WWW.PEARSON-STUDIUM.DE
CWS
COMPANION WEBSITE

it
informatik

David J. Barnes
Michael Kölling

Java lernen mit BlueJ

Eine Einführung in die objektorientierte Programmierung

4., aktualisierte Auflage

Prentice Hall

PEARSON
Studium

Kapitel

3 Objektinteraktion

Zentrale Konzepte in diesem Kapitel:

- Abstraktion
- Modularisierung
- Objekterzeugung
- Objektdiagramme
- Methodenaufrufe
- Debugger

Java-Konstrukte in diesem Kapitel:

Klassen als Typen, logische Operatoren (`&&`, `||`), Verkettung von Zeichenketten, Modulo-Operator (`%`), Objekterzeugung (`new`), Methodenaufrufe (Punkt-Notation), *this*

In den vorherigen Kapiteln haben wir untersucht, was Objekte sind und wie sie implementiert werden. Insbesondere haben wir Datenfelder, Konstruktoren und Methoden diskutiert, als wir Klassendefinitionen betrachtet haben.

Wir werden nun einen Schritt weitergehen. Für die Konstruktion interessanter Anwendungen reicht es nicht, individuell funktionierende Objekte zu erstellen. Zusätzlich müssen diese Objekte miteinander kombiniert werden, damit sie gemeinsam eine Aufgabe bearbeiten können. In diesem Kapitel werden wir eine kleine Anwendung aus drei Objekten erstellen, in der einzelne Methoden andere Methoden aufrufen müssen, um ihre Aufgabe zu erfüllen.

3.1 Das Uhren-Beispiel

Das Projekt, das wir für die Diskussion von interagierenden Objekten benutzen werden, modelliert die Anzeige einer Digitaluhr. Diese Anzeige zeigt Stunden und Minuten, voneinander getrennt durch einen Doppelpunkt (Abbildung 3.1). Wir gehen vorläufig von einer Uhr mit einer 24-Stunden-Anzeige aus. Die Anzeige zeigt also Zeitpunkte von 00:00 (Mitternacht) bis 23:59 (eine Minute vor Mitternacht). Eine amerikanische 12-Stunden-Anzeige wäre ein wenig komplizierter – deshalb schieben wir diese Variante bis zum Ende dieses Kapitels auf.

Abbildung 3.1: Die Anzeige einer Digitaluhr

3.2 Abstraktion und Modularisierung

Unser erster Ansatz könnte sein, die gesamte Anzeige in einer einzigen Klasse zu realisieren. Schließlich ist es das, was wir bisher getan haben: Klassen erstellen, die eine Aufgabe erfüllen.

Wir gehen das Problem jedoch etwas anders an. Wir werden untersuchen, ob wir bei unserer Aufgabe Teilaufgaben identifizieren können, die wir mit eigenen Klassen modellieren können. Der Grund für dieses Vorgehen ist *Komplexität*. Im Laufe dieses Buches werden die Programmbeispiele, die wir untersuchen, immer komplizierter werden. Einfache Aufgaben wie ein Ticketautomat können als Einzelaufgaben betrachtet werden. Man kann die gesamte Aufgabe betrachten und eine Lösung mit nur einer Klasse vorschlagen. Für kompliziertere Aufgaben reicht dieses Vorgehen nicht mehr aus. Mit zunehmendem Aufgabenumfang wird es auch zunehmend schwieriger, alle Details des Problembereichs im Auge zu behalten.

KONZEPT:

Abstraktion ist die Fähigkeit, Details von Bestandteilen zu ignorieren, um den Fokus der Betrachtung auf eine höhere Ebene lenken zu können.

Das Mittel, mit dem wir zu hohe Komplexität in Aufgaben angehen, ist *Abstraktion*. Wir zerteilen eine Aufgabe in Teilaufgaben, diese wieder in Unterteilaufgaben etc, bis die einzelnen Aufgaben klein genug für eine übersichtliche Lösung sind. Sobald wir eine Unteraufgabe gelöst haben, können wir diesen Teil als erledigt betrachten und als Baustein für die nächste Aufgabe ansehen. Diese Technik ist auch unter der Bezeichnung *Teile und herrsche* bekannt.

Wir wollen dies an einem Beispiel verdeutlichen. Stellen Sie sich Ingenieure bei einem Automobilhersteller vor, die ein neues Fahrzeug entwerfen sollen. Einige dieser Ingenieure betrachten spezifische Aspekte des Fahrzeugs wie die äußere Form, die Größe und die Position des Motors, die Anzahl und Größe der Sitze im Innenraum, den exakten Radstand etc. Ein anderer Ingenieur, der den Motor entwirft (das wird üblicherweise von einem ganzen Team getan, aber wir vereinfachen für diese Diskussion etwas), denkt an die Teile, aus denen ein Motor besteht: die Zylinder, die Einspritztechnik, der Vergaser, die Elektronik etc. Dieser Ingenieur betrachtet den Motor nicht als eine Einheit, sondern als ein komplexes Gebilde aus vielen Einzelteilen. Eines dieser Teile ist eine Zündkerze.

Dann gibt es einen weiteren Ingenieur (vermutlich bei einem anderen Unternehmen), der Zündkerzen entwirft. Er wiederum sieht eine Zündkerze als ein aufwendiges Gebilde aus vielen Einzelteilen an. Er hat möglicherweise aufwendige Studien angestellt, welches Metall für die Kontakte besonders geeignet ist oder welches Material und welcher Fertigungsprozess für die Isolierung verwendet werden soll.

Das Gleiche gilt für viele andere Teile. Ein Designer auf äußerster Ebene sieht einen Reifen als einzelnen Bestandteil an. Ein Ingenieur, der in der Fertigungskette an ganz anderer Stelle positioniert ist, kann hingegen Tage damit verbringen, die ideale chemische Zusammensetzung für das Material der Reifen zu entwerfen. Für den Reifeningenieur ist ein Reifen ein komplexes Gebilde. Das Automobilunternehmen hingegen kauft die Reifen lediglich ein und betrachtet sie als einzelnes Teil eines Fahrzeugs. Das ist das Prinzip der Abstraktion.

Der Ingenieur beim Automobilhersteller *abstrahiert von* den Details der Reifenherstellung, um sich auf die Konstruktion der Räder eines Fahrzeugs konzentrieren zu können. Der Fahrzeugdesigner wiederum abstrahiert von den technischen Details der Räder und des Motors, um das Gesamtfahrzeug entwerfen zu können (und ist dabei lediglich an der Größe von Motor und Rädern interessiert).

Dies gilt für alle Bestandteile. Während sich jemand mit dem Entwurf des Innenraums beschäftigt, entwickelt ein anderer den Stoff, mit dem die Sitze bezogen werden.

Der entscheidende Punkt ist: Wenn man nur genau genug hinsieht, dann besteht ein Fahrzeug aus so vielen Einzelteilen, dass es für eine einzelne Person unmöglich ist, alle Details aller Teile zu kennen. Wenn das notwendig wäre, würde kein Fahrzeug hergestellt werden können.

Ingenieure können so erfolgreich Fahrzeuge bauen, weil sie Modularisierung und Abstraktion einsetzen. Sie zerteilen ein Fahrzeug in unabhängige Module (Rad, Motor, Getriebe, Sitz, Steuerrad etc.) und lassen verschiedene Personen parallel an diesen verschiedenen Modulen arbeiten. Nachdem ein Modul erstellt wurde, verwenden sie Abstraktion. Sie sehen die Module als einzelne Komponenten an, die zu komplexeren Komponenten zusammengesetzt werden.

Modularisierung und Abstraktion sind somit komplementär. Modularisierung bedeutet, große Dinge (Probleme, Aufgaben) in kleinere Teile zu zerlegen, während Abstraktion die Fähigkeit ist, Details zu ignorieren, um das Gesamtbild erfassen zu können.

KONZEPT:

Modularisierung ist der Prozess der Zerlegung eines Ganzen in wohl definierte Teile, die getrennt erstellt und untersucht werden können und die in wohl definierter Weise interagieren.

3.3 Abstraktion in Software

Die gerade diskutierten Prinzipien von Modularisierung und Abstraktion gelten in gleicher Weise für die Softwareentwicklung. Um den Überblick in komplexen Anwendungen zu behalten, versuchen wir Subkomponenten zu identifizieren und unabhängig zu implementieren. Dann versuchen wir, diese Subkomponenten als einfache Bestandteile zu betrachten, ohne uns mit ihrer internen Komplexität zu beschäftigen.

Bei der objektorientierten Programmierung sind diese Komponenten und Subkomponenten Objekte. Wenn wir ein Fahrzeug durch Software in einer objektorientierten Sprache modellieren wollen, dann gehen wir wie die Kfz-Ingenieure vor. Statt ein Fahrzeug als einen großen, monolithischen Block zu implementieren, würden wir zuerst getrennte Objekte für den Motor, das Getriebe, einen Sitz usw. entwickeln und das Fahrzeug anschließend aus diesen kleineren Teilen zusammensetzen.

Es ist nicht immer leicht, für ein gegebenes Problem herauszufinden, welche Objekte (und damit auch Klassen) in einer Softwarelösung notwendig sind. Darauf werden wir in späteren Abschnitten dieses Buches noch zu sprechen kommen. An dieser Stelle wollen wir mit einem relativ einfachen Beispiel beginnen. Wenden wir uns also wieder unserer Digitaluhr zu.

3.4 Modularisierung im Uhren-Beispiel

Wir wollen das Beispiel der Uhrenanzeige genauer betrachten. Wir wollen die gerade besprochenen Abstraktionskonzepte benutzen, um einen geeigneten Weg zu finden, wie wir die Aufgabe mit einigen Klassen lösen können. Eine mögliche Sichtweise wäre zu sagen, dass die Anzeige aus vier Ziffern besteht (zwei für die Stunden, zwei für die Minuten). Wenn wir nun von dieser grundlegenden Sichtweise etwas abstrahieren, dann können wir die Anzeige auch als zwei getrennte Anzeigen mit je einem Ziffernpaar ansehen (ein Paar für die Stunden, ein Paar für die Minuten). Die eine Anzeige startet bei null, wird jede Stunde um eins erhöht und springt auf null zurück, sobald die 23 überschritten wird. Die andere springt nach dem Überschreiten der 59 wieder auf null zurück. Die Ähnlichkeit im Verhalten dieser beiden Anzeigen könnte uns wiederum dazu führen, von ihren Unterschieden zu abstrahieren. Wir könnten sie als Objekte ansehen, die die Werte von null bis zu einem bestimmten Wert anzeigen können. Der Wert der Anzeige kann inkrementiert werden und sobald der Wert ein Limit überschreitet, springt er wieder auf null zurück. Nun haben wir möglicherweise eine angemessene Ebene der Abstraktion erreicht, die wir durch eine Klasse repräsentieren können: eine Klasse für zweiziffrige Anzeigen.

Abbildung 3.2: Eine zweiziffrige Nummernanzeige

Für die Anzeige unserer Uhr sollten wir also zuerst eine Klasse für solche zweiziffrigen Anzeigen (Abbildung 3.2) entwickeln. Diese Klasse sollte eine sondierende Methode haben, mit der der Wert der Anzeige abgefragt werden kann, und zwei verändernde Methoden, um den Wert setzen und erhöhen zu können. Sobald wir diese Klasse haben, können wir zwei Objekte dieser Klasse mit unterschiedlichen Limits erzeugen und daraus die gesamte Uhrenanzeige zusammensetzen.

3.5 Implementierung der Uhrenanzeige

Nach der Diskussion im vorigen Abschnitt sollten wir zuerst eine zweiziffrige Nummernanzeige entwickeln. Eine solche Anzeige benötigt zwei Werte: das Limit, an dem die Anzeige zurückspringen soll, und den aktuellen Anzeigewert. Wir werden beide in unserer Klasse durch Datenfelder vom Typ `int` repräsentieren (Listing 3.1).

```
public class Nummernanzeige
{
   private int limit;
   private int wert;

   Konstruktor und Methoden hier ausgelassen
}
```

Listing 3.1: Die Klassendefinition einer zweiziffrigen Nummernanzeige

Wir werden uns die weiteren Details dieser Klasse später ansehen. Wir wollen jetzt erst einmal annehmen, dass wir die Klasse `Nummernanzeige` implementieren können, und ein wenig mehr über die gesamte Uhrenanzeige nachdenken. Wir können eine komplette Uhrenanzeige implementieren, indem wir ein Objekt entwerfen, das intern zwei Nummernanzeigen enthält (eine für die Stunden, eine für die Minuten). Jede der Nummernanzeigen wäre ein Datenfeld in der Uhrenanzeige (Listing 3.2). An dieser Stelle machen wir von einem Konzept Gebrauch, das wir vorher schon einmal erwähnt haben: Klassen definieren Typen.

KONZEPT:

Klassen definieren Typen. Ein Klassenname kann als Typname in einer Variablendeklaration verwendet werden. Variablen, die als Typ eine Klasse haben, können Objekte dieser Klasse halten.

```
public class Uhrenanzeige
{
   private Nummernanzeige stunden;
   private Nummernanzeige minuten;

   Konstruktor und Methoden hier ausgelassen
}
```

Listing 3.2: Die Klasse `Uhrenanzeige` enthält zwei Datenfelder vom Typ `Nummernanzeige`

Bei der Diskussion von Datenfeldern in Kapitel 2 haben wir gesagt, dass das Wort `private` von einem Typ und einem Namen für das Datenfeld gefolgt wird. Hier benutzen wir nun die Klasse `Nummernanzeige` als den Typ für die Datenfelder `stunden` und `minuten`. Dies veranschaulicht, dass Klassen als Typen benutzt werden können.

Der Typ eines Datenfelds legt fest, welche Arten von Werten in dem Datenfeld gespeichert werden können. Wenn der Typ eine Klasse ist, kann das Datenfeld Objekte dieser Klasse halten.

3.6 Klassendiagramme und Objektdiagramme

Die im vorigen Abschnitt beschriebene Struktur (ein Objekt der Klasse `Uhrenanzeige` enthält zwei Objekte der Klasse `Nummernanzeige`) kann in einem *Objektdiagramm* visualisiert werden, wie in Abbildung 3.3a dargestellt. Aus diesem Diagramm können Sie ersehen, dass wir mit drei Objekten umgehen. Abbildung 3.3b zeigt das Klassendiagramm für dieselbe Situation.

KONZEPT:

Ein **Klassendiagramm** zeigt die Klassen einer Anwendung und die Beziehungen zwischen diesen Klassen. Es liefert Informationen über den Quelltext. Es präsentiert eine statische Sicht auf ein Programm.

Bemerkenswert ist, dass das Klassendiagramm lediglich zwei Klassen zeigt, während das Objektdiagramm drei Objekte zeigt. Dies liegt daran, dass wir von einer Klasse mehrere Objekte erzeugen können. In diesem Fall erzeugen wir zwei Objekte der Klasse `Nummernanzeige` (Listing 3.3).

```
/**
 * Die Klasse Nummernanzeige repräsentiert Darstellungen von digitalen Werten,
 * die von null bis zu einem vorgegebenen Limit reichen können. Das Limit wird
 * definiert, wenn eine Nummernanzeige erzeugt wird. Die darstellbaren Werte
 * reichen von null bis Limit-1. Wenn beispielsweise eine Nummernanzeige für
 * die Sekunden einer digitalen Uhr verwendet werden soll, würde man ihr Limit
 * auf 60 setzen, damit die dargestellten Werte von 0 bis 59 reichen.
 * Wenn der Wert einer Nummernanzeige erhöht wird, wird bei Erreichen
 * des Limits der Wert automatisch auf null zurückgesetzt.
 *
 * @author Michael Kölling und David J. Barnes
 * @version 2008.03.30
 */
public class Nummernanzeige
{
   private int limit;
   private int wert;
   /**
    * Konstruktor für Exemplare der Klasse Nummernanzeige
    */
   public Nummernanzeige(int anzeigeLimit)
   {
      limit = anzeigeLimit;
      wert = 0;
   }

   /**
    * Liefere den aktuellen Wert als int.
    */
   public int gibWert()
   {
      return wert;
   }

   /**
    * Liefere den Anzeigewert, also den Wert dieser Anzeige als einen
    * String mit zwei Ziffern. Wenn der Wert der Anzeige kleiner als
    * zehn ist, wird die Anzeige mit einer führenden Null eingerückt.
    */

   public String gibAnzeigewert()
   {
      if(wert < 10) {
         return "0" + wert;
      }
```

Listing 3.3: Der Quelltext der Klasse `Nummernanzeige`

```
        else {
            return "" + wert;
        }
    }

    /**
     * Setze den Wert der Anzeige auf den angegebenen 'ersatzwert'. Wenn der
     * angegebene Wert unter null oder über dem Limit liegt, tue nichts.
     */
    public void setzeWert(int ersatzwert)
    {
        if((ersatzwert >= 0) && (ersatzwert < limit)) {
            wert = ersatzwert;
        }
    }

    /**
     * Erhöhe den Wert um eins. Wenn das Limit erreicht ist, setze
     * den Wert wieder auf null.
     */
    public void erhoehen()
    {
        wert = (wert + 1) % limit;
    }
}
```

Listing 3.3: Der Quelltext der Klasse `Nummernanzeige` (Fortsetzung)

Diese beiden Diagramme bieten zwei unterschiedliche Sichtweisen in Bezug auf dieselbe Anwendung. Das Klassendiagramm zeigt die *statische Sicht*. Es illustriert, welche Bestandteile existieren, während das Programm geschrieben wird. Wir haben zwei Klassen und der Pfeil besagt, dass die Klasse `Uhrenanzeige` die Klasse `Nummernanzeige` benutzt (d. h., die Klasse `Nummernanzeige` wird im Quelltext der Klasse `Uhrenanzeige` erwähnt). Wir sagen auch: `Uhrenanzeige` *ist abhängig* von `Nummernanzeige`.

Um die Anwendung zu starten, erzeugen wir ein Objekt der Klasse `Uhrenanzeige`. Wir werden die Uhrenanzeige so implementieren, dass sie automatisch zwei Objekte von `Nummernanzeige` erzeugt. Das Objektdiagramm zeigt somit die Situation zur *Laufzeit* (während das Programm ausgeführt wird). Diese Sicht wird auch die *dynamische Sicht* genannt.

Das Objektdiagramm zeigt ein weiteres wichtiges Detail: Wenn eine Variable ein Objekt enthält, dann ist dieses Objekt nicht direkt in der Variablen selbst abgelegt, sondern die Variable enthält lediglich eine *Referenz auf ein Objekt* (eine *Objektreferenz*). In dem Diagramm ist die Variable durch ein weißes Feld symbolisiert und die Referenz durch einen Pfeil. Das Objekt, das referenziert wird, ist außerhalb des Objekts gespeichert, das die Referenz hält. Die Objekte sind durch die Objektreferenz miteinander verbunden.

KONZEPT:

Ein **Objektdiagramm** zeigt die Objekte und ihre Beziehungen zu einem bestimmten Zeitpunkt während der Ausführung einer Anwendung. Es präsentiert eine dynamische Sicht auf ein Programm.

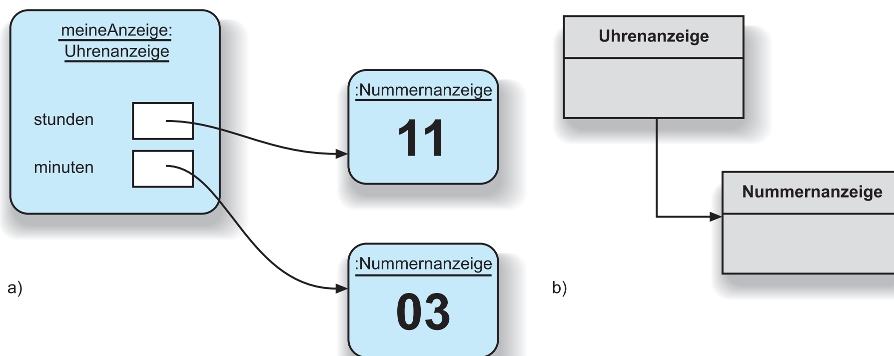

Abbildung 3.3: Objektdiagramm und Klassendiagramm für eine `Uhrenanzeige`

KONZEPT:

Objektreferenz. Variablen von **Objekttypen** speichern Referenzen auf Objekte.

Der Unterschied zwischen diesen beiden Diagrammen und den damit verbundenen Sichtweisen ist sehr wichtig. BlueJ zeigt lediglich die statische Sicht. Sie sehen das Klassendiagramm im Hauptfenster eines Projekts. Um Java-Programme planen und verstehen zu können, müssen Sie in der Lage sein, die Objektdiagramme auf Papier oder in Ihrer Vorstellung zu zeichnen. Wenn wir darüber nachdenken, was ein Programm tut, dann denken wir über die Struktur der beteiligten Objekte nach und wie diese interagieren. Das Verständnis solcher Objektstrukturen ist in der Objektorientierung unverzichtbar.

Übung 3.1 Denken Sie an das Projekt *Laborkurse* zurück, das wir in Kapitel 1 und Kapitel 2 diskutiert haben. Nehmen Sie an, wir erzeugen ein `Laborkurs`-Objekt und drei `Student`-Objekte. Anschließend tragen wir die drei Studenten in den Kurs ein. Versuchen Sie, ein Klassendiagramm und ein Objektdiagramm für diese Situation zu zeichnen. Zeigen Sie die Unterschiede zwischen den Diagrammen auf und erläutern Sie diese.

Übung 3.2 Zu welchen Zeitpunkten kann sich ein Klassendiagramm ändern? Wie wird es geändert?

Übung 3.3 Zu welchen Zeitpunkten kann sich ein Objektdiagramm ändern? Wie wird es geändert?

Übung 3.4 Schreiben Sie eine Definition für ein Datenfeld `tutor`, das eine Referenz auf ein Objekt des Typs `Lehrender` halten kann.

3.7 Primitive Typen und Objekttypen

Java unterscheidet zwei sehr unterschiedliche Typarten: *primitive Typen* und *Objekttypen*. Sämtliche primitiven Typen in Java sind vordefiniert; `int` und `boolean` zählen beispielsweise zu diesen Typen. Eine komplette Liste aller primitiven Typen findet sich in Anhang B. Objekttypen sind die Typen, die durch Klassen definiert werden. Einige Klassen sind in Java vordefiniert (wie `String`); andere schreiben wir selbst. Sowohl primitive Typen als auch Objekttypen können als Typen verwendet werden, aber es gibt Situationen, in denen sie sich unterschiedlich verhalten. Ein Unterschied ist, wie Werte gespeichert werden. Aus unseren Diagrammen konnten wir ersehen, dass primitive Werte direkt in den Variablen gespeichert werden (wir haben die Werte direkt in die Variablenfelder geschrieben, beispielsweise in Abbildung 2.3 in Kapitel 2). Objekte hingegen werden nicht direkt in den Variablen abgelegt, sondern es wird nur eine Referenz auf ein Objekt gespeichert (symbolisiert durch einen Pfeil in unseren Diagrammen, Abbildung 3.3).

KONZEPT:

Die **primitiven Typen** in Java sind die Typen, die keine Objekttypen sind. Die gebräuchlichsten primitiven Typen sind `int`, `boolean`, `char`, `double` und `long`. Primitive Typen haben keine Methoden.

Wir werden später noch weitere Unterschiede zwischen primitiven Typen und Objekttypen kennen lernen.

3.8 Der Quelltext im Projekt *Zeitanzeige*

Bevor wir den Quelltext analysieren, sollten Sie zuerst selbst einen Blick auf das Beispiel werfen.

Übung 3.5 Starten Sie BlueJ, öffnen Sie das Projekt *Zeitanzeige* und experimentieren Sie damit. Erzeugen Sie ein Objekt der Klasse `Uhrenanzeige` unter Verwendung des Konstruktors, der keine Parameter übernimmt, und öffnen Sie den Objektinspektor für dieses Objekt. Rufen Sie, während das Fenster geöffnet ist, die Methoden des Objekts auf. Beobachten Sie das Datenfeld `zeitanzeige` in dem Fenster. Im Projektkommentar (Sie öffnen diesen mit einem Doppelklick auf das Notiz-Symbol in der linken oberen Ecke des Diagramms) können Sie weitere Informationen finden.

3.8.1 Die Klasse `Nummernanzeige`

Wir werden nun eine komplette Implementierung für die genannte Aufgabe analysieren. Das beiliegende Projekt *Zeitanzeige* enthält diese Lösung. Wir werden uns zunächst die Implementierung der Klasse `Nummernanzeige` ansehen. Listing 3.3 zeigt den kompletten Quelltext. Insgesamt ist diese Klasse recht klar strukturiert. Sie hat zwei Datenfelder, die bereits in Abschnitt 3.5 diskutiert wurden, einen Konstruktor und vier Methoden (`gibWert`, `setzeWert`, `gibAnzeigewert` und `erhoehen`).

Der Konstruktor bekommt die Überlaufgrenze als Parameter. Wenn beispielsweise 24 als Limit angegeben wird, dann wird die Anzeige bei Erreichen dieses Werts auf null zurückspringen. Der Bereich, den die Anzeige dann anzeigen könnte, wäre 0 bis 23. Dies ermöglicht uns, die Anzeige sowohl für die Stunden als auch für die Minuten zu verwenden. Für die Stundenanzeige werden wir eine `Nummernanzeige` mit der Anzeigegrenze 24 erzeugen, für die Minutenanzeige eine mit der Grenze 60.

Der Konstruktor speichert den übergebenen Wert in einem Datenfeld und setzt anschließend den Wert der Anzeige auf null.

Als Nächstes sehen wir eine einfache sondierende Methode, die uns den aktuellen Anzeigewert liefert (`gibWert`). Diese erlaubt anderen Objekten, diesen Wert auszulesen.

Die verändernde Methode `setzeWert` ist da schon interessanter. Sie sieht folgendermaßen aus:

```
public void setzeWert(int ersatzwert)
{
   if((ersatzwert >= 0) && (ersatzwert < limit)) {
      wert = ersatzwert;
   }
}
```

Logische Operatoren

Logische Operatoren arbeiten mit booleschen Werten (wahr und falsch bzw. `true` und `false`) und liefern als Ergebnis wieder einen booleschen Wert. Die drei wichtigsten logischen Operatoren sind *und*, *oder* und *nicht*. In Java werden diese folgendermaßen notiert:

```
&&  (und)
||  (oder)
!   (nicht)
```

Der Ausdruck

```
a && b
```

ist dann wahr (liefert `true`), wenn sowohl `a` als auch `b` wahr sind, in allen anderen Fällen ist er falsch (liefert `false`). Der Ausdruck

```
a || b
```

liefert `true`, wenn entweder `a` oder `b` oder beide `true` liefern, und `false`, wenn beide `false` liefern. Der Ausdruck

```
!a
```

liefert `true`, wenn `a` `false` ist, und `false`, wenn `a` `true` ist.

An dieser Stelle kann ein neuer Wert für die Anzeige als Parameter an die Methode übergeben werden. Allerdings müssen wir, bevor wir den neuen Wert zuweisen, seine Gültigkeit überprüfen. Der zugelassene Bereich für den Wert, wie oben diskutiert, geht von null bis eins unter dem Limit. Wir verwenden eine bedingte Anweisung, um vor der Zuwei-

sung zu prüfen, ob der Wert zulässig ist. Das Symbol && ist der logische Und-Operator. Er sorgt dafür, dass die Prüfung in der if-Anweisung nur genau dann true liefert, wenn die Ausdrücke auf beiden Seiten des Operators true liefern. Im folgenden Erläuterungskasten *Logische Operatoren* werden diese Operatoren erläutert. Anhang C enthält eine komplette Tabelle aller logischen Operatoren in Java.

Übung 3.6 Was passiert, wenn Sie die Methode setzeWert mit einem ungültigen Wert aufrufen? Ist das eine gute Lösung? Können Sie sich eine bessere vorstellen?

Übung 3.7 Was würde passieren, wenn Sie den Operator '>=' in der Prüfung durch '>' ersetzen würden, und zwar in folgender Weise:

```
if((ersatzwert > 0) && (ersatzwert < limit))
```

Übung 3.8 Was würde passieren, wenn Sie den Operator '&&' in der Prüfung durch '||' ersetzen würden, und zwar in folgender Weise:

```
if((ersatzwert >= 0) || (ersatzwert < limit))
```

Übung 3.9 Welche der folgenden Ausdrücke liefern true?

```
! (4 < 5)
! false
(2 > 2) || ((4 == 4) && (1 < 0))
(2 > 2) || (4 == 4) && (1 < 0)
(34 != 33) && ! false
```

Nachdem Sie Ihre Antworten auf Papier notiert haben, öffnen Sie die Direkteingabe von BlueJ und testen Sie die Ausdrücke. Vergleichen Sie die Ergebnisse mit Ihren Antworten.

Übung 3.10 Schreiben Sie einen Ausdruck mit zwei booleschen Variablen a und b, der true liefert, wenn entweder a und b beide true sind oder beide false sind.

Übung 3.11 Schreiben Sie einen Ausdruck mit zwei booleschen Variablen a und b, der true liefert, wenn nur genau eine von beiden true ist, und der false liefert, wenn a und b beide false oder beide true sind (dies bezeichnet man auch als *exklusives Oder*).

Übung 3.12 Betrachten Sie den Ausdruck (a && b). Schreiben Sie einen äquivalenten Ausdruck (einen, der in exakt den gleichen Fällen für die gleichen Werte true liefert), ohne den Operator && zu benutzen.

Die nächste Methode, gibAnzeigewert, liefert ebenfalls den Wert der Anzeige zurück, allerdings in einem anderen Format. Dies liegt daran, dass wir die Anzeige als eine Zeichenkette aus zwei Zeichen anzeigen möchten, damit beispielsweise eine Uhrzeit wie 3:05 als 03:05 angezeigt wird und nicht als 3:5. Um dies einfach zu ermöglichen, haben

wir die Methode `gibAnzeigewert` implementiert. Diese Methode liefert den aktuellen Wert als Zeichenkette und fügt außerdem eine führende Null ein, wenn der Wert unter 10 liegt. Der Quelltext sieht folgendermaßen aus:

```
if(value < 10) {
    return "0" + wert;
}
else {
    return "" + wert;
}
```

Beachten Sie, dass die Null (`"0"`) in doppelten Anführungszeichen gesetzt ist. Es handelt sich also hier um die *Zeichenkette* 0, nicht um die *ganze Zahl* 0. Anschließend werden in dem Ausdruck

```
"0" + wert
```

eine Zeichenkette und eine ganze Zahl „addiert“ (denn der Typ von `wert` ist `int`, also eine ganze Zahl). Also ist der Plus-Operator an dieser Stelle wieder ein Verkettungsoperator für Zeichenketten, wie wir bereits in Abschnitt 2.8 gesehen haben. Bevor wir fortfahren, werden wir uns die Verkettung von Zeichenketten etwas genauer ansehen.

3.8.2 Verkettung von Zeichenketten

Der Plus-Operator (+) hat unterschiedliche Bedeutungen, je nach dem Typ seiner Operanden. Wenn beide Operanden Zahlen sind, dann repräsentiert er die erwartete Addition. Also addiert

```
42 + 12
```

die beiden Zahlen und das Ergebnis ist 54. Wenn die Operanden hingegen Zeichenketten sind, dann ist die Bedeutung des Operators eine Verkettung von Zeichenketten, das Ergebnis ist dann eine einzelne Zeichenkette, die aus der Verkettung der beiden Operanden besteht. Das Ergebnis des Ausdrucks

```
"Java" + "mit BlueJ"
```

ist die einzelne Zeichenkette

```
"Javamit BlueJ"
```

Beachten Sie, dass nicht automatisch ein Leerzeichen zwischen den beiden Zeichenketten eingefügt wird. Wenn dort ein Leerzeichen stehen soll, dann muss es explizit in einer der Zeichenketten angegeben werden.

Wenn einer der beiden Operanden eine Zeichenkette ist und der andere nicht, dann wird der andere Operand automatisch in eine Zeichenkette umgewandelt und anschließend eine Verkettung vorgenommen. Deshalb führt

```
"antwort: " + 42
```

zu der Zeichenkette

```
"antwort: 42"
```

Dies funktioniert für alle Typen. Ganz gleich, welchen Typ der hinzuaddierte Operand hat, er wird in eine Zeichenkette umgewandelt und dann angehängt.

Zurück zu den Anweisungen in unserer Methode `gibAnzeigewert`. Wenn `wert` eine 3 enthält, dann wird die Anweisung

```
return "0" + wert;
```

die Zeichenkette `"03"` zurückliefern. In dem Fall aber, in dem der Wert größer ist als 9, haben wir einen kleinen Trick verwendet:

```
return "" + wert;
```

An dieser Stelle haben wir `wert` mit einer leeren Zeichenkette verknüpft. Das Ergebnis ist, dass der Wert in eine Zeichenkette umgewandelt wird und ihm keine weiteren Zeichen vorangestellt werden. Wir benutzen den Plus-Operator hier nur, um einen ganzzahligen Wert vom Typ `int` in einen `String` umzuwandeln.

Übung 3.13 Arbeitet die Methode `gibAnzeigewert` in allen Fällen korrekt? Welche Annahmen werden in ihr gemacht? Was passiert beispielsweise, wenn Sie eine Nummernanzeige mit einem Limit von 800 erzeugen?

Übung 3.14 Gibt es unterschiedliche Ergebnisse, wenn in der Methode `gibAnzeigewert`

```
return wert + "";
```

statt

```
return "" + wert;
```

geschrieben wird?

3.8.3 Der Modulo-Operator

Die letzte Methode in der Klasse `Nummernanzeige` erhöht den Anzeigewert um 1. Sie sorgt außerdem dafür, dass der Wert auf null zurückgesetzt wird, wenn das Limit der Anzeige erreicht ist:

```
public void erhoehen()
{
   wert = (wert + 1) % limit;
}
```

Diese Methode benutzt den *Modulo-Operator* (`%`). Dieser Operator berechnet den Rest einer ganzzahligen Division. Beispielsweise kann das Ergebnis der Division

```
27 / 4
```

durch zwei ganze Zahlen ausgedrückt werden:

```
Ergebnis = 6, Rest = 3
```

Der Modulo-Operator liefert lediglich den Rest einer solchen Division. Das Ergebnis des Ausdrucks `(27 % 4)` wäre demnach `3`.

Übung 3.15 Erklären Sie den Modulo-Operator. Möglicherweise benötigen Sie dazu weitere Quellen (Java-Ressourcen online, andere Java-Bücher etc.), um die Details nachzulesen.

Übung 3.16 Was ist das Ergebnis des Ausdrucks (`8 % 3`)?

Übung 3.17 Testen Sie den Ausdruck (`8 % 3`) in der Direkteingabe. Variieren Sie die Zahlen. Was passiert, wenn Sie den Modulo-Operator mit negativen Zahlen verwenden?

Übung 3.18 Welches sind die möglichen Werte des Ausdrucks (`n % 5`), mit `n` als einer Variablen vom Typ `int`?

Übung 3.19 Welches sind die möglichen Werte des Ausdrucks (`n % m`), mit `n` und `m` als Variablen vom Typ `int`?

Übung 3.20 Erklären Sie ausführlich, wie die Methode `erhoehen` funktioniert.

Übung 3.21 Schreiben Sie die Methode `erhoehen` so um, dass sie statt des Modulo-Operators eine if-Anweisung benutzt. Welche Lösung ist besser?

Übung 3.22 Testen Sie die Klasse `Nummernanzeige` im *Zeitanzeige*-Projekt in BlueJ, indem Sie einige Objekte erzeugen und ihre Methoden aufrufen.

3.8.4 Die Klasse `Uhrenanzeige`

Nachdem wir nun eine Klasse schreiben können, die zweiziffrige Nummern anzeigen kann, sollten wir die Klasse `Uhrenanzeige` näher betrachten – die Klasse, die zwei Nummernanzeigen erzeugt, um die komplette Uhrzeit anzuzeigen. Listing 3.4 zeigt den kompletten Quelltext der Klasse `Uhrenanzeige`.

Wie auch bei der Klasse `Nummernanzeige` werden wir hier kurz alle Datenfelder, Konstruktoren und Methoden durchsprechen.

```
/**
 * Die Klassen Uhrenanzeige implementiert die Anzeige einer Digitaluhr.
 * Die Anzeige zeigt Stunden und Minuten. Der Anzeigebereich reicht von
 * 00:00 (Mitternacht) bis 23:59 (eine Minute vor Mitternacht).
 *
 * Eine Uhrenanzeige sollte minütlich "Taktsignale" (über die Operation
 * "taktsignalGeben") erhalten, damit sie die Anzeige aktualisieren
 * kann. Dies geschieht, wie man es bei einer Uhr erwartet: Die
 * Stunden erhöhen sich, wenn das Minutenlimit einer Stunde erreicht
 * ist.
 *
 * @author Michael Kölling und David J. Barnes
 * @version 2008.03.30
 */
```

Listing 3.4: Der Quelltext der Klasse `Uhrenanzeige`

```
➔ public class Uhrenanzeige
  {
      private Nummernanzeige stunden;
      private Nummernanzeige minuten;
      private String zeitanzeige;    // simuliert die tatsächliche Anzeige

          /**
       * Konstruktor für ein Exemplar von Uhrenanzeige.
       * Mit diesem Konstruktor wird die Anzeige auf 00:00 initialisiert.
       */
      public Uhrenanzeige()
      {
          stunden = new Nummernanzeige(24);
          minuten = new Nummernanzeige(60);
          anzeigeAktualisieren();
      }

      /**
       * Konstruktor für ein Exemplar von Uhrenanzeige.
       * Mit diesem Konstruktor wird die Anzeige auf den Wert
       * initialisiert, der durch 'stunde' und 'minute'
       * definiert ist.
       */
      public Uhrenanzeige(int stunde, int minute)
      {
          stunden = new Nummernanzeige(24);
          minuten = new Nummernanzeige(60);
          setzeUhrzeit(stunde, minute);
      }

      /**
       * Diese Operation sollte einmal pro Minute aufgerufen werden -
       * sie sorgt dafür, dass diese Uhrenanzeige um eine Minute
       * weitergestellt wird.
       */
      public void taktsignalGeben()
      {
          minuten.erhoehen();
          if(minuten.gibWert() == 0) {  // Limit wurde erreicht!
              stunden.erhoehen();
          }
          anzeigeAktualisieren();
      }

      /**
       * Setze die Uhrzeit dieser Anzeige auf die gegebene 'stunde' und
       * 'minute'.
       */
      public void setzeUhrzeit(int stunde, int minute)
      {
```

Listing 3.4: Der Quelltext der Klasse Uhrenanzeige (Fortsetzung) ➔

```
➔       stunden.setzeWert(stunde);
        minuten.setzeWert(minute);
        anzeigeAktualisieren();
    }

    /**
     * Liefere die aktuelle Uhrzeit dieser Uhrenanzeige im Format SS:MM.
     */
    public String gibUhrzeit()
    {
        return zeitanzeige;
    }

    /**
     * Aktualisiere die interne Zeichenkette, die die Zeitanzeige hält.
     */
    private void anzeigeAktualisieren()
    {
        zeitanzeige = stunden.gibAnzeigewert() + ":"
                      + minuten.gibAnzeigewert();
    }
}
```

Listing 3.4: Der Quelltext der Klasse `Uhrenanzeige` (Fortsetzung)

In diesem Projekt benutzen wir das Datenfeld `zeitanzeige`, um die tatsächliche Anzeige der Uhr zu simulieren (wie Sie in Übung 3.2 gesehen haben). Wenn die Software in einer echten Uhr laufen würde, dann würden wir die Ausgabe entsprechend auf ihrer Anzeige erzeugen. Diese Zeichenkette dient uns also als Softwaresimulation einer echten Uhrenanzeige.

Um dies zu erreichen, benutzen wir ein Datenfeld vom Typ `String` und eine Methode:

```
public class Uhrenanzeige
{
    private String zeitanzeige;

    andere Datenfelder und Methoden hier ausgelassen

    /**
     * Aktualisiere die interne Zeichenkette, die die Zeitanzeige hält.
     */
    private void anzeigeAktualisieren()
    {
        Implementierung der Methode hier ausgelassen
    }
}
```

Jedes Mal, wenn die Anzeige der Uhr sich ändern soll, rufen wir die interne Methode `anzeigeAktualisieren` auf. In unserer Simulation ändert die Methode die Zeichenkette für die Zeitanzeige (die Realisierung sehen wir uns weiter unten an). In einer echten Uhr würde diese Methode ebenfalls existieren – dort würde sie die echte Anzeige verändern.

Außer `zeitanzeige` hat die Klasse `Uhrenanzeige` zwei weitere Datenfelder: `stunden` und `minuten`. Jedes dieser Datenfelder kann ein Objekt vom Typ `Nummernanzeige` halten. Der logische Wert der Uhrenanzeige (die aktuelle Zeit) ist in diesen Objekten gespeichert. Abbildung 3.4 zeigt ein Objektdiagramm dieser Anwendung mit der aktuellen Zeit 15:23.

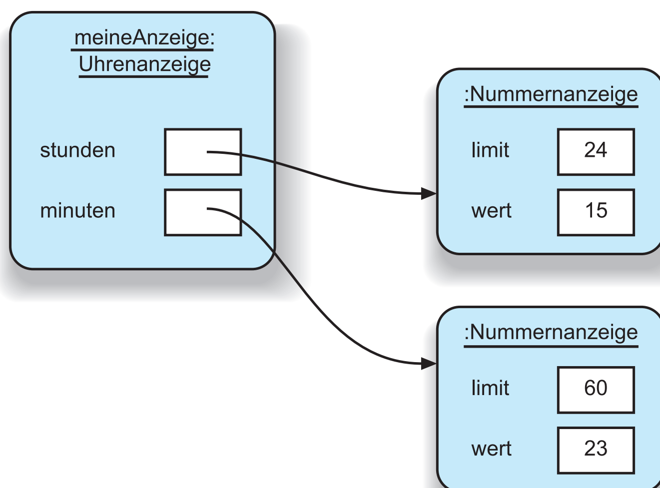

Abbildung 3.4: Ein Objektdiagramm der Uhrenanzeige

3.9 Objekte erzeugen Objekte

Die erste Frage, die wir uns stellen müssen, ist: Woher kommen die drei Objekte in diesem Diagramm? Wenn wir eine Uhrenanzeige haben wollen, dann werden wir ein Objekt der Klasse `Uhrenanzeige` erzeugen. Wir werden dann einfach annehmen, dass unsere Uhrenanzeige auch Stunden und Minuten hat. Wir erwarten also, dass durch die Erzeugung der Uhrenanzeige auch zwei Nummernanzeigen für die Stunden und Minuten erzeugt wurden.

KONZEPT:

Objekterzeugung. Objekte können andere **Objekte** mit dem *new*-Operator **erzeugen**.

Als Entwickler der Klasse `Uhrenanzeige` müssen wir dafür sorgen, dass das tatsächlich passiert. Wir schreiben Anweisungen in den Konstruktor der Klasse `Uhrenanzeige`, die zwei `Nummernanzeigen` erzeugen und abspeichern. Da der Konstruktor automatisch ausgeführt wird, wenn ein Objekt der Klasse `Uhrenanzeige` erzeugt wird, werden die Objekte von `Nummernanzeige` zur gleichen Zeit mit erzeugt. Hier sind die Anweisungen des Konstruktors, in denen das geschieht:

```
public class Uhrenanzeige
{
   private Nummernanzeige stunden;
   private Nummernanzeige minuten;

   restliche Datenfelder hier ausgelassen

   public Uhrenanzeige()
   {
      stunden = new Nummernanzeige(24);
      minuten = new Nummernanzeige(60);
      anzeigeAktualisieren();
   }

   Methoden hier ausgelassen
}
```

In jeder der ersten beiden Zeilen wird ein Objekt von `Nummernanzeige` erzeugt und einer Variablen zugewiesen. Die Syntax einer Anweisung zum Erzeugen eines Objekts ist

```
new Klassenname ( Parameterliste )
```

Die `new`-Anweisung macht zwei Dinge:

1. Sie erzeugt ein neues Objekt der angegebenen Klasse (hier: `Nummernanzeige`).

2. Sie führt den Konstruktor dieser Klasse aus.

Wenn der Konstruktor der Klasse Parameter definiert, dann müssen aktuelle Parameter in der `new`-Anweisung angegeben werden. Der Konstruktor der Klasse `Nummernanzeige` beispielsweise ist mit einem `int`-Parameter definiert:

formaler Parameter

```
public Nummernanzeige(int anzeigeGrenze)
```

Deshalb muss die `new`-Anweisung, die diesen Konstruktor aufruft, einen aktuellen Parameter vom Typ `int` übergeben, damit der Aufruf zum Kopf des Konstruktors passt:

aktueller Parameter

```
new Nummernanzeige (24);
```

Dies ist dasselbe Prinzip, das wir in Abschnitt 2.4 für Methoden besprochen haben. Mit diesem Konstruktor haben wir erreicht, was wir erreichen wollten: Wenn jemand ein Objekt der Klasse `Uhrenanzeige` erzeugt, dann wird der Konstruktor der Klasse `Uhrenanzeige` automatisch ausgeführt und zwei Objekte der Klasse `Nummernanzeige` werden erzeugt. Dann ist die Uhrenanzeige in einem vernünftigen Ausgangszustand.

Übung 3.23 Erzeugen Sie ein Objekt der Klasse `Uhrenanzeige`, indem Sie den folgenden Konstruktor benutzen:

```
new Uhrenanzeige()
```

Rufen Sie an diesem Objekt die Methode `eoc` auf, um die Anfangszeit herauszufinden, auf die die Anzeige gesetzt wurde. Können Sie erklären, warum sie gerade mit dieser Zeit startet?

Übung 3.24 Wie oft müssen Sie die Methode `taktsignalGeben` an einem neu erzeugten `Uhrenanzeige`-Objekt aufrufen, damit die Anzeige den Wert 01:00 erreicht? Welchen anderen Weg könnte es geben, die Anzeige auf diesen Wert zu bringen?

Übung 3.25 Erzeugen Sie in der Direkteingabe ein Objekt der Klasse `Nummernanzeige` mit dem Limit 80:

```
Nummernanzeige na = new Nummernanzeige(80);
```

Rufen Sie anschließend in der Direkteingabe die Methoden `gibWert()`, `setzeWert(int wert)` und `erhoehen()` des Objekts auf (z.B. indem Sie `na.gibWert()` eintippen). Beachten Sie, dass Anweisungen (verändernde Methoden) im Gegensatz zu Ausdrücken (sondierenden Methoden) ein Semikolon am Ende erfordern.

Übung 3.26 Schreiben Sie die Signatur eines Konstruktors, der zu folgender Objekterzeugung passt:

```
new Editor("readme.txt",-1)
```

Übung 3.27 Schreiben Sie Java-Anweisungen, die eine Variable `fenster` vom Typ `Rechteck` definieren, anschließend ein `Rechteck`-Objekt erzeugen und es dann dieser Variablen zuweisen. Der Konstruktor der Klasse `Rechteck` hat zwei `int`-Parameter.

3.10 Mehrere Konstruktoren

Sie haben möglicherweise beim Erzeugen des `Uhrenanzeige`-Objekts bemerkt, dass das Kontextmenü Ihnen zwei Möglichkeiten dazu angeboten hat:

```
new Uhrenanzeige()
new Uhrenanzeige(stunde, minute)
```

Dies liegt daran, dass die Klasse `Uhrenanzeige` zwei Konstruktoren hat. Diese bieten unterschiedliche Wege zur Initialisierung von Uhrenanzeige-Objekten an. Wenn der parameterlose Konstruktor benutzt wird, dann ist die Anfangszeit der Anzeige 00:00. Wenn Sie aber eine andere Startzeit für die Anzeige wählen möchten, dann können Sie diese mit dem zweiten Konstruktor setzen. Es ist durchaus üblich, dass eine Klasse mehrere Versionen von Konstruktoren oder Methoden anbietet, die unterschiedliche Wege zum Erfüllen einer Aufgabe über unterscheidbare Parametersätze anbieten. Dies wird allgemein als *Überladen* eines Konstruktors oder einer Methode bezeichnet.

KONZEPT:

Überladen. Eine Klasse kann **mehr als einen Konstruktor** oder mehr als eine Methode mit dem gleichen Namen enthalten, solange jede von ihnen einen unterscheidbaren Satz von Parametertypen definiert.

Übung 3.28 Sehen Sie sich den zweiten Konstruktor im Quelltext der Klasse `Uhrenanzeige` an. Erklären Sie, was dieser tut und wie er es tut.

Übung 3.29 Welche Gemeinsamkeiten haben die beiden Konstruktoren, worin unterscheiden sie sich? Warum steht in dem zweiten Konstruktor beispielsweise kein Aufruf der Methode `anzeigeAktualisieren`?

3.11 Methodenaufrufe

3.11.1 Interne Methodenaufrufe

KONZEPT:

Methoden können andere Methoden der eigenen Klasse als Teil ihrer Implementierung aufrufen. Dies wird als **interner Methodenaufruf** bezeichnet.

Die letzte Zeile des ersten Konstruktors in `Uhrenanzeige` besteht aus der Anweisung

```
anzeigeAktualisieren();
```

Diese Anweisung ist ein *Methodenaufruf*. Wie wir bereits oben gesehen haben, hat die Klasse `Uhrenanzeige` eine Methode mit der folgenden Signatur:

```
private void anzeigeAktualisieren()
```

Der obige Methodenaufruf bewirkt die Ausführung dieser Methode. Da diese Methode in derselben Klasse steht wie der Aufruf der Methode, nennen wir ihn auch einen *internen Methodenaufruf*. Interne Methodenaufrufe haben folgende Syntax:

```
Methodenname ( Parameterliste )
```

In unserem Beispiel hat die Methode keine Parameter, deshalb ist die Parameterliste leer. Dies wird durch das leere Klammerpaar deutlich.

Wenn ein Methodenaufruf erfolgt, dann wird die passende Methode ausgeführt und anschließend kehrt die Ausführung zur Aufrufstelle zurück und wird bei der nächsten Anweisung nach dem Aufruf fortgesetzt. Damit eine Methode zu einem Methodenaufruf passt, müssen sowohl der Name als auch die Parameterliste der Methode passen. In diesem Beispiel sind beide Parameterlisten leer, daher passen sie. Dass Methodenname und Parameterliste passen, ist sehr wichtig, denn es könnte mehr als eine Methode mit dem gleichen Namen in der Klasse geben – wenn diese überladen ist.

In unserem Beispiel ist der Zweck der Methode, die Zeichenkette für die Anzeige zu aktualisieren. Nachdem die beiden Nummernanzeigen erzeugt wurden, wird diese Zeichenkette auf den Wert gesetzt, den die beiden Anzeigeobjekte repräsentieren. Die Implementierung der Methode `anzeigeAktualisieren` werden wir später diskutieren.

3.11.2 Externe Methodenaufrufe

Lassen Sie uns nun die nächste Methode untersuchen: `taktsignalGeben`. Die Definition lautet:

```
public void taktsignalGeben()
{
   minuten.erhoehen();
   if(minuten.gibWert() == 0) {  // Limit wurde erreicht!
      stunden.erhoehen();
   }
   anzeigeAktualisieren();
}
```

Wenn diese Anzeige Teil einer echten Uhr wäre, dann würde diese Methode alle 60 Sekunden vom elektronischen Taktgeber der Uhr aufgerufen. Wir rufen sie hier zu Testzwecken einfach von Hand auf.

Wenn die Methode `taktsignalGeben` aufgerufen wird, führt sie zuerst die Anweisung

```
minuten.erhoehen();
```

aus. Diese Anweisung ruft die Methode `erhoehen` des `minuten`-Objekts auf. Somit ruft, wenn eine der Methoden eines `Uhrenanzeige`-Objekts aufgerufen wird, diese wiederum eine Methode eines anderen Objekts auf, um einen Teil der Arbeit von diesem erledigen zu lassen. Der Aufruf einer Methode eines anderen Objekts wird als *externer Methodenaufruf* bezeichnet. Die Syntax eines externen Methodenaufrufs ist:

```
Objekt . Methodenname ( Parameterliste )
```

Diese Syntax bezeichnet man auch als *Punkt-Notation*. Sie besteht aus einem Objektnamen, einem Punkt, dem Methodennamen und den Parametern für den Aufruf. Es ist sehr wichtig, zu erkennen, dass an dieser Stelle der Name eines *Objekts* verwendet wird und nicht der Name einer Klasse. Wir benutzen hier den Namen `minuten` statt `Nummernanzeige`.

KONZEPT:

Methoden können Methoden von anderen Objekten über die Punkt-Notation aufrufen. Dies wird als **externer Methodenaufruf** bezeichnet.

Die Methode `taktsignalGeben` benutzt dann eine if-Anweisung, um zu überprüfen, ob die Stunden ebenfalls erhöht werden müssen. Als Teil der Prüfung in der if-Anweisung wird eine weitere Methode des `minuten`-Objekts aufgerufen: `gibWert`. Diese Methode liefert den aktuellen Wert der Minuten. Wenn dieser Wert null ist, dann wissen wir, dass gerade das Limit überschritten wurde und dass wir deshalb auch die Stunden erhöhen sollten. Das ist genau das, was in diesem Quelltextabschnitt passiert.

Wenn der Wert der Minuten nicht null ist, dann sind wir fertig. In diesem Fall brauchen wir die Stunden nicht zu verändern. Deshalb hat die if-Anweisung keinen `else`-Teil.

Wir sollten nun auch in der Lage sein, die anderen drei Methoden der Klasse `Uhrenanzeige` zu verstehen (siehe Listing 3.4). Die Methode `setzeUhrzeit` fordert zwei Parameter – die Stunden und die Minuten – und setzt die Anzeige auf den entsprechenden Wert. Wenn wir uns den Rumpf der Methode ansehen, dann sehen wir, dass sie dies durch Aufrufe der Methode `setzeWert` an beiden Nummernanzeigen vornimmt. Anschließend

ruft sie `anzeigeAktualisieren` auf, um die Zeichenkette für die Anzeige entsprechend zu aktualisieren, genau wie der Konstruktor dies tut.

Die Methode `gibUhrzeit` ist trivial – sie liefert lediglich den aktuellen Wert der Zeichenkette für die Anzeige. Da diese immer auf dem aktuellen Stand gehalten wird, brauchen wir hier nichts weiter zu tun.

Die Methode `anzeigeAktualisieren` schließlich ist dafür zuständig, dass die Zeichenkette für die Anzeige korrekt den Wert anzeigt, der durch die beiden Nummernanzeigen repräsentiert wird. Sie wird jedes Mal aufgerufen, wenn sich die anzuzeigende Zeit ändert. Sie erfüllt ihre Aufgabe, indem sie an beiden `Nummernanzeige`-Objekten die Methode `gibWert` aufruft. Diese Methodenaufrufe liefern die Werte der Nummernanzeigen. Zuletzt verknüpft sie diese beiden Werte über den Operator zur Verkettung von Zeichenketten mit einem Doppelpunkt in der Mitte zu einer einzelnen Zeichenkette.

Übung 3.30 Gegeben sei eine Variable

```
Drucker d1;
```

die aktuell eine Referenz auf ein `Drucker`-Objekt hält, sowie zwei Methoden in der Klasse `Drucker` mit den Signaturen

```
public void drucke(String dateiname, boolean doppelseitig)
public int gibStatus(int wartezeit)
```

Schreiben Sie zwei mögliche Aufrufe zu jeder dieser beiden Methoden.

3.11.3 Zusammenfassung der Uhrenanzeige

Wir sollten an dieser Stelle noch einmal kurz Revue passieren lassen, wie in diesem Beispiel Abstraktion benutzt wird, um eine Aufgabe in kleinere Teilaufgaben zu zerlegen. Ein Blick auf den Quelltext der Klasse `Uhrenanzeige` zeigt, dass wir dort lediglich `Nummernanzeige`-Objekte erzeugen, ohne dass wir daran interessiert wären, wie diese Objekte intern funktionieren. Wir rufen ihre Methoden auf (`erhoehen`, `gibWert`), damit sie einen Teil der Arbeit für uns erledigen. Auf dieser Ebene nehmen wir einfach an, dass `erhoehen` korrekt den Wert der Anzeige erhöht, ohne dass uns interessiert, wie sie dies tut.

In echten Softwareentwicklungsprojekten werden solche Klassen häufig von zwei unterschiedlichen Personen geschrieben. Es ist Ihnen vermutlich inzwischen klar geworden, dass diese beiden Personen sich lediglich darüber einig sein müssen, welche Methodensignaturen eine Klasse haben sollte und was diese Methoden tun. Dann kann die eine Person sich auf die Implementierung dieser Methoden konzentrieren, während die andere sie einfach nur benutzt.

Die Menge der Methoden, die ein Objekt anderen Objekten zugänglich macht, wird als seine *Schnittstelle* bezeichnet. Wir werden Schnittstellen in diesem Buch noch sehr ausführlich diskutieren.

Übung 3.31 *Zusatzaufgabe* Ändern Sie die 24-Stunden-Anzeige in eine 12-Stunden-Anzeige. Vorsicht: Dies ist nicht so einfach, wie es auf den ersten Blick scheint. Bei einer 12-Stunden-Anzeige werden die Minuten nach Mitternacht und nach dem Mittag nicht als 00:30, sondern als 12:30 angezeigt. Somit zeigt die Minutenanzeige Werte von 0 bis 59, während die Stundenanzeige Werte von 1 bis 12 anzeigt!

Übung 3.32 Es gibt (mindestens) zwei Wege, wie Sie eine 12-Stunden-Anzeige realisieren können. Eine Möglichkeit ist, die Stundenwerte nur von 1 bis 12 zu speichern. Eine andere lässt die Anzeige intern nach wie vor als 24-Stunden-Anzeige arbeiten, passt aber die Zeichenkette zur Ausgabe auf `4:23` oder `4.23pm` an, wenn der interne Wert `16:23` ist. Welche Version ist einfacher? Welche ist besser? Warum?

3.12 Ein weiteres Beispiel für Objektinteraktion

Wir werden nun dieselben Konzepte mit einem anderen Beispiel und mit anderen Werkzeugen betrachten. Wir wollen noch immer ein gutes Verständnis davon bekommen, wie Objekte andere Objekte erzeugen und wie diese Objekte gegenseitig ihre Methoden aufrufen. Im ersten Teil dieses Kapitels haben wir dazu die grundlegendste Technik zur Analyse eines Programms benutzt: das Lesen von Quelltext. Die Fähigkeit, einen Quelltext zu lesen und zu verstehen, ist eine der wichtigsten Eigenschaften für einen Softwareentwickler, und wir werden diese Fähigkeit in jedem Projekt brauchen, in dem wir mitarbeiten. Manchmal kann es aber nützlich sein, mit Hilfe von zusätzlichen Werkzeugen ein besseres Verständnis davon zu bekommen, was bei einer Programmausführung passiert. Ein solches Werkzeug, das wir jetzt näher betrachten wollen, ist ein *Debugger*.

KONZEPT:

Ein **Debugger** ist ein Softwarewerkzeug, mit dessen Hilfe die Ausführung eines Programms untersucht werden kann. Es kann benutzt werden, um Fehler (Bugs) zu finden.

Ein Debugger ist ein Werkzeug, mit dem ein Entwickler ein Programm Schritt für Schritt ausführen lassen kann. Er bietet üblicherweise Funktionen zum Stoppen und Starten eines Programms an ausgewählten Stellen im Quelltext und zum Betrachten der Werte von Variablen.

Der Begriff „Debugger“

Fehler in Computerprogrammen werden im Jargon häufig auch „Bugs“ (englisch für Käfer) genannt. Deshalb werden Programme, die bei der Beseitigung von Fehlern helfen, „De-bugger“ genannt.

Es ist nicht vollständig geklärt, worauf diese Verwendung des Begriffs zurückgeht. Es gibt den berühmten Fall des „ersten Computer-Bugs“ – ein echter Käfer (genauer eine Motte), der im Jahr 1945 im Innern des Mark II-Computers durch Grace Murray Hopper, einer frühen Computerpionierin, gefunden wurde. Es existiert noch immer ein Logbuch im „National Museum of American History of the Smithsonian Institute“, in dem bei einem Eintrag die Motte eingeklebt ist, mit der Bemerkung „erster

tatsächlicher Fall eines gefundenen Bugs“. Diese Formulierung lässt allerdings vermuten, dass der Begriff „Bug“ schon benutzt wurde, bevor dieser echte „Bug“ Probleme im Mark II bereitet hat.

Für weitere Details suchen Sie im Web nach „first computer bug“ – Sie werden sogar Bilder der Motte finden!

Debugger unterscheiden sich stark in ihrer Komplexität. Solche für die professionelle Entwicklung bieten eine umfangreiche Funktionalität, um sehr detailliert die unterschiedlichen Aspekte einer Anwendung ausleuchten zu können. Der Debugger, der in BlueJ integriert ist, ist sehr viel simpler. Wir können damit die Ausführung eines Programms anhalten, zeilenweise den Quelltext ausführen lassen und die Werte von Variablen untersuchen. Trotz dieser abgespeckten Funktionalität ist er leistungsfähig genug, um uns nützliche Informationen zu liefern.

Bevor wir den Debugger ausprobieren, werfen wir zuerst einen Blick auf das Beispiel, mit dem wir ihn demonstrieren wollen: eine Simulation eines E-Mail-Systems.

3.12.1 Das Beispiel eines Mail-Systems

Wir beginnen, indem wir uns die Funktionalität des Projekts *Mail-System* ansehen. An dieser Stelle müssen wir noch keinen Quelltext lesen, wir starten das System einfach und untersuchen sein Verhalten.

Übung 3.33 Öffnen Sie das Projekt *Mail-System*, das Teil des Zusatzmaterials zu diesem Buch ist. In diesem Projekt soll simuliert werden, dass Benutzer sich gegenseitig E-Mails in Form von Nachrichten schicken. Dazu verwendet ein Benutzer einen so genannten Mail-Client (ein Programm zum Lesen und Schreiben von E-Mails), der Nachrichten an einen Server schickt, der sie wiederum an den Mail-Client eines anderen Benutzers weiterleitet. Erzeugen Sie zuerst ein Objekt der Klasse `MailServer`. Erzeugen Sie dann ein Objekt der Klasse `MailClient` für einen der Benutzer. Bei der Erzeugung eines Mail-Clients müssen Sie eine `MailServer`-Instanz als Parameter angeben. Verwenden Sie dazu diejenige, die Sie gerade erzeugt haben. Sie müssen außerdem einen Benutzernamen für den Mail-Client angeben. Erzeugen Sie anschließend einen weiteren Mail-Client mit einem anderen Benutzernamen.

Experimentieren Sie mit den `MailClient`-Objekten herum. Sie können benutzt werden, um Nachrichten von einem Mail-Client zum anderen zu schicken (über die Methode `sendeNachricht`) und um Nachrichten zu empfangen (über die Methoden `gibNaechsteNachricht` und `naechsteNachrichtAusgeben`).

Bei der Untersuchung des Projekts *Mail-System* stellen wir folgende Dinge fest:

- Es enthält drei Klassen: `MailServer`, `MailClient` und `Nachricht`.
- Ein `MailServer`-Objekt muss erzeugt werden, das von allen Mail-Clients benutzt wird. Es kümmert sich um den Austausch der Nachrichten.

- Es können mehrere Mail-Clients erzeugt werden. Jeder Mail-Client ist mit einem Benutzernamen verknüpft.
- Nachrichten können von einem Mail-Client zu einem anderen Mail-Client über eine Methode in der Klasse `MailClient` verschickt werden.
- Nachrichten können von einem Mail-Client durch eine Methode in `MailClient` empfangen werden, die Nachrichten einzeln vom Server abholt.
- Der Benutzer erzeugt nicht explizit Objekte der Klasse `Nachricht`. Diese Klasse wird intern in den beiden anderen Klassen verwendet, um Nachrichtentexte zu erzeugen, zu speichern und auszutauschen.

Übung 3.34 Zeichnen Sie ein Objektdiagramm von der Situation unmittelbar nach der Erzeugung eines Mail-Servers und dreier Mail-Clients. Objektdiagramme wurden in Abschnitt 3.6 diskutiert.

Die drei Klassen sind unterschiedlich komplex. Die Klasse `Nachricht` ist recht trivial. Wir werden nur ein kleines Detail diskutieren und die weitere Erkundung dem Leser überlassen. Die Klasse `MailServer` ist an diesem Punkt der Diskussion noch sehr komplex – sie benutzt Konzepte, die wir erst sehr viel später in diesem Buch diskutieren werden. Wir werden sie deshalb hier nicht näher betrachten. Stattdessen verlassen wir uns darauf, dass sie ihre Aufgabe erfüllt – ein weiteres Beispiel dafür, dass Abstraktion uns ermöglicht, unnötige Details zu ignorieren.

Die Klasse `MailClient` ist am interessantesten, wir werden sie deshalb ausführlich diskutieren.

3.12.2 Das Schlüsselwort this

Der einzige Abschnitt der Klasse `Nachricht`, den wir hier betrachten wollen, ist der Konstruktor. Er benutzt ein Java-Konstrukt, das wir bisher noch nicht kennen gelernt haben. Der Quelltext ist in Listing 3.5 zu sehen.

```
/**
 * Eine Klasse, die einfache Nachrichten modelliert. Eine Nachricht hat
 * einen Absender und einen Empfänger und enthält Text.
 * @author David J. Barnes and Michael Kolling
 * @version 2008.03.30
 */
public class Nachricht
{
   // Der Absender der Nachricht
   private String absender;
   // Der gewünschte Empfänger der Nachricht
   private String empfaenger;
   // Der Text der Nachricht
```

Listing 3.5: Datenfelder und Konstruktor der Klasse `Nachricht`

```
➔   private String text;

    /**
     * Erzeuge eine Nachricht vom gegebenen 'absender' an den gegebenen
     * 'empfaenger' mit dem gegebenen 'text'.
     * @param absender der Absender dieser Nachricht
     * @param empfaenger der gewünschte Empfänger dieser Nachricht.
     * @param text der Text der Nachricht.
     */
    public Nachricht(String absender, String empfaenger, String text)
    {
        this.absender = absender;
        this.empfaenger = empfaenger;
        this.text = text;
    }

    Methoden hier ausgelassen

}
```

Listing 3.5: Datenfelder und Konstruktor der Klasse `Nachricht` (Fortsetzung)

Das neue Java-Konstrukt in diesem Quelltextabschnitt ist die Verwendung des Schlüsselworts `this`:

```
this.absender = absender;
```

Die gesamte Zeile ist eine Zuweisung. Sie weist den Wert auf der rechten Seite (`absender`) der Variablen auf der linken Seite (`this.absender`) zu.

Dieses Konstrukt wird benutzt, weil in dieser Situation ein Name überladen ist – derselbe Name wird für zwei verschiedene Dinge verwendet. Die Klasse enthält drei Datenfelder, `absender`, `empfaenger` und `nachricht`. Der Konstruktor hat drei Parameter, die ebenfalls `absender`, `empfaenger` und `nachricht` heißen!

Wie viele Variablen existieren denn dann, während der Konstruktor ausgeführt wird? Die Antwort lautet sechs – drei Datenfelder und drei Parameter. An dieser Stelle ist sehr wichtig, zu verstehen, dass die Datenfelder und Parameter unterschiedliche Variablen sind, die unabhängig voneinander existieren, auch wenn sie die gleichen Namen tragen. Ein Parameter und ein Datenfeld, die den gleichen Namen tragen, sind in Java kein wirkliches Problem.

Wir haben allerdings das Problem, dass wir die sechs Variablen benennen und dabei die beiden Sätze an Variablen unterscheiden wollen. Wenn wir einfach nur den Variablennamen `absender` im Konstruktor verwenden (beispielsweise in einer Anweisung wie `System.out.println(absender)`), welche Variable wird dann benutzt – der Parameter oder das Datenfeld?

Die Spezifikation der Sprache Java gibt hier die Antwort. Sie spezifiziert, dass immer die Definition des nächsten umschließenden Blocks verwendet wird. Da der Parameter `absender` im Konstruktor definiert ist und das Datenfeld `absender` in der Klasse, wird der Parameter verwendet. Seine Definition ist „näher“ an der Anweisung, die ihn benutzt.

Alles, was wir nun brauchen, ist ein Mechanismus, mit dem wir auf ein Datenfeld zugreifen können, wenn eine Variable mit gleichem Namen in größerer Nähe definiert ist. Genau dafür wird das Schlüsselwort `this` verwendet. Der Ausdruck `this` bezieht sich auf das aktuelle Objekt. Wenn wir `this.absender` schreiben, beziehen wir uns auf das Datenfeld `absender` im aktuellen Objekt. Dieses Konstrukt gibt uns somit die Möglichkeit, statt auf einen Parameter auf ein Datenfeld mit gleichem Namen zuzugreifen. Wir können die Zuweisung nun erneut betrachten:

```
this.absender = absender;
```

Diese Anweisung hat, wie wir nun wissen, den folgenden Effekt:

Datenfeld namens `absender` = *Parameter namens* `absender;`

Mit anderen Worten: Sie weist den Wert des Parameters an das Datenfeld mit dem gleichen Namen zu. Und das ist genau das, was wir für die Initialisierung des Objekts tun wollen.

Es bleibt eine letzte Frage: Warum tun wir das alles? Das ganze Problem könnte leicht umgangen werden, wenn wir die Parameter und Datenfelder unterschiedlich benennen würden. Der Grund ist die Lesbarkeit des Quelltextes.

Manchmal gibt es einen Namen, der die Bedeutung einer Variablen perfekt beschreibt. Er passt so gut, dass wir uns keinen weiteren Namen für sie ausdenken wollen. Wir wollen ihn für den Parameter verwenden, weil er dem Aufrufer einen Hinweis darauf gibt, was übergeben werden soll. Und wir wollen ihn für das Datenfeld verwenden, weil er dem Entwickler der Klasse deutlich macht, zu welchem Zweck es definiert ist. Wenn ein Name die Benutzung perfekt beschreibt, dann sollte man ihn für beide verwenden und den geringen Mehraufwand für das Einfügen des Schlüsselworts `this` in die Zuweisung in Kauf nehmen.

3.13 Die Benutzung eines Debuggers

Die interessanteste Klasse in unserem Mail-System ist `MailClient`. Wir werden diese nun mit Hilfe des Debuggers näher untersuchen. Sie hat drei Methoden: `gibNaechsteNachricht`, `naechsteNachrichtAusgeben` und `sendeNachricht`. Wir werden zuerst die Methode `naechsteNachrichtAusgeben` betrachten.

Bevor wir den Debugger starten, werden wir ein Szenario erstellen, das wir dann näher untersuchen können (Übung 3.35).

Übung 3.35 Erstellen Sie ein Szenario zur Untersuchung: Erzeugen Sie einen Mail-Server, dann zwei Mail-Clients für die Benutzer „Sophie“ und „Juan“ (Sie sollten auch die Instanzen so benennen, damit Sie sie auf der Objektleiste besser unterscheiden können). Benutzen Sie dann Sophies Methode `sendeNachricht`, um eine Nachricht an Juan zu schicken. Rufen Sie die Nachricht noch nicht ab.

Nach dem Aufbau in Übung 3.35 haben wir eine Situation, in der eine Nachricht für Juan auf dem Server gespeichert ist, die abgerufen werden soll. Wir haben bereits gesehen, dass die Methode `naechsteNachrichtAusgeben` diese Nachricht abruft und sie auf der Konsole ausgibt. Nun wollen wir genau wissen, wie das abläuft.

3.13.1 Haltepunkte setzen

Um unsere Untersuchung zu starten, setzen wir zuerst einen Haltepunkt (Übung 3.36). Ein Haltepunkt ist ein Hinweis, der einer Zeile im Quelltext angeheftet wird. Wenn bei der Ausführung einer Methode diese Stelle erreicht wird, dann wird die Ausführung angehalten. Ein Haltepunkt ist im BlueJ-Editor durch ein kleines Stoppzeichen symbolisiert (Abbildung 3.5).

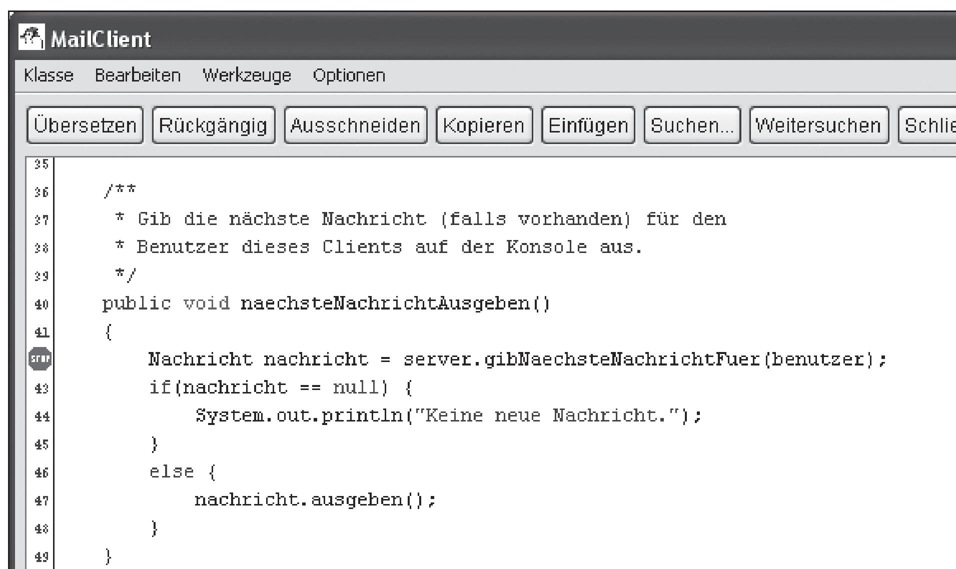

Abbildung 3.5: Ein Haltepunkt im BlueJ-Editor

Sie können einen Haltepunkt setzen, indem Sie den BlueJ-Editor öffnen, eine geeignete Zeile selektieren (in unserem Beispiel die erste Zeile der Methode `naechsteNachrichtAusgeben`) und aus dem Menü *Werkzeuge* den Eintrag *Haltepunkt setzen/entfernen* auswählen. Alternativ können Sie auch einfach in den Bereich mit den Haltepunkten klicken, um dort einen Haltepunkt zu setzen oder zu entfernen. Beachten Sie, dass die Klasse hierzu übersetzt sein muss und dass ein Übersetzen alle gesetzten Haltepunkte entfernt.

Übung 3.36 Öffnen Sie den Editor für die Klasse `MailClient` und setzen Sie einen Haltepunkt in die erste Zeile der Methode `naechsteNachrichtAusgeben`, wie in Abbildung 3.5 gezeigt.

Nachdem Sie den Haltepunkt gesetzt haben, rufen Sie die Methode `naechsteNachrichtAusgeben` an Juans Mail-Client auf. Das Editorfenster für die Klasse `MailClient` und ein Debugger-Fenster werden in den Vordergrund geholt (Abbildung 3.6).

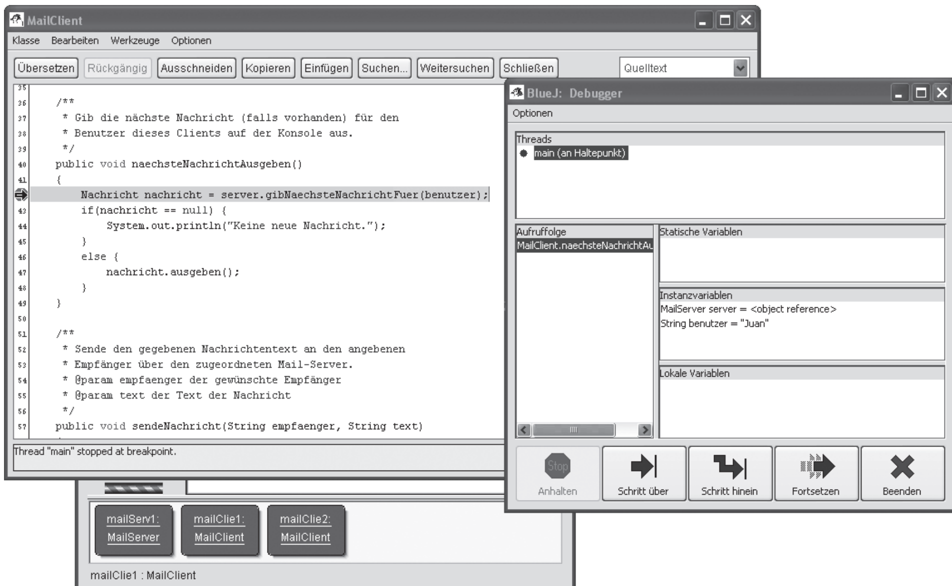

Abbildung 3.6: Das Debugger-Fenster, Ausführung gestoppt an Haltepunkt

Am unteren Rand des Debugger-Fensters befinden sich einige Kontrollknöpfe. Diese können benutzt werden, um die Ausführung eines Programms anzuhalten oder fortzusetzen. (Eine ausführlichere Erläuterung der Kontrollmöglichkeiten des Debuggers finden Sie in Anhang F.)

Auf der rechten Seite des Debugger-Fensters befinden sich drei Bereiche für die Anzeige von Variablen, benannt mit *statische Variablen, Instanzvariablen* und *lokale Variablen*. Wir werden den Bereich für statische Variablen vorläufig ignorieren. Statische Variablen werden erst später diskutiert, außerdem hat diese Klasse keine statischen Variablen.

Wir sehen, dass dieses Objekt zwei Instanzvariablen (oder Datenfelder) hat, `server` und `benutzer`, und wir können die aktuellen Werte dieser Variablen sehen. Die Variable `benutzer` hält die Zeichenkette "Juan", und die Variable `server` hält eine Referenz auf ein anderes Objekt. Die Objektreferenz haben wir in unseren Objektdiagrammen als Pfeil gezeichnet.

Beachten Sie, dass bis jetzt keine lokale Variable angezeigt wird. Dies liegt daran, dass die Ausführung immer vor der Zeile mit dem Haltepunkt anhält. Da die Zeile mit dem Haltepunkt die Deklaration der einzigen lokalen Variablen enthält und diese Zeile noch nicht ausgeführt wurde, existiert zu diesem Zeitpunkt noch keine lokale Variable.

Der Debugger ermöglicht uns nicht nur, ein laufendes Programm anzuhalten und die Variablen zu inspizieren, mit ihm kann man das Programm auch schrittweise weiter ausführen.

3.13.2 Einzelausführung

Wenn ein Haltepunkt erreicht wurde, führt ein Klick auf die Schaltfläche *Schritt über* eine Zeile aus und die Ausführung stoppt erneut.

Übung 3.37 Gehen Sie in der Ausführung der Methode `naechsteNachrichtAusgeben` einen Schritt weiter, indem Sie auf die Schaltfläche *Schritt über* klicken.

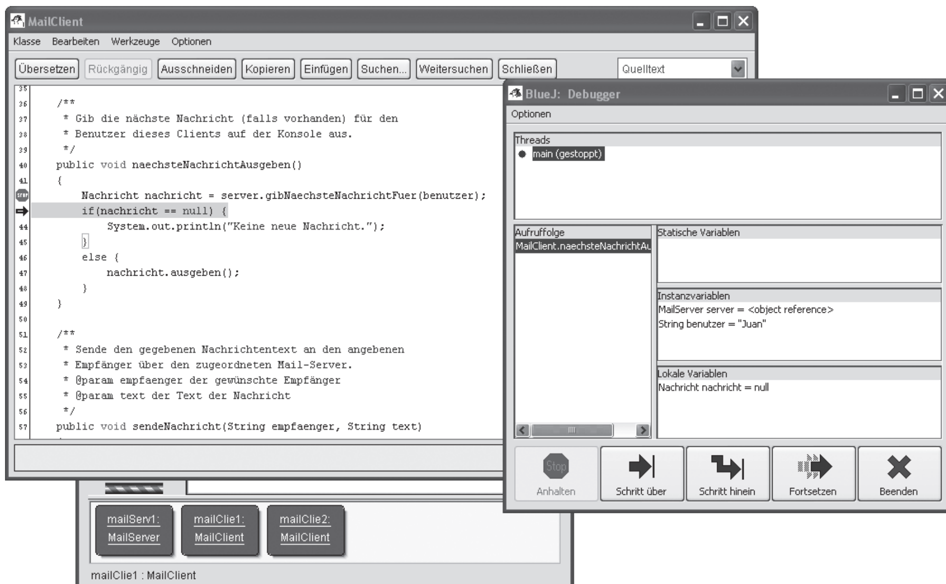

Abbildung 3.7: Erneut gestoppt nach einem einzelnen Schritt

In Abbildung 3.7 sehen Sie das Ergebnis der Ausführung der ersten Zeile von `naechsteNachrichtAusgeben`. Wir können erkennen, dass die Ausführung einen Schritt weitergegangen ist (ein kleiner schwarzer Pfeil neben der Quelltextzeile markiert die aktuelle Position) und dass im Bereich für lokale Variablen im Debugger-Fenster angezeigt wird, dass eine lokale Variable angelegt wurde und dieser Variablen ein Objekt zugewiesen wurde.

Übung 3.38 Sagen Sie voraus, welche Zeile als die nächste auszuführende Zeile markiert sein wird, wenn Sie einen Schritt weitergehen. Führen Sie diesen Schritt dann aus und überprüfen Sie Ihre Vorhersage. Lagen Sie richtig oder falsch? Erklären Sie, was passiert ist und weshalb es passiert ist.

Wir können uns nun durch wiederholtes Klicken auf die *Schritt über*-Schaltfläche an das Ende der Methode bewegen. Dies ermöglicht uns, den Ausführungspfad für diese Me-

thode nachzuvollziehen. Dies ist insbesondere bei bedingten Anweisungen interessant: Wir können genau erkennen, welcher Zweig einer if-Anweisung gewählt wird, und überprüfen, ob dies unseren Erwartungen entspricht.

Übung 3.39 Rufen Sie dieselbe Methode noch einmal auf. Bewegen Sie sich wie vorher durch die Methode. Was beobachten Sie? Erklären Sie dieses Verhalten.

3.13.3 Hineinschreiten in Methoden

Während wir uns durch die Methode `naechsteNachrichtAusgeben` schrittweise bewegt haben, haben wir zwei Methodenaufrufe an Objekte unserer Klassen gesehen. Die Zeile

```
Nachricht nachricht = server.gibNaechsteNachrichtFuer(benutzer);
```

enthält einen Aufruf der Methode `gibNaechsteNachrichtFuer`[1] des `server`-Objekts. Durch Prüfen der Instanzvariablen im Debugger-Fenster können wir sehen, dass das `server`-Objekt von der Klasse `MailServer` ist.

Die Zeile

```
nachricht.ausgeben();
```

ruft die Methoden `ausgeben` an der Nachricht auf. Wir können in der ersten Zeile der Methode `naechsteNachrichtAusgeben` sehen, dass `nachricht` von der Klasse `Nachricht` deklariert ist.

Mit der Benutzung der *Schritt über*-Funktion im Debugger haben wir Abstraktion eingesetzt: Wir haben die Methode `ausgeben` der Klasse `Nachricht` als eine einzelne Anweisung angesehen und konnten beobachten, dass ihr Effekt die Ausgabe der Details einer Nachricht (Absender, Empfänger und Inhalt) auf der Konsole war.

Wenn wir an mehr Details interessiert sind, können wir tiefer in den Ausführungsprozess hinabsteigen und auch die Methode `ausgeben` selbst Schritt für Schritt ausführen lassen. Dazu verwenden wir im Debugger statt der *Schritt über*-Funktion die Funktion *Schritt hinein*. *Schritt hinein* schreitet in die Methode, die aufgerufen wird, hinein und hält an der ersten Anweisung dieser Methode an.

Übung 3.40 Erzeugen Sie erneut die Ausgangsposition wie vorher, indem Sie eine Nachricht von Sophie an Juan schicken. Rufen Sie dann wieder die Methode `naechsteNachrichtAusgeben` am Mail-Client von Juan auf. Schreiten Sie durch die Methode wie zuvor. Aber dieses Mal klicken Sie, wenn Sie die Zeile

```
nachricht.ausgeben();
```

erreichen, nicht auf *Schritt über*, sondern auf *Schritt hinein*. Sorgen Sie dafür, dass die Konsole sichtbar ist, wenn Sie nun weiterschreiten. Was beobachten Sie? Erklären Sie es.

---

[1] Auf der CD als „gibNaechsteNachrichtFür“ bezeichnet; „ue“-Version auf der CWS zum Herunterladen.

WWW.PEARSON-STUDIUM.DE CWS COMPANION WEBSITE

3.14 Mehr zu Methodenaufrufen

In unseren Experimenten in Abschnitt 3.13 haben wir ein weiteres Beispiel von Objektinteraktion gesehen, das denen ähnlich ist, die wir schon kannten: Objekte rufen Methoden von anderen Objekten auf. In der Methode `naechsteNachrichtAusgeben` macht ein `MailClient`-Objekt einen Aufruf an ein `MailServer`-Objekt, um die nächste Nachricht abzuholen. Diese Methode (`gibNaechsteNachrichtFuer`) liefert einen Wert – ein Objekt der Klasse `Nachricht`. Außerdem gab es den Aufruf der Methode `ausgeben` an einer Nachricht. Mit Hilfe von Abstraktion können wir diesen Aufruf als eine einzelne Anweisung betrachten. Oder wir können, wenn wir an mehr Details interessiert sind, eine Abstraktionsstufe hinabsteigen und in die `ausgeben`-Methode hineinsehen.

Auf ähnliche Weise können wir den Debugger benutzen, um zu beobachten, wie ein Objekt ein anderes erzeugt. Die Methode `sendeNachricht` in der Klasse `MailClient` liefert ein gutes Beispiel. In dieser Methode wird ein Objekt der Klasse `Nachricht` in der ersten Zeile erzeugt:

```
Nachricht nachricht = new Nachricht(absender, empfaenger, text);
```

Der Grundgedanke ist, dass eine Nachricht eine Botschaft in einem Nachrichtensystem modelliert. Sie enthält Informationen über den Absender, den Empfänger und den Text der Nachricht. Wenn eine Botschaft versendet werden soll, dann kapselt ein Mail-Client diese Informationen in einem Objekt der Klasse `Nachricht` und legt diese auf dem Mail-Server ab. Dort kann sie später vom Mail-Client des Empfängers abgeholt werden.

In der Quelltextzeile oben sehen wir, wie das Schlüsselwort `new` für die Erzeugung eines neuen Objekts verwendet wird, und wir sehen, wie die Parameter an den Konstruktor übergeben werden. (Erinnern Sie sich: Bei der Konstruktion eines Objekts geschehen zwei Dinge – das Objekt wird erzeugt und der Konstruktor wird ausgeführt.) Der Aufruf eines Konstruktors ist dem Aufruf einer Methode sehr ähnlich. Dies können wir beobachten, wenn wir die Funktion *Schritt hinein* des Debuggers an der Stelle verwenden, an der das Objekt erzeugt wird.

Übung 3.41 Setzen Sie einen Haltepunkt in die erste Zeile der Methode `sendeNachricht` der Klasse `MailClient`. Rufen Sie dann diese Methode auf. Benutzen Sie die Funktion *Schritt hinein*, um in den Konstruktor der Klasse `Nachricht` zu schreiten. Im Debugger-Fenster können Sie nun die Instanzvariablen und die lokalen Variablen gleichen Namens sehen, wie wir sie für ein Nachricht-Objekt in Abschnitt 3.12.2 diskutiert haben. Schreiten Sie durch den Konstruktor und beobachten Sie, wie die Instanzvariablen initialisiert werden.

Übung 3.42 Benutzen Sie eine Kombination aus Lesen von Quelltext, Ausführen von Methoden, Haltepunkten und schrittweiser Ausführung, um sich mit den Klassen `Nachricht` und `MailClient` vertraut zu machen. Beachten Sie, dass Sie noch nicht genügend Informationen haben, um die Implementierung der Klasse `MailServer` zu verstehen, so dass Sie sie vorläufig ignorieren können. (Sie können natürlich trotzdem einen Blick darauf werfen, aber seien Sie nicht überrascht, wenn Sie sie etwas verwirrend finden …) Erklären Sie schriftlich, wie die Klassen `MailClient` und `Nachricht` miteinander interagieren. Zeichnen Sie Objektdiagramme als Teil Ihrer Erklärung.

3.15 Zusammenfassung

In diesem Kapitel haben wir diskutiert, wie man eine Aufgabe in Teilaufgaben zerlegen kann. Wir können versuchen, die Subkomponenten in den Objekten zu identifizieren, die wir modellieren wollen, und wir können diese Subkomponenten in unabhängigen Klassen implementieren. Dieses Vorgehen hilft uns, die Komplexität bei der Implementierung großer Anwendungen zu reduzieren, da es uns ermöglicht, die Klassen unabhängig voneinander zu implementieren, zu testen und zu warten.

Wir haben gesehen, dass ein solches Vorgehen zu Strukturen von Objekten führt, die gemeinsam eine Aufgabe erledigen. Objekte können andere Objekte erzeugen und sie können die Methoden anderer Objekte aufrufen. Ein Verständnis dieser Objektinteraktionen ist grundlegend sowohl für die Planung und Umsetzung von Anwendungen als auch für die Fehlersuche in ihnen.

Wir können handgezeichnete Diagramme, das Lesen von Quelltext und Debugger verwenden, um das Verhalten einer Anwendung zu verstehen und um Fehler zu finden.

NEUE BEGRIFFE IN DIESEM KAPITEL

Abstraktion, Modularisierung, Teile und herrsche, Klassendiagramm, Objektdiagramm, Objektreferenz, Überladen, interner Methodenaufruf, externer Methodenaufruf, Punkt-Notation, Debugger, Haltepunkt

ZUSAMMENFASSUNG DER KONZEPTE

- **Abstraktion** Abstraktion ist die Fähigkeit, Details von Bestandteilen zu ignorieren, um den Fokus der Betrachtung auf eine höhere Ebene lenken zu können.
- **Modularisierung** Modularisierung ist der Prozess der Zerlegung eines Ganzen in wohl definierte Teile, die getrennt erstellt und untersucht werden können und die in wohl definierter Weise interagieren.
- **Klassen definieren Typen** Ein Klassenname kann als Typname in einer Variablendeklaration verwendet werden. Variablen, die als Typ eine Klasse haben, können Objekte dieser Klasse halten.
- **Klassendiagramm** Ein Klassendiagramm zeigt die Klassen einer Anwendung und die Beziehungen zwischen diesen Klassen. Es liefert Informationen über den Quelltext. Es präsentiert eine statische Sicht auf ein Programm.
- **Objektdiagramm** Ein Objektdiagramm zeigt die Objekte und ihre Beziehungen zu einem bestimmten Zeitpunkt während der Ausführung einer Anwendung. Es präsentiert eine dynamische Sicht auf ein Programm.
- **Objektreferenz** Variablen von Objekttypen speichern Referenzen auf Objekte.

- **Primitiver Typ** Die primitiven Typen in Java sind die Typen, die keine Objekttypen sind. Die gebräuchlichsten primitiven Typen sind `int`, `boolean`, `char`, `double` und `long`. Primitive Typen haben keine Methoden.
- **Objekterzeugung** Objekte können andere Objekte mit dem `new`-Operator erzeugen.
- **Überladen** Eine Klasse kann mehr als einen Konstruktor oder mehr als eine Methode mit dem gleichen Namen enthalten, solange jede von ihnen einen unterscheidbaren Satz von Parametertypen definiert.
- **Interner Methodenaufruf** Methoden können andere Methoden der eigenen Klasse als Teil ihrer Implementierung aufrufen. Dies wird als interner Methodenaufruf bezeichnet.
- **Externer Methodenaufruf** Methoden können Methoden von anderen Objekten über die Punkt-Notation aufrufen. Dies wird als externer Methodenaufruf bezeichnet.
- **Debugger** Ein Debugger ist ein Softwarewerkzeug, mit dessen Hilfe die Ausführung eines Programms untersucht werden kann. Es kann benutzt werden, um Fehler (Bugs) zu finden.

Übung 3.43 Benutzen Sie den Debugger, um das Projekt *Zeitanzeige* zu untersuchen. Setzen Sie Haltepunkte in den Konstruktor von `Uhrenanzeige` und in jede der Methoden und führen Sie sie dann Schritt für Schritt aus. Verhält sich das Projekt so, wie Sie es erwarten? Konnten Sie neue Erkenntnisse gewinnen? Wenn ja, welche?

Übung 3.44 Benutzen Sie den Debugger, um die Methode `geldEinwerfen` des Projekts *Besserer-Ticketautomat* aus Kapitel 2 zu untersuchen. Führen Sie Tests durch, in denen beide Zweige der if-Anweisung ausgeführt werden.

Übung 3.45 Fügen Sie eine Betreffzeile für eine E-Mail in eine Nachricht aus dem Projekt *Mail-System* ein. Lassen Sie auch bei der Ausgabe auf der Konsole die Betreffzeile mit ausgeben. Passen Sie den Mail-Client entsprechend an.

Übung 3.46 Gegeben sei die folgende Klasse (nur auszugsweise angegeben):

```
public class Bildschirm
{
   public Bildschirm(int xAufloesung, int yAufloesung)
   { ... }
   public int anzahlBildpunkte()
   { ... }
   public void loeschen(boolean invertieren)
   { ... }
}
```

Schreiben Sie einige Zeilen Java-Code, die ein `Bildschirm`-Objekt erzeugen und dann seine Methode `loeschen` nur genau dann aufrufen, wenn die Anzahl der Bildpunkte größer als 2 Millionen ist (kümmern Sie sich nicht darum, ob das logisch ist – das Ziel ist lediglich, dass etwas syntaktisch Korrektes aufgeschrieben wird, also etwas, das sich von einem Compiler übersetzen lassen könnte).