntes **Offscreen Image**, das heißt ein Bild, das nicht auf dem Bildschirm zu sehen ist:

  ```
  public void paint(Graphics graphics) {
    // Zeichne nicht direkt auf dem Bildschirm, sondern
    // verwende ein so genanntes Offscreen-Image
    int width = getWidth();
    int height = getHeight();
    Image bild = createImage(width,height);
    Graphics g = bild.getGraphics();
  ```

  Mit dem auf diese Weise gewonnenen `Graphics`-Objekt werden wir unsere Uhr zeichnen. Erst wenn wir damit fertig sind, bringen wir das komplette Bild in einem Schritt auf den Bildschirm:

  ```
    graphics.drawImage(bild,0,0,this);
  ```

- Bevor wir anfangen, unsere Uhr zu zeichnen, füllen wir das Bild mit unserer Hintergrundfarbe:

  ```
    g.setColor(getBackground());
    g.fillRect(0,0,width,height);
  ```

  Auf diese Weise wird alles übermalt, was sich eventuell noch auf dem Bildschirm befunden haben mag. Ferner berechnen wir gewisse Grunddaten unserer (kreisfoermigen) Uhr, indem wir den Mittelpunkt des Bildes errechnen und den größtmöglichen Kreisradius (abzüglich 10 Bildpunkte, damit wir nicht zu sehr an den Rand stoßen) bestimmen:

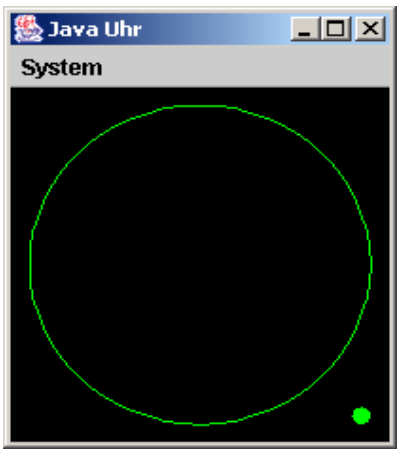


Abbildung 6.3: Die Analoguhr im ersten Schritt

```
int mitteX = width/2;
int mitteY = height/2;
int radius = Math.min(width,height) / 2 - 10;
```

- Nun können wir anfangen, unser Zifferblatt zu malen. Im ersten Schritt besteht unser Zifferblatt lediglich aus einem Kreis in der Vordergrundfarbe – wir fangen klein an, werden uns aber bald steigern:

```
g.setColor(getForeground());
g.drawOval(mitteX - radius, mitteY - radius,
```

- Zu guter Letzt wollen wir den Inhalt des Feldes `zeigePM` auswerten. Falls der Inhalt `true` ist, malen wir einen kleinen Punkt an die rechte untere Hälfte des Fensters:

```
if (zeigePM)
  g.fillOval(width - 20, height - 20 , 10 , 10);
```

Betrachten wir die komplette Methode noch einmal in ihrer Gesamtheit:

```
public void paint(Graphics graphics) {
  // Zeichne nicht direkt auf dem Bildschirm, sondern
  // verwende ein so genanntes Offscreen-Image
  int width = getWidth();
  int height = getHeight();
  Image bild = createImage(width,height);
  Graphics g = bild.getGraphics();
  // Berechne den Mittelpunkt unseres Uhrenkreises
  int mitteX = width/2;
  int mitteY = height/2;
  int radius = Math.min(width,height) / 2 - 10;
  // Zuerst einmal malen wir den Hintergrund
```

```
    g.setColor(getBackground());
    g.fillRect(0,0,width,height);
    // Nun zum Zifferblatt
    g.setColor(getForeground());
    g.drawOval(mitteX - radius, mitteY - radius,
      2 * radius, 2 * radius);
    // Im Falle von "PM", zeige einen kleinen Punkt an
    if (zeigePM)
      g.fillOval(width - 20, height - 20 , 10 , 10);
    // Zu guter Letzt zeichnen wir das Offscreen-Image
    // in der wirklichen Komponente
    graphics.drawImage(bild,0,0,this);
  }
```

Abbildung 6.3 stellt das Ergebnis dieser Methode dar, wenn wir unser Programm laufen lassen. Wir haben unsere ersten Schritte zu unserem Iterationsziel gemacht und dabei die prinzipielle Lauffähigkeit unserer Änderungen sichergestellt. Nun wollen wir uns daran machen, die grafische Darstellung weiter zu verfeinern.

6.4.2 Von Kreisen und Winkeln

Eine zentrale Rolle beim Zeichnen unserer Uhr stellt die Darstellung ihrer Zeiger dar. Stunden-, Minuten- und Sekundenzeiger. Für jeden dieser Zeiger gilt:

- Er ist in der Mitte des Kreises (unserer Uhr) befestigt und verläuft zum äußeren Rand.
- Abhängig von der Uhrzeit wechselt der Winkel (wir definieren 12 Uhr als 0 Grad).
- Keiner der Zeiger kann länger als das Zifferblatt selbst sein (also größer als der Radius).
- Die Enden der Zeiger sollten abgerundet sein, da dies besser aussieht.

Neben diesen Gemeinsamkeiten gibt es aber auch viele Unterschiede. So sind die verschiedenen Zeiger unterschiedlich lang und eventuell auch unterschiedlich breit. Auch drehen sich die Zeiger unterschiedlich schnell und durchlaufen eine volle Kreisdrehung in 60 Sekunden, 60 Minuten oder 12 Stunden. Sprich, abhängig vom Zeiger müssen wir eine andere Berechnungsformel zugrunde legen.
Bevor wir uns daran machen, das unterschiedliche Verhalten zu realisieren, kümmern wir uns um die Gemeinsamkeiten. Jeder Zeiger kann eine ihm zugeordnete Breite haben – wir brauchen also eine Methode, um die Pinseldicke ändern zu können:

```
private void setzeBreite(Graphics g,int breite,boolean rundeEnden) {
}
```

Neben dem gerade verwendeten `Graphics`-Objekt und der Pinseldicke (in Bildschirmpunkten) übergeben wir als dritten Parameter einen Booleschen Wert

`rundeEnden`. Ist dieser auf `true` gesetzt, sollen alle gezeichneten Striche am Ende abgerundet werden. Da wir momentan noch nicht genau wissen, wie dies zu bewerkstelligen ist, lassen wir die Methode allerdings zunächst leer.
Kommen wir zur zweiten Gemeinsamkeit. Abhängig von ihrem Winkel sollen alle drei Zeiger vom Mittelpunkt des Kreises nach außen gezeichnet werden (wenn auch mit unterschiedlicher Länge). Wir definieren eine Methode `zeichneLinie`, der wir folgende Parameter übergeben:

- das zu verwendende `Graphics`-Objekt,
- die x- und y-Koordinate des Kreismittelpunktes (`mitteX` und `mitteY`, bereits in der `paint()`-Methode berechnet),
- den Winkel, gemessen im Uhrzeigersinn von „zwölf Uhr", in dem sich der Zeiger befindet,
- den Radius unseres Kreises (`radius`, bereits in der `paint()`-Methode berechnet),
- bei wie viel Prozent der Strecke Mittelpunkt-Kreisbogen von der Mitte aus gesehen der Strich beginnen soll (`von`) und
- bei wie viel Prozent der Strecke Mittelpunkt-Kreisbogen von der Mitte aus gesehen der Strich enden soll (`bis`).

Da wir momentan allerdings noch nicht genau wissen, *wie* wir diese Aufgabe bewerkstelligen wollen, verschieben wir auch hier die eigentliche Implementierung auf später:

```
private void zeichneLinie(Graphics g,int mitteX,
  int mitteY,double winkel,int radius,double von,double bis) {
}
```

Der Hauptunterschied zwischen den verschiedenen Zeigern ist die Berechnung des eingenommenen Winkels: Wenn der Zeiger beim Winkel 0 beginnt und eine volle Umdrehung des Kreises einen Winkel von 2π bedeutet, dann ist die Berechnung wie folgt:

- Für den Sekundenzeiger unterteilen wir die Skala in 60 Bestandteile:

```
private double sekundenWinkel(int sekunde) {
  return 2 * Math.PI * sekunde / 60;
}
```

- Für den Minutenzeiger unterteilen wir die Skala in 3600 Bestandteile (60 Minuten mal 60 Sekunden):

```
private double minutenWinkel(int minute,int sekunde) {
  return 2 * Math.PI * (sekunde + minute * 60) /3600;
}
```

- Für den Stundenzeiger verwenden wir einen vollen 12-Stunden-Takt (in Sekunden gemessen):

```
private double stundenWinkel(int stunde,int minute,int sekunde) {
  return 2 * Math.PI * (sekunde + minute * 60 + stunde * 3600)
    / (3600 * 12);
}
```

Nun haben wir alle Hilfsmethoden definiert, um unsere `paint`-Methode zu vervollständigen. Zuerst polieren wir unser Zifferblatt ein wenig auf. Im ersten Schritt machen wir den äußeren Kreis ein wenig dicker:

```
setzeBreite(g,4,false);
g.setColor(getForeground());
g.drawOval(mitteX - radius, mitteY - radius,
  2 * radius, 2 * radius);
```

Als nächstes fügen wir noch eine kleine Skala für die vollen Stunden hinzu. Denn was ist ein solche Strich anderes als ein Zeiger für volle Stunden, der ziemlich weit außerhalb beginnt (also einen hohen Wert für `von` hat)?

```
for (int i = 0; i < 12; i++)
  zeichneLinie(g,mitteX,mitteY,stundenWinkel(i,0,0),
    radius,0.75,1);
```

Kommen wir nun zu unseren drei Zeigern. Wir verwenden die Methoden `setzeBreite` und `zeichneLinie`, um die drei Zeiger mit unterschiedlichem Aussehen zu gestalten. Die dazugehörigen Winkel werden von unseren Hilfsmethoden berechnet:

```
// Zeichne den Stundenzeiger
setzeBreite(g,9,true);
zeichneLinie(g,mitteX,mitteY,
  stundenWinkel(stunden,minuten,sekunden),radius,0,0.5);
// Zeichne den Minutenzeiger
zeichneLinie(g,mitteX,mitteY,
  minutenWinkel(minuten,sekunden),radius,0,0.9);
// Zeichne den Sekundenzeiger
setzeBreite(g,2,true);
zeichneLinie(g,mitteX,mitteY,
  sekundenWinkel(sekunden),radius,0,0.9);
```

Auch wenn sich unser Bildschirm nach dem Programmstart um keinen Deut unterscheidet, ist die Methode `paint` somit komplett. Wir werden die Früchte unserer Arbeit sehen, sobald wir die beiden fehlenden Methoden ausprogrammiert haben.

6.4.3 Die Methode `setzeBreite`

Finden Sie nicht auch, dass die Möglichkeiten der Klasse `Graphics` ziemlich eingeschränkt sind? Falls wir Ihnen bislang nichts Wesentliches verschwiegen haben, beschränkt sich die Funktionalität der Klasse auf das simple Malen einiger Linien, Kreise und vordefinierter Bilder. Ist das wirklich alles?
In den Anfangszeiten von Java hätten wir die Frage mit ja beantworten müssen. In den ersten beiden Versionen der Sprache waren die Grafikmöglichkeiten stark

beschränkt – sofern man nicht aufwendig selbst entsprechende Routinen definierte. Mit der Version 1.2 von Java war dies aber vorbei. Sun unterzog die Klasse einer kompletten Rundumerneuerung – Ergebnis war eine neue, verbesserte Variante: `Graphics2D`. Diese neue Klasse ist Einstiegspunkt in ein komplexes und sehr fortgeschrittenes 2D-Rendering-System, das dem versierten Grafikprogrammierer kaum Wünsche offenlässt.
Wie kommen wir in unserer Klasse aber an ein `Graphics2D`-Objekt heran? Die Antwort lautet: wir *haben* es bereits. Jedes `Graphics`-Objekt der neueren Java-Versionen lässt sich auf dessen Unterklasse `Graphics2D` casten. Wir können also im vollen Vertrauen darauf mit der erweiterten Funktionalität arbeiten.
Eine der Neuerungen der neuen Klasse ist die Methode `setStroke()`. Der `Stroke`, grob übersetzt Pinselstrich, gibt Auskunft über verschiedene Eigenschaften einer zu zeichnenden Linie. In der konkreten Ausprägung `BasicStroke` lassen sich unter anderem die Liniendicke und die Form der Linienenden (quadratisch oder abgerundet) setzen. Wir verwenden dieses Wissen, um unsere Methode `setzeBreite` zu formulieren:

```
private void setzeBreite(Graphics g,int breite,boolean rundeEnden) {
  Graphics2D g2d = (Graphics2D) g;
  g2d.setStroke(new BasicStroke(breite,
    rundeEnden ? BasicStroke.CAP_ROUND : BasicStroke.CAP_SQUARE,
    BasicStroke.JOIN_MITER));
}
```

Der dritte Parameter im Konstruktor, `JOIN_MITER` besagt, nach welchem Prinzip ineinanderlaufende Linien vereint werden sollen. Wir setzen hier den standardgemäß verwendeten Wert.
Für nähere Informationen über `Graphics2D` und `Stroke` sei auf die API-Beschreibung und weiterführende Literatur wie das Java Tutorial verwiesen.

6.4.4 Die Methode `zeichneLinie`

Kommen wir nun zu unserer letzten Methode: wir wollen eine Linie vom Kreismittelpunkt (`mitteX`/`mitteY`) im Winkel `winkel` zur Ausgangsstellung zeichnen. Die Linie soll `radius` Bildschirmpunkte lang sein, aber nur der Prozentbereich von `von` bis `bis` soll dargestellt werden.
Würde uns der Winkel nicht in die Quere kommen, wäre das Ganze kein Problem. Angenommen, der Winkel wäre 0, also „zwölf Uhr“. Dann wären die y-Koordinaten von Start- und Endpunkt nach folgendem Schema einfach berechenbar:

```
double startY = mitteY - radius * von;
double endY = mitteY - radius * bis;
```

Die x-Koordinate wäre genau der Mittelpunkt. Wir müssten den Strich also nur noch zeichnen:

```
g.drawLine(mitteX,(int)startY,mitteX,(int)endY);
```

Wie aber bringen wir den Winkel ins Spiel? Es ist der Klasse `Graphics2D` zu verdanken, dass dieser Abschnitt des Buchs in keine Geometrievorlesung ausartet. Anstatt die Koordinaten anhand des Winkels berechnen zu müssen, können wir uns nämlich einfach unser Zeichenblatt „zurechtdrehen".
Java unterstützt so genannte **affine Transformationen**. Ohne in die lineare Algebra abrutschen zu wollen, stellen diese Funktionen vereinfacht gesagt eine Möglichkeit dar, das zum Zeichnen verwendete Koordinatensystem zu verrücken. Die Klasse `java.awt.geom.AffineTransform` repräsentiert diese Transformationen und stellt auch einige statische Hilfsmethoden zur Verfügung. Die Methode `getRotateInstance(winkel,mitteX,mitteY)` erzeugt eine Transformation, die unsere Koordinatenachse um den Mittelpunkt unseres Kreises gegen den Uhrzeigersinn dreht. Mit anderen Worten: schleusen wir dieses Objekt mit der Methode `transform` in unser `Graphics2D`-Objekt ein, dann befindet sich die von uns zu zeichnende Line gerade auf „zwölf Uhr".
Betrachten wir die Methode in ihrer Gesamtheit

```
private void zeichneLinie(Graphics g,int mitteX,
  int mitteY,double winkel,int radius,double von,double bis) {
   // Drehe die Bildschirmflaeche zurecht
   Graphics2D g2d = (Graphics2D) g;
   g2d.transform(AffineTransform.getRotateInstance
     (winkel,mitteX,mitteY));
   // Nach dieser Drehung muessen wir den Strich nur noch
   // nach oben malen :-)
   double startY = mitteY - radius * von;
   double endY = mitteY - radius * bis;
   g.drawLine(mitteX,(int)startY,mitteX,(int)endY);
   // Zu guter Letzt drehen wir das Bild wieder zurueck
   g2d.transform(AffineTransform.getRotateInstance
     (-winkel,mitteX,mitteY));
}
```

Wir drehen unser Bild zuerst in eine für uns angenehme Position und zeichnen dann die Linie. Zu guter Letzt machen wir die Drehung wieder rückgängig, indem wir in die entgegengesetzte Richtung drehen.

6.4.5 Zusammenfassung

Wie Abbildung 6.4 zeigt, haben wir es geschafft: unsere Analoguhr befindet sich fix und fertig auf dem Bildschirm.
Nach dieser kolossalen Verbesserung wollen wir einen Moment innehalten und sehen, welche Ziele wir für die Anwendung eigentlich noch verwirklichen wollen:

- Bislang haben wir es noch nicht geschafft, unsere Uhr in Bewegung zu setzen. Die Anzeige ist auf der zuerst dargestellten Uhrzeit eingefroren.
- Wir verfügen nun über eine Analog- und eine Digitaluhr, haben aber keine Möglichkeit, zwischen diesen beiden Darstellungen hin- und herzuschalten.
- Unsere Uhr ist eine Java-Anwendung und als solche noch nicht für den Web-Browser geeignet. Ein Applet muss her!

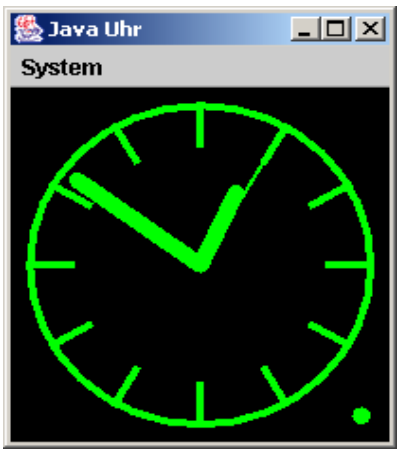


Abbildung 6.4: Die fertige Analoguhr

- Für das Einstellen der Uhrzeit wollten wir unsere Uhr mit einem „Zeitserver" synchronisieren.

Wir werden die nächsten Praxisbeispiele darauf verwenden, diese offenen Punkte zu klären. Nichtsdestotrotz sollten wir aber auch nicht unterschlagen, welche kolossalen Fortschritte unser Programm von seinen ersten Schritten aus gemacht hat. Die grafischen Teile sind geschafft, vier von insgesamt acht Iterationsschritten erledigt. Herzlichen Glückwunsch – Halbzeit!

6.5 Iteration 5: Mehr Einstellungen

Ziel dieser Iteration ist es,

- zwischen analoger und digitaler Zeitdarstellung per Einstellungsmenü wechseln zu können, und
- dem Benutzer die Wahl zu lassen, in welchen Farben die Uhr dargestellt werden soll.

Hierzu werden wir unseren Einstellungsdialog aus Abschnitt 6.3 so erweitern, dass diese Funktionalität zur Verfügung steht.

6.5.1 Vorbereitungen

Werfen wir einen kurzen Blick auf unsere `SwingUhr`. Mit den Methoden `setAnalog` und `setDigital` haben wir bereits einen Schalter zur Verfügung gestellt, mit dem zwischen den Darstellungen hin- und hergeschaltet werden kann.

So weit, so gut. Wir fügen ferner noch eine Methode `isAnalog` hinzu, mit der sich der Status der aktuell verwendeten Darstellung abfragen lässt:

```
public boolean isAnalog() {
  return aktuell.equals(analog);
}
```

Neben diesen Methoden benötigen wir aber auch noch eine Möglichkeit, die Farben unserer Darstellung zu beeinflussen. Die aktuell definierten Methoden `setForeground` und `setBackground` reichen die Einstellungen nicht an unsere inneren Darstellungen weiter. Wir überschreiben diese Methoden deshalb entsprechend:

```
/** Setze die Farbe fuer den Vordergrund der Uhr */
public void setForeground(Color c) {
  if (digital != null)
    ((JComponent)digital).setForeground(c);
  if (analog != null)
    ((JComponent)analog).setForeground(c);
  super.setForeground(c);
}

/** Setze die Farbe fuer den Hintergrund der Uhr */
public void setBackground(Color c) {
  if (digital != null)
    ((JComponent)digital).setBackground(c);
  if (analog != null)
    ((JComponent)analog).setBackground(c);
  super.setBackground(c);
}
```

Konsequent überschreiben wir auch die get-Methoden:

```
/** Frage die aktuelle Vordergrundfarbe ab */
public Color getForeground() {
  if (aktuell != null)
    return aktuell.getForeground();
  return super.getForeground();
}

/** Frage die aktuelle Hintergrundfarbe ab */
public Color getBackground() {
  if (aktuell != null)
    return aktuell.getBackground();
  return super.getForeground();
}
```

Nach diesen geringfügigen Anpassungen haben wir alle notwendigen „Schalter“ definiert, die wir in unserem Einstellungs-Dialog nur noch nach außen reichen müssen.

6.5.2 Layout in der Klasse `SetzeDarstellung`

Erinnern wir uns: In Abschnitt 6.3 haben wir einen generellen Einstellungs-Dialog definiert, der eine vorgegebene Sammlung von `Einstellung`-Objekten grafisch

darstellt und das Setzen der Einstellungen kontrolliert. Um diesen Dialog erweitern zu können, benötigen wir also nur eine neue Klasse für unsere zusätzlichen Optionen:

```
public class SetzeDarstellung extends JPanel
  implements Einstellungen.Einstellung
```

Innerhalb dieser Klasse werden wir folgende Komponenten verwenden:

- Für den Wechsel zwischen Analog- und Digitalanzeige definieren wir zwei `JRadioButton`-Objekte:

```
/** RadioButton: analoge Anzeige */
private JRadioButton analog;

/** RadioButton: digitale Anzeige */
private JRadioButton digital;
```

- Für das Setzen von Vorder- und Hintergrundfarbe verwenden wir so genannte `JColorChooser`-Objekte, die speziell für die Auswahl von Farben durch den Benutzer entwickelt wurden:

```
/** ColorChooser fuer den Vordergrund */
private JColorChooser foreground;

/** ColorChooser fuer den Hintergrund */
private JColorChooser background;
```

- Ferner definieren wir noch zwei weitere Instanzvariablen, in denen wir die `SwingUhr` und das Fenster hinterlegen, in dem die Darstellung vollzogen wird:

```
/** Diese SwingUhr wird mit den Einstellungen beeinflusst */
private SwingUhr uhr;

/** Dieses Fenster muss beim Wechsel des Look and feel
  * aktualisiert werden.
  **/
private Window uhrenFenster;
```

  Diese Objekte müssen beim Setzen der Einstellungen von uns beeinflusst werden.

Im Konstruktor unserer Klasse werden wir diese Objekte nun so kombinieren, dass ein anspruchsvoller Einstellungs-Dialog entsteht. Zuerst initialisieren wir die Instanzvariablen:

```
public SetzeDarstellung(SwingUhr uhr,Window uhrenFenster) {
  // Initialisiere die Instanzvariablen.
  this.uhr = uhr;
  this.uhrenFenster = uhrenFenster;
  analog = new JRadioButton("analog");
  digital = new JRadioButton("digital");
  foreground = new JColorChooser(uhr.getForeground());
  background = new JColorChooser(uhr.getBackground());
```

Im nächsten Schritt kümmern wir uns um die Wahl der Darstellungsform. Wir arrangieren beide Buttons nebeneinander und sorgen mit einer `ButtonGroup` dafür, dass die Auswahl der einen Darstellungsart die andere ausschließt:

```
JPanel buttons = new JPanel();
buttons.setLayout(new GridLayout(1,2));
buttons.add(analog);
buttons.add(digital);
ButtonGroup group = new ButtonGroup();
group.add(analog);
group.add(digital);
if (uhr.isAnalog())
  analog.setSelected(true);
else
  digital.setSelected(true);
```

Kommen wir nun zur Wahl der Farbe. Darstellungen der Form `JColorChooser` benötigen unglücklicherweise recht viel Platz. Wir verwenden deshalb ein `JTabbedPane`, um nicht beide gleichzeitig auf dem Bildschirm anzeigen zu müssen:

```
JTabbedPane colors = new JTabbedPane();
colors.addTab("Vordergrund",foreground);
colors.addTab("Hintergrund",background);
```

Nun müssen wir die `buttons` und `colors` nur noch auf dem Bildschirm darstellen. Achten Sie auf die Verwendung der Methode `setBorder`:

```
// Fuege Rahmen um die inneren Panels ein
buttons.setBorder(new TitledBorder("Darstellungsform"));
colors.setBorder(new TitledBorder("Farbgebung"));
// Fuege beide Panels in dieses Panel ein
setLayout(new BorderLayout());
add(buttons,BorderLayout.NORTH);
add(colors,BorderLayout.CENTER);
```

Das Interface `javax.swing.border.Border` und seine konkreten Ausprägungen sind für das Malen von Rahmen um eine Swing-Komponente zuständig. Unser `TitledBorder` malt einen einfachen Rahmen mit einer Beschriftung um unsere Komponenten. Sie können das fertige Ergebnis in Abbildung 6.5 betrachten.

6.5.3 Vom Layout zur Anwendungslogik

Nachdem wir nun das grafische Layout bewerkstelligt haben, müssen wir die vom Benutzer gesetzten Einstellungen noch an unseren Dialog weiterleiten. Hierzu prägen wir die Methode `anwenden` aus dem Einstellungs-Interface aus:

```
public void anwenden() {
  // Analog oder digital?
  if (analog.isSelected())
    uhr.setAnalog();
  else
    uhr.setDigital();
  // Farbgebung
```

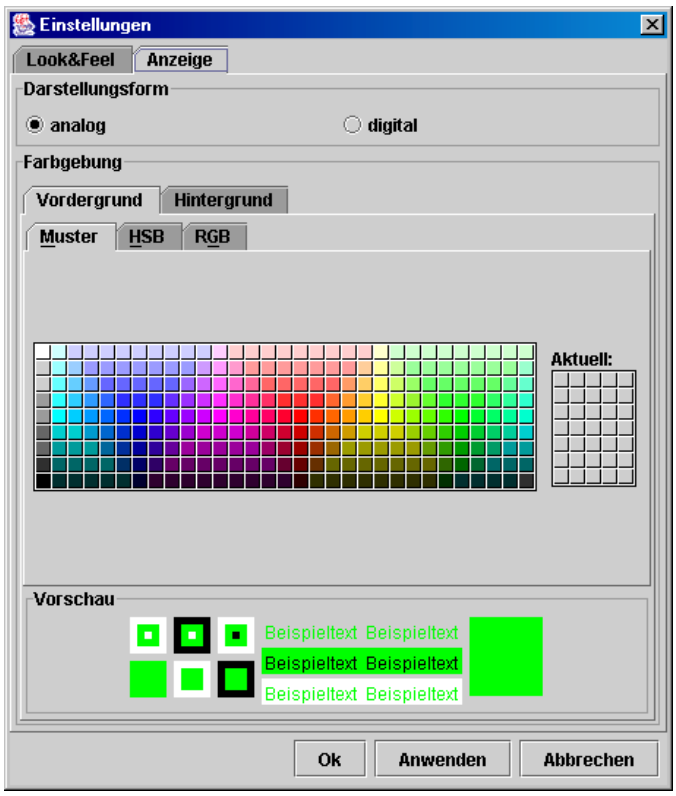


Abbildung 6.5: Einstellungsdialog: Darstellung und Farbgebung

```
    uhr.setForeground(foreground.getColor());
    uhr.setBackground(background.getColor());
    // Groessenanpassungen
    if (uhrenFenster != null)
      uhrenFenster.pack();
  }
```

Wie in früheren Beispielen lesen wir die gesetzten Werte aus unseren grafischen Komponenten aus und leiten sie in die entsprechenden Methoden unserer Steuerungslogik weiter. Aus unserem `JColorChooser` können wir Vorder- bzw. Hintergrundfarbe mittels der Methode `getColor` ableiten.
Nun bleibt uns nur noch, diese Einstellungen in unsere Klasse `JavaUhr` einzubauen. Betrachten wir die angepasste Methode `zeigeEinstellungen`:

```
  private void zeigeEinstellungen() {
    // Erzeuge nur neue Objekte, wenn der Dialog noch nicht existiert
    if (einstellungen == null) {
      // Initialisiere die Look-and-feel-Einstellung
      SetzeLookAndFeel lookAndFeel=new SetzeLookAndFeel();
```

```
      // Initialisiere die Darstellungs-Einstellungen
      SetzeDarstellung darstellung=new SetzeDarstellung(anzeige,this);
      // Initialisiere den Dialog
      einstellungen =
      new Einstellungen(this,true,new Einstellungen.Einstellung[]{
        lookAndFeel,darstellung
      });
      // Gegen Ende noch einige letzte Einstellungen
      lookAndFeel.setZuAktualisieren(new Window[] {
        this,einstellungen
      });
    }
    // Mache den Dialog sichtbar
    einstellungen.pack();
    einstellungen.setVisible(true);
  }
```

Wie wir sehen, ist der Aufwand dank unseres strukturierten Designs minimal.

6.6 Iteration 6: Vom Fenster in den Browser

Wir kommen nun zum Applet-Teil unseres Uhren-Projektes. Ziel dieser Iteration ist es, unsere Uhr in Form eines Applets auf dem Web-Browser darzustellen.

6.6.1 Schritt 1: Auf den Schirm

Beginnen wir damit, eine erste lauffähige Version zu generieren. Wie Sie sich nach den vorherigen Abschnitten wahrscheinlich denken können, ist das gar nicht so schwer. Wir erzeugen eine neue Klasse `UhrenApplet`, die sich von `JApplet` ableitet. In dieser Klasse stellen wir unsere Uhr dar, indem wir unsere `SwingUhr` mittels der Methode `setContentPane()` als alleinigen Inhalt setzen:

```
1  import javax.swing.*;
2
3  /** Diese Klasse verwendet die SwingUhr, um die Uhrzeit
4    * in Form eines Applet darzustellen.
5    **/
6  public class UhrenApplet extends JApplet {
7
8    /** Intern verwenden wir eine SwingUhr */
9    private SwingUhr uhr;
10
11   /** Init-Methode */
12   public void init() {
13     // Instantiiere eine SwingUhr und mache
14     // diese zum alleinigen Inhalt des Applet
15     uhr = new SwingUhr();
16     setContentPane(uhr);
17   }
18
19   /** Start-Methode */
20   public void start() {
```

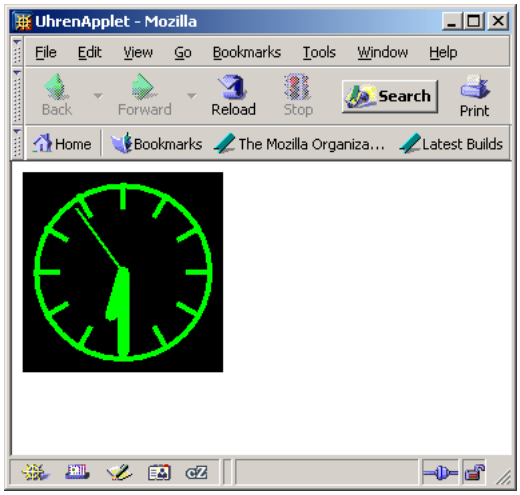


Abbildung 6.6: Uhrenapplet im ersten Schritt

```
21      uhr.aktivieren();
22    }
23
24    /** Stop-Methode */
25    public void stop() {
26      uhr.beenden();
27    }
28
29  }
```

Anschließend benötigen wir nur noch eine entsprechende HTML-Seite, um das Applet in einem Browser anzeigen zu können:

```
1  <html>
2    <head>
3      <title>
4        UhrenApplet
5      </title>
6    </head>
7    <body>
8      <applet code="UhrenApplet.class" width=150 height=150>
9      </applet>
10   </body>
11 </html>
```

Abbildung 6.6 zeigt das fertige Ergebnis auf dem Bildschirm. Ist unser Praxisabschnitt nun zu Ende?

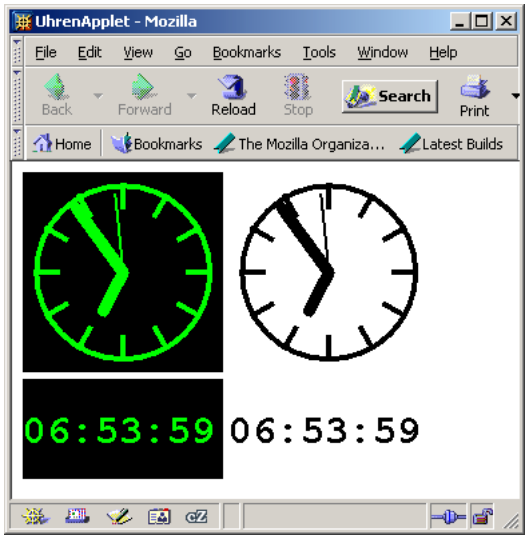


Abbildung 6.7: Das fertige Applet

6.6.2 Schritt 2: Eine Frage der Einstellung

Wie Sie sicher vermutet haben, lautet die Antwort auf obige Frage „nein“. Momentan können wir lediglich die Größe unserer Uhr bestimmen. Wie sieht es aber mit digitaler Anzeige oder dem Setzen von Vorder- und Hintergrundfarbe aus? Da uns nun unser Einstellungs-Dialog fehlt, müssen wir diese Parameter in der HTML-Datei setzen können.
Beginnen wir damit, unsere HTML-Datei entsprechend zu erweitern. Anbei die verbesserte Variante, die nun vier verschiedene Applets darstellen soll (vergleiche Abbildung 6.7). Wir haben drei neue Applet-Parameter eingeführt:

- Einen Parameter `darstellung`, der das Applet mit der Einstellung `digital` auf digitale Darstellung schaltet.
- Einen Parameter `vordergrund`, mit dessen Hilfe sich die Vordergrundfarbe des Applets setzen lässt.
- Einen Parameter `hintergrund`, mit dessen Hilfe sich die Hintergrundfarbe des Applets setzen lässt.

Die beiden Parameter für die Farbe sind hierbei ganzzahlig und kodieren die Rot-, Grün- und Blauwerte einer Farbe. Unsere neue HTML-Datei sieht wie folgt aus:

```
1  <html>
2    <head>
3      <title>
4        UhrenApplet
5      </title>
```

```
6     </head>
7     <body>
8       <applet code="UhrenApplet.class" width=150 height=150>
9       </applet>
10
11      <applet code="UhrenApplet.class" width=150 height=150>
12        <param name="vordergrund" value="0"/>
13        <param name="hintergrund" value="16777215"/>
14      </applet>
15
16      <applet code="UhrenApplet.class" width=150 height=75>
17        <param name="darstellung" value="digital"/>
18      </applet>
19
20      <applet code="UhrenApplet.class" width=150 height=75>
21        <param name="darstellung" value="digital"/>
22        <param name="vordergrund" value="0"/>
23        <param name="hintergrund" value="16777215"/>
24      </applet>
25    </body>
26  </html>
```

Damit sich diese Einstellungen in unserem Applet widerspiegeln, erweitern wir unsere `init`-Methode. Die einzelnen Werte lassen sich mit der Methode `getParameter()` auslesen. Für das Umwandeln unserer Farbwerte in `Color`-Objekte stellt die Klasse `java.awt.Color` eine statische Hilfsmethode namens `decode` zur Verfügung:

```
// 1. Analog oder Digital?
String darstellung = getParameter("darstellung");
if (darstellung != null &&
  darstellung.trim().toUpperCase().equals("DIGITAL"))
    uhr.setDigital();
else
    uhr.setAnalog();
// 2. Vordergrundfarbe?
String vordergrund = getParameter("vordergrund");
if (vordergrund != null)
  try {
    uhr.setForeground(Color.decode(vordergrund));
  }
  catch(Exception e) {
    e.printStackTrace();
  }
// 3. Hintergrundfarbe?
String hintergrund = getParameter("hintergrund");
if (hintergrund != null)
  try {
    uhr.setBackground(Color.decode(hintergrund));
  }
  catch(Exception e) {
    e.printStackTrace();
  }
```

6.6.3 Schritt 3: Alles hübsch verpackt

Im letzten Schritt unseres Praxisbeispieles wollen wir uns um die Auslieferung eines Applets kümmern. Werfen wir einen Blick in unser Verzeichnis:

Konsole

```
C:\buch2\Code\praxis-c\uhr6\version2>dir
 Volume in drive C is HP_PAVILION
 Volume Serial Number is 50E2-5DBF

 Directory of C:\buch2\Code\praxis-c\uhr6\version2

04.06.2003  07:04 AM    <DIR>          .
04.06.2003  07:04 AM    <DIR>          ..
31.05.2003  12:51 PM             3.101 AnalogAnzeige.class
25.05.2003  11:10 PM               157 Anzeige.class
25.05.2003  11:10 PM             1.115 DigitalAnzeige.class
27.05.2003  09:29 PM               548 Einstellungen$1.class
27.05.2003  09:29 PM               506 Einstellungen$2.class
27.05.2003  09:29 PM               514 Einstellungen$3.class
27.05.2003  09:29 PM               298
  Einstellungen$Einstellung.class
27.05.2003  09:29 PM             1.905 Einstellungen.class
03.06.2003  07:13 AM               476 JavaUhr$1.class
03.06.2003  07:13 AM               476 JavaUhr$2.class
03.06.2003  07:13 AM               449 JavaUhr$3.class
03.06.2003  07:13 AM             2.392 JavaUhr.class
03.06.2003  07:13 AM             2.400 SetzeDarstellung.class
27.05.2003  09:29 PM             2.294 SetzeLookAndFeel.class
25.05.2003  11:10 PM               143 Steuerung.class
03.06.2003  07:13 AM             1.886 SwingUhr.class
25.05.2003  11:10 PM               462 Systemzeit.class
04.06.2003  06:49 AM             1.209 UhrenApplet.class
04.06.2003  06:53 AM               725 UhrenApplet.html
04.06.2003  07:26 AM             1.336 UhrenApplet.java
25.05.2003  11:10 PM               242 Uhrwerk.class
              21 File(s)         22.634 bytes
               2 Dir(s)  14.321.483.776 bytes free
```

Wie Sie sehen, haben wir eine Menge `class`-Dateien, von denen einige auch in unserem Applet verwendet werden. Was auf der lokalen Festplatte kein Problem darstellt, kostet im Internet massiv Zeit. Jede dieser `class`-Dateien muss vom Browser explizit angefordert werden; das Programm funktioniert nicht, solange auch nur eine einzige Datei fehlt! Wäre es nicht viel besser, wenn man stattdessen das ganze Programm auf einmal laden könnte?
Sie kennen die Autoren inzwischen gut genug um zu wissen, dass derartige Fragen nicht ohne Grund gestellt werden. Java bietet die Möglichkeit, Klassen in

einem Archiv zusammenzufassen. Dieses Java-Archiv, oder abgekürzt **jar**, kann dann von einem Web-Browser in einem Schritt geladen werden.
Das eigentliche Programm zum Einpacken dieser Dateien wird mit der Java-Entwicklungsumgebung geliefert und lässt sich von der Kommandozeile aus starten[5]:

Konsole

```
C:\buch2\myCode\uhr6>jar cvf uhr.jar *.class
added manifest
adding: AnalogAnzeige.class(in = 3101) (out= 1814)(deflated 41%)
adding: Anzeige.class(in = 157) (out= 131)(deflated 16%)
adding: DigitalAnzeige.class(in = 1115) (out= 677)(deflated 39%)
adding: Einstellungen$1.class(in = 548) (out= 361)(deflated 34%)
adding: Einstellungen$2.class(in = 506) (out= 334)(deflated 33%)
adding: Einstellungen$3.class(in = 514) (out= 340)(deflated 33%)
adding: Einstellungen$Einstellung.class(in = 298)
  (out= 200)(deflated 32%)
adding: Einstellungen.class(in = 1905) (out= 1066)(deflated 44%)
adding: JavaUhr$1.class(in = 476) (out= 329)(deflated 30%)
adding: JavaUhr$2.class(in = 476) (out= 330)(deflated 30%)
adding: JavaUhr$3.class(in = 449) (out= 313)(deflated 30%)
adding: JavaUhr.class(in = 2392) (out= 1335)(deflated 44%)
adding: SetzeDarstellung.class(in = 2400)
  (out= 1323)(deflated 44%)
adding: SetzeLookAndFeel.class(in = 2294)
  (out= 1305)(deflated 43%)
adding: Steuerung.class(in = 143) (out= 122)(deflated 14%)
adding: SwingUhr.class(in = 1886) (out= 987)(deflated 47%)
adding: Systemzeit.class(in = 462) (out= 300)(deflated 35%)
adding: UhrenApplet.class(in = 1209) (out= 733)(deflated 39%)
adding: Uhrwerk.class(in = 242) (out= 179)(deflated 26%)
```

Unser Ergebnis ist eine Datei namens `uhr.jar`. Diese können wir dem Browser in unserer HTML-Datei bekanntmachen, indem wir den Parameter `archive` verwenden:

```
1  <html>
2    <head>
3      <title>
4        UhrenApplet
5      </title>
6    </head>
7    <body>
8      <applet code="UhrenApplet.class" archive="uhr.jar"
9        width=150 height=150>
10     </applet>
11
```

[5] Theoretisch können Sie stattdessen auch ein visuelles Archivprogramm verwenden, wenn dieses Archive im Zip-Format erstellt.

```
12      <applet code="UhrenApplet.class" archive="uhr.jar"
13        width=150 height=150>
14        <param name="vordergrund" value="0"/>
15        <param name="hintergrund" value="16777215"/>
16      </applet>
17
18      <applet code="UhrenApplet.class" archive="uhr.jar"
19        width=150 height=75>
20        <param name="darstellung" value="digital"/>
21      </applet>
22
23      <applet code="UhrenApplet.class" archive="uhr.jar"
24        width=150 height=75>
25        <param name="darstellung" value="digital"/>
26        <param name="vordergrund" value="0"/>
27        <param name="hintergrund" value="16777215"/>
28      </applet>
29    </body>
30  </html>
```

Selbst wenn Sie nun alle `class`-Dateien aus dem Verzeichnis löschen, wird die HTML-Seite weiterhin funktionieren. Der Browser bezieht seine Dateien nun aus dem angegebenen Jar-File.
Jar-Archive sind übrigens nicht nur für die Verwendung bei Applets gedacht. Fast jede Klassenbibilothek, die Sie aus dem Internet herunterladen, finden Sie in diesem Format vor. Denken Sie zum Beispiel an die Prog1Tools ...

6.7 Iteration 7: Die Zeit steht nicht still

Auf den folgenden Seiten werden wir unser neu erworbenes Wissen über Threads einsetzen, um über die finale Hürde zu einer vollwertigen Uhrenanwendung zu springen: wir sorgen dafür, dass unsere Uhr „tickt".
Bis zu diesem Moment ist unsere Uhr nicht viel mehr als ein Standbild: die Zeit wird nicht aktualisiert, die Zeiger bewegen sich nicht. Wir haben zwar alle Voraussetzungen geschaffen,[6] so dass wir unserer Uhr mit wenigen Handgriffen Leben einhauchen können, bislang hatten wir uns aber auf die Grafik konzentriert. Das wird sich nun ändern – Zeit, sich noch einmal mit dem Konzept von **Thread** im Buch vertraut zu machen ...
Anhand unseres Designs [7] wird ziemlich schnell klar, welche unserer Klassen die aktive Rolle übernehmen sollte: unsere `Steuerung`, also konkret die Klasse `SwingUhr` war von Anfang an dafür vorgesehen. Wir erweitern unsere Klassendefinition deshalb um das Interface `Runnable`:

```
public class SwingUhr extends JPanel
  implements Steuerung,Runnable {
```

[6] Wir haben ein `Uhrwerk`, das die aktuelle Zeit in Erfahrung bringt, und eine `Anzeige`, die besagte Uhrzeit visuell darstellt. Wir haben sogar eine `Steuerung` vorgesehen, die die zyklische Aktualisierung bewerkstelligen soll.
[7] Erinnern Sie sich an Abbildung 6.1 auf Seite 164 ?

Die Implementierung der Methode `run` ist denkbar einfach. Solange der Thread nicht unterbrochen wird, durchlaufen wir eine Endlosschleife. Innerhalb dieser Schleife führen wir die folgenden Aktionen durch:

- Wir fordern unsere beiden Anzeigen auf, sich zu aktualisieren.
- Wir fordern den Thread auf, für eine Sekunde zu schlafen.

```
public void run() {
  // Wir durchlaufen eine (beinahe) Endlos-Schleife
  while(!Thread.currentThread().isInterrupted()) {
    // Aktualisiere die Zeit
    digital.zeigeZeit();
    analog.zeigeZeit();
    // Schlafe fuer 1000 Millisekunden
    try {
      Thread.sleep(1000);
    }
    catch(InterruptedException e) {
      Thread.currentThread().interrupt();
    }
  }
}
```

Das war ja gar nicht so schwer! Wie sorgen wir aber dafür, dass die Methode `run` auch aufgerufen wird? An dieser Stelle kommen die bislang verkümmerten Methoden `aktivieren` und `beenden` unserer Klasse ins Spiel. Diese beiden Methoden sind wie geschaffen dafür, einen Thread zu aktivieren oder anzuhalten!
Im ersten Schritt definieren wir eine Instanzvariable `unruh`, die unseren gerade aktiven Thread beinhaltet:

```
private Thread unruh;
```

Zu Beginn der Lebenszeit unserer Instanz ist diese Variable leer. In der Methode `aktivieren` weisen wir ihr ein Objekt zu und aktivieren selbiges. Um zu verhindern, dass versehentlich zwei Threads gestartet werden, überprüfen wir auch, ob die Variable wirklich leer ist:

```
public void aktivieren() {
  // Falls schon ein Thread laeuft, beenden wir die Methode
  if (unruh != null)
    return;
  // Andernfalls erzeugen wir einen neuen Thread
  unruh = new Thread(this);
  unruh.start();
}
```

Analog verfahren wir, wenn die Methode `beenden` aufgerufen wird. Falls die Variable nicht leer ist, beenden wir den dort gespeicherten Thread. Anschließend setzen wir `unruh` auf `null`, um den Neustart mittels `aktivieren()` wieder zu ermöglichen:

```
public void beenden() {
  // Falls noch kein Thread existiert, beenden wir die Methode
```

```
    if (unruh == null)
      return;
    // Andernfalls beenden wir den Thread
    unruh.interrupt();
    unruh = null;
  }
```

Kaum zu glauben, aber wahr – wir haben es tatsächlich geschafft! Mit der modifizierten Klasse `SwingUhr` sind sowohl unsere `JavaUhr` als auch unser `UhrenApplet` voll funktionsfähig Die analogen Zeiger bewegen sich, die Digitaluhr wird aktualisiert. Wer hätte gedacht, wie sich die einfache Konsolenausgabe

Konsole

```
Es ist gerade 14:16 Uhr und 04 Sekunden.
```

aus Abschnitt 6.1 zu einer derartigen Anwendung mausert? Iterative Entwicklung macht's möglich!

6.8 Iteration 8: Ein Zeit–Server

Nach diesem relativ kurzen Abschnitt über Threads kommen wir nun zu unserem „Meisterstück". Bis zum Ende dieses Kapitels wollen wir

- einen Zeitserver geschrieben haben, der über das Netzwerk anderen Rechnern die Uhrzeit mitteilen kann,
- eine Client-Komponente geschrieben haben, die sich mit besagtem Zeitserver „unterhält", und
- dies so in unsere `JavaUhr` integriert haben, dass wir die Uhrzeit des Servers auf unserem Rechner darstellen können.

Wenn Sie unser Mini-Projekt von seinen ersten Babyschritten bis ins Vorschulalter begleitet haben, wird Sie diese Aufgabe kaum schrecken. Wie gelernt, unterteilen wir diesen großen Brocken in kleinere, leichter zu bewältigende Teilschritte. Diese Iteration ist dem Zeitserver gewidmet; der Client folgt in Abschnitt 6.9.

6.8.1 Hätten wir nur *einen* Socket, ...

... dann wäre der Zeitserver nicht mehr als eine lineare Aneinanderreihung von Befehlen. Wir könnten ein einfaches Kommunikations-Protokoll entwerfen, etwa:

- Gib einen Test-String zurueck, wenn der Befehl `TEST` übermittelt wird (Testanfrage).
- Gib die Uhrzeit als `long`-Wert (`System.currentTimeMillis()`) zurück, wenn der Befehl `WIESPAET` übermittelt wird (Zeitanfrage).
- Schließe den Socket, wenn der Befehl `ENDE` übermittelt wird.

Wie wir im Buch gelernt haben, ist der Schritt von einem auf mehrere Sockets allerdings relativ leicht. Wir konzentrieren uns deshalb auf besagtes Protokoll und fassen es in einer Klasse `SocketKommunikation` zusammen:

```
1  import java.net.*;
2  import java.io.*;
3  /** Diese Klasse repraesentiert einen einzelnen Thread,
4    * ueber den eine Uhrzeitkommunikation ablaeuft.
5    **/
6  public class SocketKommunikation extends Thread {
7    /** Der verwendete Socket */
8    private Socket socket;
9    /** Konstruktor */
10   public SocketKommunikation(Socket socket) {
11     this.socket = socket;
12   }
13   /** Ausfuehrung der Kommunikation */
14   public void run() {
15     try {
16       // Erzeuge die notwendigen Ein- und Ausgabestroeme
17       System.out.println("Erzeuge I/O  Stroeme");
18       PrintWriter out = new PrintWriter
19         (socket.getOutputStream(),true);
20       BufferedReader in = new BufferedReader(
21         new InputStreamReader(socket.getInputStream()));
22       // Nun beginne mit der Kommunikation
23       for (String line = in.readLine(); !line.equals("ENDE");
24                                              line = in.readLine()) {
25         // Fall 1: Test String
26         if (line.equals("TEST")) {
27           System.out.println("Testanfrage");
28           out.println("Zeitserver Version 1.0 gefunden");
29         }
30         // Fall 2: Zeitanfrage
31         else if (line.equals("WIESPAET")) {
32           System.out.println("Zeitansage");
33           out.println(System.currentTimeMillis());
34         }
35         // Fall 3: Unbekannt
36         else {
37           System.out.println("Unbekannte Anfrage: \"" + line + '\"');
38           out.println("Unbekannte Anfrage");
39         }
40       }
41     }
42     // Fange eventuelle Exceptions ab
43     catch(Exception e) {
44       e.printStackTrace();
45     }
46     // Gib die Ressourcen wieder frei
47     finally {
48       try {
49         System.out.println("Schliesse Socket");
50         socket.close();
51       }
52       catch(IOException e) {
```

```
53          e.printStackTrace();
54        }
55      }
56    }
57  }
```

Unsere Klasse stellt einen einzelnen `Thread` dar, der das von uns definierte Kommunikationsprotokoll abwickeln kann. Es fehlt uns also nur noch eine allgemeine Steuerungsklasse, die auf eingehende Anfragen lauscht und die entsprechenden Threads initiiert.

6.8.2 Die Klasse `Zeitserver`

In den vorigen Kapiteln haben wir gelernt, wie man beinahe nach „Schema F" mit Hilfe der `ServerSocket`-Klasse einen Server aufbauen kann. Dieses Schema wollen wir auch hier verwenden, um unseren Zeitserver zu realisieren:

```
1  import java.net.*;
2  import java.io.*;
3  /** Diese Klasse steuert die zentrale Verwaltung der
4    * Serverkommunikation unseres JavaUhr Zeitservers
5    **/
6  public class Zeitserver {
7    /** Main-Methode: erzeugt einen Zeitserver auf dem
8      * angegebenen Port.
9      **/
10   public static void main(String[] args) {
11     System.out.println("JavaUhr Zeitserver Version 1.0");
12     // Geh sicher, dass die Argumente stimmen
13     int port = -1;
14     if (args.length == 1) {
15       try {
16         port = Integer.parseInt(args[0]);
17       }
18       catch(NumberFormatException e) {
19         System.out.println("Ungueltige Port-Nummer");
20       }
21     }
22     if (port < 0) {
23       System.out.println("Benutzung: java ZeitServer <portnummer>");
24       System.exit(-1);
25     }
26     // Erzeuge den ServerSocket
27     System.out.println("Erzeuge ServerSocket");
28     ServerSocket s = null;
29     try {
30       s = new ServerSocket(port);
31     }
32     catch(IOException e) {
33       System.out.println("Konnte Socket auf Port " + port
34         + " nicht oeffnen.");
35       System.exit(-2);
36     }
37     // Und nun lauschen wir...
```

```
38      try {
39        while(true) {
40          new SocketKommunikation(s.accept()).start();
41        }
42      }
43      catch(IOException e) {
44        System.out.println("Es ist eine Exception aufgetreten.");
45        e.printStackTrace();
46      }
47      // Bevor wir beenden, schliessen wir den Socket
48      finally {
49        try {
50          s.close();
51          System.exit(-3);
52        }
53        catch(IOException e) {
54          e.printStackTrace();
55          System.exit(-4);
56        }
57      }
58    }
59  }
```

Nachdem wir unsere Klassen compiliert haben, ist der Zeitserver auch schon einsatzbereit:

Konsole

```
java Zeitserver 1099
JavaUhr Zeitserver Version 1.0
Erzeuge ServerSocket
```

Zugegebenermaßen – mit diesem Programm fühlt man sich schon ein wenig an die ersten Schritte in diesem Grundkurs erinnert. Unsere Klasse ist lediglich Behälter für eine `main`-Methode, in der wir eigentlich recht „prozeduralen“ Programmierstil pflegen. Bedenken Sie aber, dass hinter der schlicht erscheinenden Methode ein komplexes Objekt-Modell des Java-Netzwerkpakets mit Multithreading und komplexen I/O-Protokollen steckt. Es ist diese hervorragende objektorientierte Implementierung, die unsere letzten Beispielkapitel nun so einfach erscheinen lässt.

6.8.3 Ein Testprogramm

Wie können wir sichergehen, dass unser Zeitserver tatsächlich funktioniert? Ein Testprogramm muss her! Wir modifizieren unser allererstes Uhrenprogramm, die Klasse `WieSpaet` aus Abschnitt 6.1. Diese soll die Uhrzeit vom Server erfragen können.
Dazu benötigen wir ein neues `Uhrwerk`. Wir definieren eine Klasse `Serverzeit`, die statt der lokalen Systemzeit unseren `Zeitserver` befragt:

```
public class Serverzeit implements Uhrwerk {
```

```
/** Der Rechnername des Zeitservers */
private String rechner;

/** Die Nummer des Ports */
private int port;

/** Konstruktor */
public Serverzeit(String rechner, int port) {
  this.rechner = rechner;
  this.port = port;
}
```

Im ersten Schritt definieren wir eine private Hilfsmethode `anfrage`, die in folgenden Schritten arbeitet:

- Öffne die Verbindung zum Zeitserver.
- Übermittle den als Parameter übergebenen Kommandostring.
- Hole vom Server das bereitgestellte Ergebnis ab.
- Beende die Kommunikation.
- Gib das Ergebnis zurück.

Unsere Methode stellt also einen kompletten Kommunikationszyklus dar. Anbei die Realisierung dieser Idee:

```
private String anfrage(String kommando) {
  Socket s = null;
  try {
    s = new Socket(rechner,port);
    PrintWriter out = new
      PrintWriter(s.getOutputStream(),true);
    BufferedReader in = new BufferedReader(
      new InputStreamReader(s.getInputStream()));
    out.println(kommando);
    String resultat = in.readLine();
    out.println("ENDE");
    out.flush();
    return resultat;
  }
  catch(Exception e) {
    e.printStackTrace();
  }
  finally {
    if (s != null) {
      try {
        s.close();
      }
      catch (IOException e) {
        e.printStackTrace();
      }
    }
  }
  return null;
}
```

Anhand dieser Methode ließe sich also beispielsweise das Erfragen unseres Teststrings durch folgenden Dreizeiler erledigen:

```
public String testString() {
  return anfrage("TEST");
}
```

Die Realisierung unserer Methode `getZeit` ist nur unwesentlich komplizierter. Wir müssen lediglich die vom Server erhaltene Information in einen `Long`-Wert umwandeln und diesen dann in ein Datumsobjekt überführen. Sollte hierbei ein Problem auftreten, verwenden wir als Notlösung die lokale Systemzeit:

```
public Date getZeit() {
  try {
    String zeit = anfrage("WIESPAET");
    return new Date(Long.parseLong(zeit));
  }
  catch(Exception e) {
    e.printStackTrace();
    return new Date();
  }
}
```

Um dieses `Uhrwerk` in unserem Programm `WieSpaet` einsetzen zu können, müssen wir nun lediglich dessen `main`-Methode anpassen. Wir lesen Servername und Port von der Tastatur ein und erzeugen unsere `Serverzeit`-Instanz:

```
public static void main(String[] args) throws Exception {
  // Instantiiere eine Uhr
  Uhrwerk uhrwerk = null;
  if (args.length == 2)
    uhrwerk = new Serverzeit(args[0],Integer.parseInt(args[1]));
  else
    uhrwerk = new Systemzeit();
  // Instantiiere ein einzelnes Steuerungsobjekt
  Steuerung steuerung =
    new WieSpaet(uhrwerk,new KonsolenAnzeige());
  // Aktiviere die Steuerung und gib somit die Zeit aus
  steuerung.aktivieren();
  // Beende die Steuerung und das Programm
  steuerung.beenden();
}
```

Nun wird es Zeit, unser erweitertes Programm zu starten. Wie erhofft, erhalten wir eine aktuelle Zeitausgabe:

Konsole

```
java WieSpaet localhost 1099
Es ist gerade 08:59 Uhr und 18 Sekunden.
```

Handelt es sich hierbei aber auch um die vom Server abgeholte Zeit? Werfen wir einen Blick auf die Konsole unseres Zeitservers:

Konsole

```
java Zeitserver 1099
JavaUhr Zeitserver Version 1.0
Erzeuge ServerSocket
Erzeuge I/O  Stroeme
Zeitansage
Schliesse Socket
```

Die Bildschirmausgabe spricht für sich. Der Zeitserver hat eine Zeitansage durchgeführt und danach die Kommunikation beendet. Das Programm funktioniert!

6.9 Iteration 9: Wenn's am schönsten ist, ...

... dann sollte man ja bekanntlich aufhören. Leider ist es auch für uns nun beinahe soweit. In der letzten Iteration unseres Uhrenprojekts werden wir unsere `JavaUhr` um die Verwendung eines Zeitservers erweitern. Konkret bedeutet dies einen weiteren Dialog, in dem sich auf Wunsch Servername und Port unseres Zeitservers einstellen lassen.
Natürlich ist dieser Teil nicht unbedingt diesem Dialog gewidmet – wir haben die Abschnitte über Swing bereits hinter uns gelassen. Wie Sie aber sehen werden, ist auch die Kommunikation mit der GUI nicht vollkommen ohne Threads zu bewältigen. Somit sind wir also wieder einmal beim Thema.

6.9.1 Einige Vorbereitungen

Wie lässt sich die Uhrzeit unseres Servers am besten aktualisieren? Die einfachste Lösung wäre es, schlicht und ergreifend unsere neue Uhrwerks-Klasse `Serverzeit` zu verwenden. Wir könnten unsere `SwingUhr` um Methoden erweitern, mit denen sich das Uhrwerk austauschen lässt:

```
/** Diese Methode laesst die SwingUhr statt der Systemzeit ein
  * anderes Uhrwerk verwenden.
  **/
public void setzeUhrwerk(Uhrwerk uhrwerk) {
  digital.setUhrwerk(uhrwerk);
  analog.setUhrwerk(uhrwerk);
}
/** Diese Methode verwirft Aenderungen am Uhrwerk und verwendet
  * wieder die originale Systemzeit.
  **/
public void setzeSystemzeit() {
  setzeUhrwerk(systemZeit);
}
```

Leider hat diese Idee einen kleinen Schönheitsfehler. Das in der `SwingUhr` verwendete `Uhrwerk` wird von der Methode `run()` zweimal pro Sekunde aufgerufen. Dies bedeutet eine Menge Netzwerk-Kommunikation – und das lediglich, um

die Zeit vom Server zu erfahren. Schlimmer noch: Abhängig davon, wie gut oder schlecht die Verbindung über das Netzwerk ist, kann die Anfrage an den Server durchaus auch ein wenig länger als gedacht dauern. Während also unser Thread auf die Antwort vom Zeitserver wartet, „friert" die Uhrenanzeige ein. Das sieht nicht nur unprofessionell aus, es ist auch eigentlich vollkommen unnötig. Um wie viel Zeit mag sich die Uhr des Zeitservers seit der letzten Sekunde wohl verstellt haben?
Um diese Probleme zu vermeiden, wählen wir einen anderen Ansatz: wir verwenden die Verbindung zum Zeitserver nur, um unsere lokale Uhr zu *stellen*. Dieser „Uhrenvergleich" kann alle paar Minuten stattfinden und von einem unabhängigen Thread durchgeführt werden. Er beeinflusst unsere Darstellung also nicht.
Wir werden uns im nächsten Abschnitt mit dieser speziellen Threadklasse `Synchronisierer` beschäftigen. Um dies allerdings vorzubereiten, benötigen wir zuerst ein `Uhrwerk`, das sich stellen lässt. Die folgende Klasse `EingestellteZeit` erfüllt diese Anforderung:

```
1  import java.util.Date;
2  /** Dieses Uhrwerk wird mit einer Zeit eingestellt und verwendet die
3    * interne Systemuhr, um neue Zeiten zu berechnen.
4    **/
5  public class EingestellteZeit implements Uhrwerk {
6    /** Der Unterschied zwischen eingestellter Zeit
7      * und Systemzeit in Millisekunden.
8      **/
9    private long drift;
10   /** Gibt die aktuelle Uhrzeit in Form eines Date-Objektes zurueck.
11     * @return die aktuelle Zeit
12     **/
13   public Date getZeit() {
14     return new Date(System.currentTimeMillis() + drift);
15   }
16   /** Stellt das Werk dieser Uhr auf eine bestimmte Zeit. Diese
17     * Methode ist optional und muss nicht immer implementiert sein.
18     * @param zeit die aktuelle Uhrzeit
19     * @exception UnsupportedOperationException falls
20     *  diese Methode nicht unterstuetzt wird
21     **/
22   public void setZeit(Date zeit)
23     throws UnsupportedOperationException {
24     drift = zeit.getTime() - System.currentTimeMillis();
25   }
26 }
```

Ihnen wird wahrscheinlich aufgefallen sein, dass dies das erste Mal ist, dass wir die Methode `setZeit` tatsächlich ausformulieren. Dabei haben wir sie schon von Anfang an im Entwurf auf Seite 164 vorgesehen! War das Weitblick im Design? Instinkt? Oder pures Glück?
An dieser Stelle möchten die Autoren vor einem klassischen Designfehler warnen, der von Martin Fowler und Kent Beck in [2] als „spekulative Flexibilität" bezeichnet wurde. So mancher ist schon in diese Falle getappt, und auch wir sind keine Ausnahme. Spekulative Flexibilität ist der Versuch, im Design bereits alle mögli-

chen Erweiterungen vorauszuahnen, die zu Lebzeiten eines Entwurfs an ein Programm herangetragen werden. Ohne bereits einen konkreten Anwendungsfall zu kennen, werden Entwürfe darauf getrimmt, mit allen möglichen – fiktiven – Kundenwünschen fertig zu werden. Das Resultat ist ein Entwurf, der zu umfangreich ist. Solche Klassen sind nicht nur langwieriger zu implementieren, sie sind oft auch fehleranfälliger. Je mehr Funktionalität man hat, desto mehr Fehler können sich einschleichen. Schließlich sind wir alle nur Menschen! Insbesondere Funktionen, bei denen man momentan noch keinen konkreten Anwendungsfall hat, sind üblicherweise schwerer zu testen und deshalb selten ausgereift.
Sind wir hier also in die klassische Entwurfsfalle getappt? In dubio pro reo – nennen wir es einfach Weitblick im Design …

6.9.2 Uhrenvergleich

Kommen wir nun zur Klasse `Synchronisierer`. Diese Klasse soll in der Lage sein, die Uhrzeit von einem Uhrwerk `quelle` zu lesen und in ein Uhrwerk `ziel` zu schreiben. Dies soll in einem eigenen Thread `thread` geschehen, der alle `takt` Millisekunden aktiv wird. Wir definieren entsprechende Instanzvariablen in unserer Klasse:

```
public class Synchronisierer implements Steuerung, Runnable {
  /** Die Uhr, von der die Zeit gelesen wird */
  private Uhrwerk quelle;
  /** Die Uhr, die gestellt werden soll */
  private Uhrwerk ziel;
  /** Der Takt, in dem die Aktualisierung vonstatten gehen
    * soll (in Millisekunden, mindestens 100)
    **/
  private int takt;
  /** Der Thread, der gerade aktiv ist */
  private Thread thread;
  /** Konstruktor. */
  public Synchronisierer(Uhrwerk ziel) {
    this.ziel = ziel;
    takt = 100;
  }
```

Unsere Klasse realisiert hierbei das Interface `Runnable`, so dass unser Thread die Ausführungslogik innerhalb der Methode wiederfindet. Wir leiten die Klasse ferner von `Steuerung` ab und geben somit vor, dass wir die Synchronisierung durch die Methoden `aktivieren()` und `beenden()` starten und anhalten wollen. Diesen Fakt wollen wir in den Methoden `setTakt()` und `setQuelle()` ausnutzen: wir halten unseren aktuellen Thread erst an, bevor wir Werte innerhalb der Klasse verändern:

```
  /** Setze den fuer die Aktualisierung zu verwendenden Takt
    * (in Millisekunden, mindestens 100)
    **/
  public void setTakt(int takt) {
    boolean aktiv = thread != null;
```

```
    if (aktiv)
      beenden();
    this.takt = Math.max(100,takt);
    if (aktiv)
      aktivieren();
  }
  /** Setze die fuer die Aktualisierung verwendete Quelle */
  public void setQuelle(Uhrwerk quelle) {
    boolean aktiv = thread != null;
    if (aktiv)
      beenden();
    this.quelle = quelle;
    if (aktiv)
      aktivieren();
  }
```

Wir speichern also den gerade aktiven Thread in einer Instanzvariable. Ist diese `null`, ist gerade kein Thread aktiviert. Diese Logik spiegelt sich auch in der konkreten Ausprägung von `aktivieren()` und `beenden()` wider:

```
  /** Diese Methode signalisiert der Steuerung, dass
    * sie mit ihrer Arbeit beginnen soll.
    **/
  public void aktivieren() {
    synchronized(this) {
      if (thread != null)
        return;
      thread = new Thread(this);
      thread.start();
    }
  }
  /** Diese Methode signalisiert der Steuerung, dass
    * sie ihre Arbeit jetzt beenden kann.
    **/
  public void beenden() {
    synchronized(this) {
      if (thread == null)
        return;
      thread.interrupt();
      thread = null;
    }
  }
```

Nun fehlt uns also nur noch die eigentliche Ablauflogik in der Methode `run`. Unser momentan aktiver Thread funktioniert nur mit einer bereits gegebenen `quelle` und einem vordefinierten `takt`. Sobald wir mittels der set-Methoden eines von beiden ändern, wird der aktuelle Thread beendet und ein neuer gestartet. Anstatt also mit sich verändernden Instanzvariablen kämpfen zu müssen, können wir die für diesen Thread verwendeten Werte einmal aus der Instanz auslesen:

```
  public void run() {
    // Kopiere die fuer diesen Thread gueltigen
    // Quelle und Takt in lokale Felder
    Uhrwerk quelle = null;
    int takt = 0;
```

```
synchronized(this) {
  quelle = this.quelle;
  takt = this.takt;
}
```

Anschließend wiederholen wir die folgenden Anweisungen, solange der Thread nicht unterbrochen und das Quelluhrwerk nicht leer sind:

```
while(!Thread.currentThread().isInterrupted() && quelle != null) {
```

- Wir übertragen die Zeit von der `quelle` ins `ziel`:

```
ziel.setZeit(quelle.getZeit());
```

- Anschließend halten wir den Thread für die in `takt` vorgegebene Zeit an:

```
Thread.sleep(takt);
```

Somit ist unsere Klasse `Synchronisierer` komplett. Wir müssen sie lediglich noch in unsere Anwendung einbauen.

6.9.3 Der Einstellungs-Dialog

Eine neue Einstellung in unser Einstellungs-Fenster einzubauen ist inzwischen beinahe banal. Wir wollen den Benutzer zwischen lokaler Zeit und einem Zeitserver wählen lassen; entsprechend wählen wir die grafischen Elemente für unsere `Einstellung`-Klasse:

```
public class SetzeZeitserver extends JPanel
  implements Einstellungen.Einstellung {
  /** Diese SwingUhr wird mit den Einstellungen beeinflusst */
  private SwingUhr uhr;
  /** Dieses Objekt synchronisiert die Uhrzeiten */
  private Synchronisierer sync;
  /** Dieses Fenster ist der Dialog, in dem die Einstellungen
    * durchgefuehrt werden
    **/
  private Window dialog;
  /** RadioButtons: Systemzeit */
  private JRadioButton system;
  /** RadioButton: Zeitserver */
  private JRadioButton server;
  /** Textfeld: ServerName */
  private JTextField serverName;
  /** Textfeld: Port */
  private JTextField port;
  /** Textfeld: Taktrate fuer die Aktualisierung */
  private JTextField takt;
```

Die Gestaltung des grafischen Layouts sei Ihnen an dieser Stelle erspart. Kommen wir stattdessen direkt zu jenem Teil, wo die Benutzereinstellungen in das Programm mit einfließen.

Da der Benutzer Servername und Port über die Tastatur eingibt, können sich natürlich diverse Tippfehler einschleichen. Entsprechende Probleme müssen wir dem Benutzer mitteilen. Wir verwenden Javas Hilfsklasse `JOptionPane`, um eventuelle Probleme als Dialog auf den Bildschirm zu bringen:

```
private void zeigeProblem(String nachricht) {
  JOptionPane.showMessageDialog(dialog,nachricht,getLabel(),
    JOptionPane.WARNING_MESSAGE);
}
```

In unserer Methode `anwenden()` verwenden wir diese Hilfsmethode, um auf eventuelle Eingabefehler hinzuweisen. Wir lesen Servername, Port und Taktrate und speichern diese in lokalen Feldern:

```
public void anwenden() {
  // Verwende Systemzeit
  if (system.isSelected()) {
    uhr.setzeSystemzeit();
    sync.setQuelle(null);
    return;
  }
  // Andernfalls lies die aktuellen Einstellungen
  String serverName = this.serverName.getText();
  int port = 0, takt = 0;
  try {
    port = Integer.parseInt(this.port.getText());
  }
  catch(NumberFormatException e) {
    zeigeProblem("Ungueltige Eingabe fuer ServerPort");
    return;
  }
  try {
    takt = Integer.parseInt(this.takt.getText());
  }
  catch(NumberFormatException e) {
    zeigeProblem("Ungueltige Eingabe fuer Zeittakt");
    return;
  }
```

Im nächsten Schritt wollen wir die Verbindung zu unserem Zeitserver testen. Wir erzeugen ein neues `Serverzeit`-Objekt und verwenden die Methode `testString()`, um eine kurze Verbindung zum Zeitserver aufzubauen:

```
System.out.println("Teste Verbindung zu " +serverName +':' +takt);
Serverzeit verbindung = new Serverzeit(serverName,port);
String test = verbindung.testString();
if (test != null)
  System.out.println(test);
else {
  zeigeProblem("Kann Verbindung zu " + serverName + ':' +
    port + " nicht herstellen.");
  return;
}
```

War dies erfolgreich, übernehmen wir die neue `verbindung` und den `takt` in unseren `Synchronisierer`:

```
sync.setTakt(takt * 1000);
sync.setQuelle(verbindung);
uhr.setzeUhrwerk(sync.getZiel());
```

Unsere Klasse ist somit komplett. Wir müssen sie lediglich noch in unsere `JavaUhr` einbauen. Dies machen wir in folgenden Schritten:

- Zuerst definieren wir eine lokale Instanzvariable für unseren Synchronisierer:

```
private Synchronisierer sync;
```

- Dieser `Synchronisierer`, sofern nicht `null`, wird immer genau dann aktiviert, wenn unser Fenster sichtbar gemacht wird:

```
public void setVisible(boolean value) {
  if (value != isVisible()) {
    super.setVisible(value);
    if (value) {
      anzeige.aktivieren();
      if (sync != null)
        sync.aktivieren();
    }
    else
      anzeige.beenden();
      if (sync != null)
        sync.beenden();
  }
}
```

- Der `Synchronisierer` wird erzeugt, wenn wir das erste Mal unseren Einstellungs-Dialog öffnen. Zu diesem Zeitpunkt (das Fenster ist momentan sichtbar) wird das Objekt auch aktiviert:

```
private void zeigeEinstellungen() {
  // Erzeuge nur neue Objekte, wenn
  // der Dialog noch nicht existiert
  if (einstellungen == null) {
    // Initialisiere die Look-and-feel-Einstellung
    SetzeLookAndFeel lookAndFeel = new SetzeLookAndFeel();
    // Initialisiere die Darstellungs-Einstellungen
    SetzeDarstellung darstellung
      = new SetzeDarstellung(anzeige,this);
    // Initialisiere die Zeitserver-Einstellungen
    sync = new Synchronisierer(new EingestellteZeit());
    SetzeZeitserver zeitserver
      = new SetzeZeitserver(anzeige,sync);
    // Initialisiere den Dialog
    einstellungen =
    new Einstellungen(this,true,new Einstellungen.Einstellung[]{
      lookAndFeel,darstellung,zeitserver
    });
    // Gegen Ende noch einige letzte Einstellungen
    lookAndFeel.setZuAktualisieren(new Window[] {
      this,einstellungen
    });
```

```
            zeitserver.setzeDialog(einstellungen);
            sync.aktivieren();
          }
          // Mache den Dialog sichtbar
          einstellungen.pack();
          einstellungen.setVisible(true);
        }
```

Mit diesen geringfügigen Modifikationen sind wir nun so weit: Die neue Einstellung ist in unsere Applikation integriert. *Unser Programm ist komplett!*

6.9.4 Zusammenfassung

Mit Abschluss unserer neunten Iteration haben wir es geschafft: unser Uhrenprojekt ist erfolgreich abgeschlossen. Zu diesem Zeitpunkt mag es interessant sein, ein paar kleine Statistiken über die Entwicklung unserer Anwendung heranzuziehen:

- Am Ende dieses Kapitels besteht unser Uhrenprojekt aus insgesamt 27 Klassen oder Interfaces (anonyme Listener-Klassen mitgezählt). Im ersten Moment klingt diese Zahl recht hoch – wir müssen sie aber in Relation zu den Iterationen setzen.
- Zu Beginn unseres Projektes (Abschnitt 6.1) waren es lediglich drei Klassen und drei Interfaces. Im Schnitt ist unser Programm pro Iteration also um weniger als drei Klassen gewachsen. Oder anders ausgedrückt: Wir haben vom großen Brocken der Aufgabe niemals mehr abgebissen, als wir kauen konnten.
- Von unseren 27 Klassen verblieben fast alle nach der ursprünglichen Version in jenem Zustand, in dem wir sie definiert haben. Lediglich die Klassen `SwingUhr` (vier neue Versionen) und `JavaUhr` (drei neue Versionen) waren regelmäßigen Erweiterungen unterworfen.[8] Diese Änderungen wurden nötig, da wir neue Funktionalität aus den anderen Klassen an diesen zentralen Stellen integrieren mussten. Jede dieser Änderungen bestand allerdings nur aus wenigen Zeilen Code und war zu jeder Zeit vollkommen abwärtskompatibel.

Diese Werte lassen sich unterschiedlich interpretieren. Bitte verstehen Sie sie nicht als Hinweis, unser Projekt sei doch eigentlich gar nicht so schwer gewesen. Keine drei Klassen pro Iteration – ist das denn überhaupt Fortschritt?
Schon Aesop erkannte in einer seiner Fabeln, dass sich ein großes Bündel dünner Zweige nur schwer durchbrechen lässt – es sei denn, man nimmt sich einen Zweig nach dem anderen vor. Ähnlich ist es mit dem iterativen Programmieren. Wenn Sie immer nur jenes Problem angehen, das Sie momentan zu lösen vermögen, mag Ihnen jeder Schritt einfach vorkommen. Nichtsdestotrotz bringt er Sie dem Gesamterfolg unaufhaltsam näher.

[8] Weitere Änderungen fanden in der Klasse `UhrenApplet` (eine Änderung, allerdings in derselben Iteration, in der sie auch entstanden ist) und `WieSpaet` (eine Änderung, um den Zeitserver zu testen) statt.

Literaturverzeichnis

Bücher

[1] T. H. Cormen, Ch. E. Leiserson, R. L. Rivest, C. Stein: *Introduction to Algorithms*. MIT Press, 2001.

[2] M. Fowler, K, Beck, J. Brant, W. Opdyke, D. Roberts: *Refactoring: Improving the Design of Existing Code*. Addison Wesley, 1999.

[3] E. Gamma, R. Helm. R. Johnson, J. Vlissides: *Entwurfsmuster. Elemente wiederverwertbarer objektorientierter Software*. Addison Wesley, 2004.

[4] A. Hunt, D.Thomas: *The Pragmatic Programmer*. Pearson Education, 1999.

[5] A. Hunt, D.Thomas: *Der Pragmatische Programmierer*. Hanser, 2003.

[6] V. Massol, T. Husted: *Junit in Action*. Manning Publications, 2003.

[7] I. Wegener: *Theoretische Informatik*. B. G. Teubner, 2005.

Internet-Links

[8] *Regular Expression Tutorial.*
`http://www.regular-expressions.info/tutorial.html`

[9] *JUnit Homepage.*
`http://www.JUnit.org`

[10] *Extreme Programming.*
`http://www.extremeprogramming.org`

[11] *Cuise Control.*
`http://cruisecontrol.sourceforge.net`

[12] *Bumper-Sticker API Design.*
`http://www.infoq.com/articles/API-Design-Joshua-Bloch`

[13] *Annotations for Software Defect Detection.*
`http://jcp.org/en/jsr/detail?id=305`

[14] *How and When To Deprecate APIs*
`http://java.sun.com/j2se/1.5.0/docs/guide/javadoc/deprecation/deprecation.html`

[15] *Die Klasse* `ImmutableMap.Builder`
`http://google-collections.googlecode.com/svn/trunk/javadoc/com/google/common/collect/ImmutableMap.Builder.html`

[16] *Dependency Injection durch Google Guice*
`http://code.google.com/p/google-guice`

[17] *Getting Started with the Annotation Processing Tool (apt)*
`http://java.sun.com/j2se/1.5.0/docs/guide/apt/GettingStarted.html`