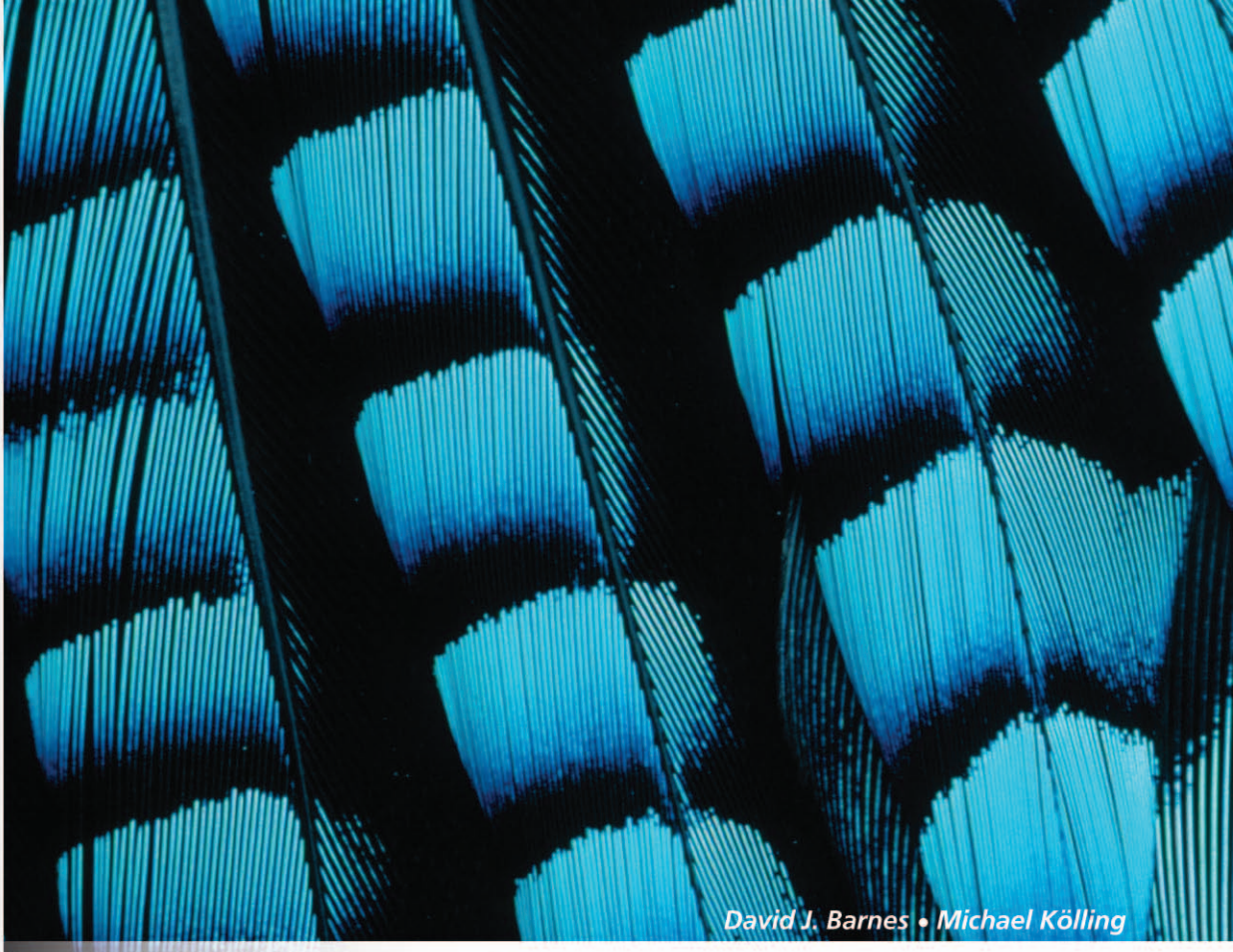


David J. Barnes • Michael Kölling

Java lernen mit BlueJ

Eine Einführung in die objektorientierte Programmierung

5. Auflage

ALWAYS LEARNING

PEARSON

Bibliografische Information der Deutschen Nationalbibliothek
Die Deutsche Nationalbibliothek verzeichnet diese Publikation in der Deutschen Nationalbibliografie; detaillierte bibliografische Daten sind im Internet über http://dnb.dnb.de abrufbar.

Authorized translation from the English language edition, entitled OBJECTS FIRST WITH JAVA – A PRACTICAL INTRODUCTION USING BLUEJ, 5th Edition by DAVID BARNES and MICHAEL KÖLLING, published by Pearson Education, Inc, publishing as Prentice Hall, Copyright © 2012 Pearson Education Inc.

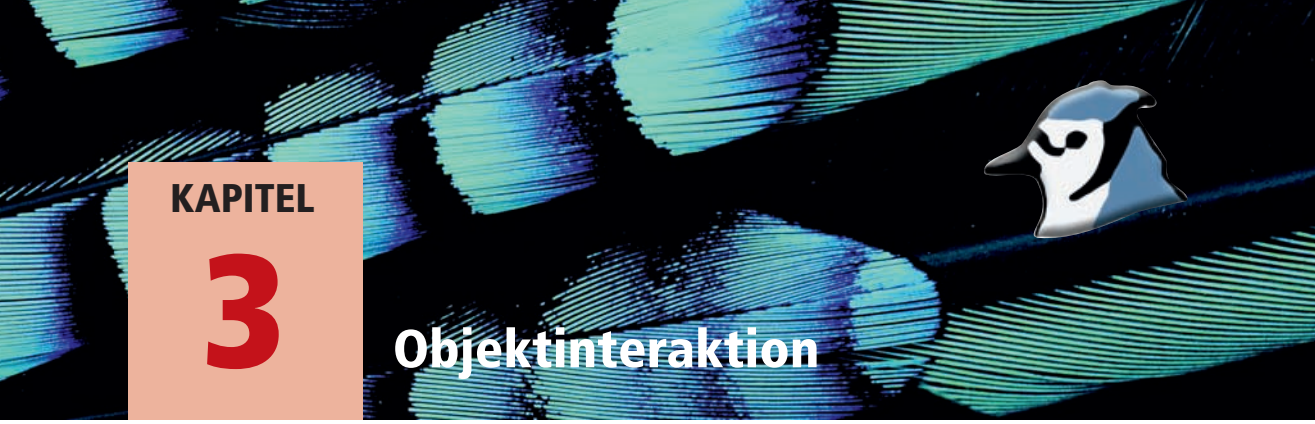
GERMAN language edition published by PEARSON DEUTSCHLAND GMBH, Copyright © 2013.

10 9 8 7 6 5 4 3 2 1

15 14 13

ISBN 978-3-86894-907-0 Print; 978-3-86326-957-9 PDF; 978-3-86326-907-4 ePUB

Programmleitung: Birger Peil, bpeil@pearson.de
Fachlektorat: Prof. Dr. Carsten Schulte, Freie Universität Berlin
Übersetzung: Petra Alm, Saarbrücken
Korrektorat: Katharina Pieper, Berlin, pieper.katharina@googlemail.com
Einbandgestaltung: Thomas Arlt, tarlt@adesso21.net
Herstellung: Claudia Bäurle, cbaeurle@pearson.de
Satz: Nadine Krumm, mediaService, Siegen (www.mediaservice.tv)
Druck- und Verarbeitung: Drukarnia Dimograf, Bielsko-Biala
Printed in Poland



KAPITEL

3 Objektinteraktion

Lernziele

Zentrale Konzepte in diesem Kapitel: Abstraktion, Objektdiagramme, Modularisierung, Methodenaufrufe, Objekterzeugung, 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.

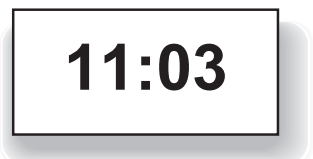


Abbildung 3.1
Die Anzeige einer Digitaluhr.

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 zerlegen 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 wohldefinierte Teile, die getrennt erstellt und untersucht werden können und die in wohldefinierter 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. Anstatt ein Fahrzeug als einen großen, monolithischen Block zu implementieren, würden wir zuerst getrennte Objekte für den Motor, das Getriebe, einen Sitz etc. 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 Zeitanzeige 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.

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.

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*.

```
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.

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.

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.

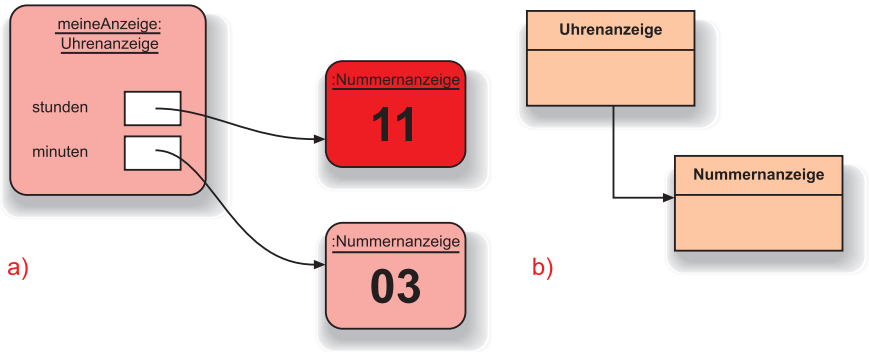


Abbildung 3.3
Objektdiagramm und Klassendiagramm für eine Uhrenanzeige.

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.

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).

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`.

Konzept

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

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.

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.

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.

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).

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

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.

Listing 3.3
Der Quelltext der Klasse **Nummernanzeige**.

```
/**
 * 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 31.07.2011
 */
```

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

    /**
     * Konstruktor für Exemplare der Klasse Nummernanzeige.
     * Setzt das Limit, bei dem die Anzeige zurückgesetzt wird.
     */
    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;
        }
        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;
    }
}
```

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;
    }
}
```

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 Zuweisung 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.

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.

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 2.9 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 Zei-

chenketten, 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 usw.), 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.

Listing 3.4
Der Quelltext der Klasse **Uhrenanzeige**.

```
/**
 * 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 31.07.2011
 */
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
     * weiter gestellt 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)
    {
        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();
    }
}
```

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 entspre-

chend auf ihrer Anzeige erzeugen. Diese Zeichenkette dient uns also als Softwaresimulation einer echten Uhrenanzeige.[1]

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.

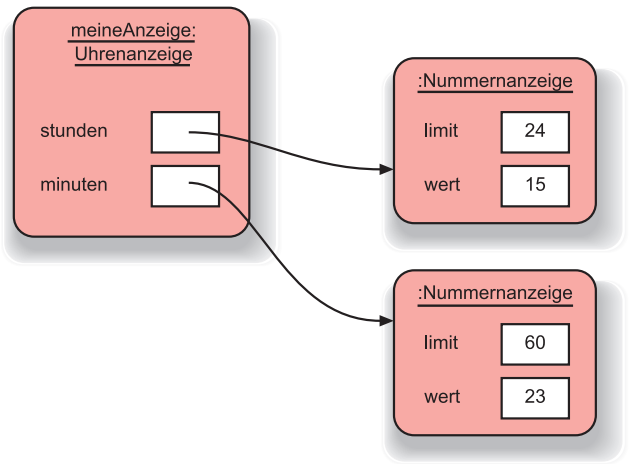


Abbildung 3.4
Ein Objektdiagramm der Uhrenanzeige.

1 Der Ordner *Projekte* umfasst eine Version dieses Projekts mit einer einfachen grafischen Benutzeroberfläche (GUI) namens *Zeitanzeige-GUI*. Der interessierte Leser kann mit diesem Projekt experimentieren, auch wenn es nicht in diesem Buch besprochen wird.

Copyright